Prácticas de ordenador con wxMaxima

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

ugr Universidad de Granada

24 septiembre 2014

Índice

Introducción

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

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

Por qué

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

Siempre hay que intentar escoger la herramienta que mejor se adapte al problema que se presenta y, en nuestro caso, *Maxima* cumple con creces las necesidades de un curso de Cálculo. Es evidente que *Mathematica* o *Maple* también pero creemos que el uso de programas de software libre permite al alumno y al profesor estudiar cómo está hecho, ayudar en su mejora y, si fuera necesario y posible, adaptarlo a sus propias necesidades.

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

Dónde y cómo

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

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

En esta página puedes descargarte el programa y encontrar abundante documentación sobre cómo instalarlo. Al momento de escribir estas notas, en dicha página puedes encontrar versiones

listas para funcionar disponibles para entornos Windows y Linux e instrucciones detalladas para ponerlo en marcha en Mac OS X.

wxMaxima

En estas notas no estamos usando *Maxima* directamente sino un entorno gráfico que utiliza *Maxima* como motor para realizar los cálculos. Este entorno (programa) es *wxMaxima*. Nos va a permitir que la curva de aprendizaje sea mucho más suave. Desde el primer día el alumno será capaz de realizar la mayoría de las operaciones básicas. A pesar de ello, en todos los ejemplos seguimos utilizando la notación de *Maxima* que es la que le aparece al lector en pantalla y que nunca está de más conocer. *wxMaxima* se puede descargar de su página web

http://wxmaxima.sourceforge.net/

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

Granada a 10 de septiembre de 2008

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):[1]

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

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

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

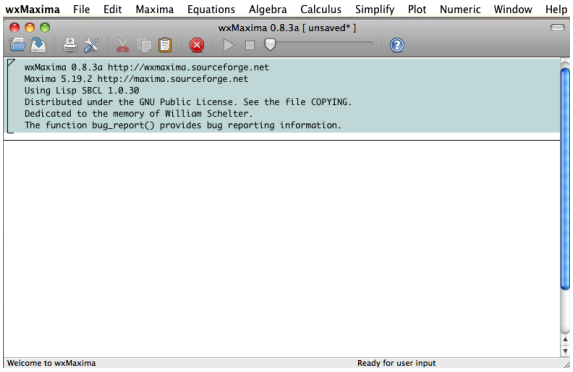


Figura 1.1 Ventana inicial de *wxMaxima*

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.

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

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

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

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

Operaciones básicas

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

El producto se indica con "∗":

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

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

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

También podemos dividir

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

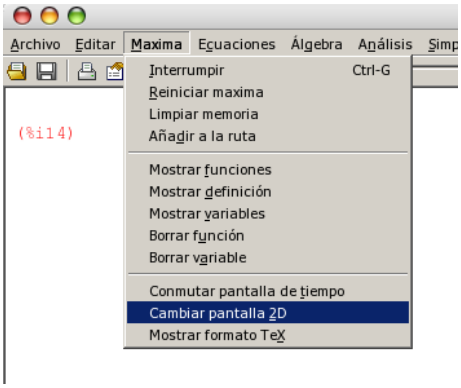
eso sí, teniendo cuidado con la precedencia de las operaciones. En estos casos el uso de paréntesis es obligado.

Podemos escribir potencias

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

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

```
(%i7)  3^1000;
                                    1322070819480806636890455259
(%o7)  5[418 digits]614366132173102
                                    768902855220001
```



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;
                                    1322070819480806636890455259752144365965422032752148167664
                                    9203682268285973467048995407783138506080619639097776968725
                                    8235595095458210061891186534272525795367402762022519832080
                                    3878014774228964841274390400117588618041128947815623094438
(%o9)  0615661730540866744905061781254803444055470543970388958174
                                    6536825491613622083026856377858229022841639830788789691855
                                    6404084898937609373242171846359938695516765018940588109060
                                    4260896714388641028143503856487471658320106143661321731027
                                    68902855220001
```

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;
                                    1322070819480806636890455259 75[418 digits]61436613217310
(%o11)
                                    2768902855220001
```

Observación 1.2. Antes de seguir, ¿por qué hemos escrito $ y no punto y coma al final de la salida anterior? El punto y coma sirve para terminar un comando o separar varios de ellos. El

dólar, $, también termina un comando o separa varios de ellos, pero, a diferencia del punto y coma, *no* muestra el resultado en pantalla.

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

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

simplificando cuando sea posible

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

Cálculo simbólico

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

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

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

Raíces

```
(%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.

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

Podemos operar con ellas como con cualquier otro número.

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

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

¿Cuál era el resultado anterior?

`%`	último resultado
`%inúmero`	entrada *número*
`%onúmero`	resultado *número*

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

```
(%i19)   %o15
(%o19)   √5
```

además podemos usar esa información como cualquier otro dato.

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

1.2 Resultados exactos y aproximación decimal

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

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

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

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

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

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)  0.83333333333333
(%i26)  numer:false$
```

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

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

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

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

En la ayuda de *Maxima* podemos leer

> `Valor por defecto: 16.`
> `La variable ‘fpprec’ guarda el número de dígitos significativos en la aritmética con números decimales de punto flotante grandes ("bigfloats"). La variable ‘fpprec’ no afecta a los cálculos con números decimales de punto flotante ordinarios.`

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

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

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

```
(%i30)  bfloat(%pi);
(%o30)  3.14159265358979323846264338327950288419716939937510582
        25342117068b0
(%i31)  set_display('ascii)$
(%i32)  bfloat(%pi);
(%o32)  3.14159265358979323846264338327950288419716939937510582
        0974944592307816406286208998628034825342117068b0
```

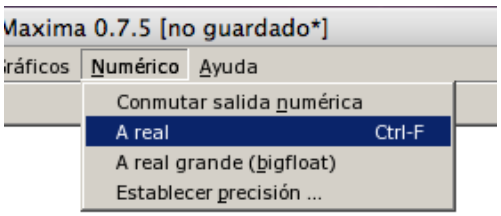


Figura 1.2 En el menú, la pestaña **Numérico** permite obtener la expresión decimal con la precisión que se desee

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

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

1.3 Funciones usuales

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

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

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

Potencias, raíces y exponenciales

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

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

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

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

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

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

Logaritmos

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

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

y si lo que nos interesa es su expresión decimal

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

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

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

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

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

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

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

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

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

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

Funciones trigonométricas e hiperbólicas

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

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

Por defecto, las funciones trigonométricas están expresadas en radianes.

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

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

De forma análoga, puedes utilizar las correspondientes funciones hiperbólicas.

Otras funciones

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

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

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

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

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

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

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

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

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

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

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

1.4 Operadores lógicos y relacionales

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

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

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

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

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

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

```
(%i55)  is((x+1)^2=x^2+2*x+1);
(%o55)  false
```

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

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

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

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

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

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

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

1.5 Variables

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

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

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

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

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

```
(%i63)  largo:10;
(%o63)  10
(%i64)  ancho:7;
(%o64)  7
(%i65)  largo*ancho;
(%o65)  70
```

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

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

para luego poder usarlas.

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

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

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

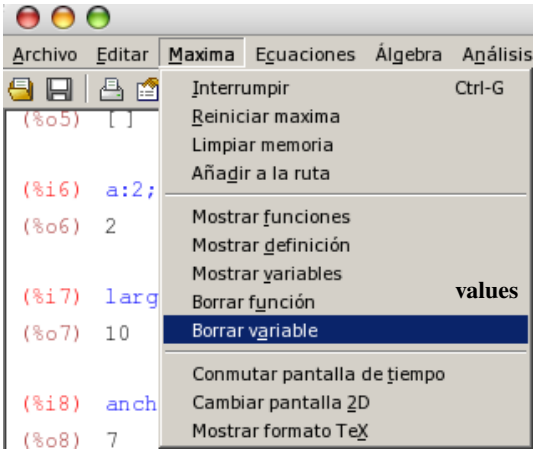
```
(%i68)  diff(sin(x),x);
        Non-variable 2nd argument to diff:
        3
        - an error. To debug this try debugmode(true);
```

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

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

```
(%o69)   cos(x)
```

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



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

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

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

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

o todas.

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

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

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

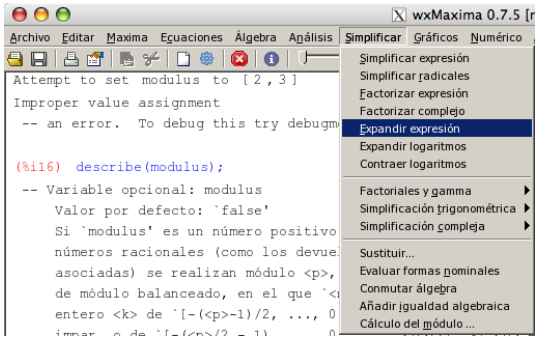
Una de las pequeñas diferencias entre `kill` y `remvalue` es que la primera no comprueba si la variable, o lo que sea, estaba previamente definida y siempre responde *done*. Existe también la posibilidad de borrar *todo*:

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

```
(%o0)   done
```

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

1.6 Expresiones simbólicas



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

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

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

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

Aunque parece que no siempre obtenemos el resultado esperado

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

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

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

1.6.1 Desarrollo de expresiones simbólicas

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

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

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

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

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

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

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

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

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

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

y cocientes.

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

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

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

num **denom**

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

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

Comportamiento de expand

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

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

Si quisiéramos desarrollar la función

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

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

```
(%i13)  expand((x+1)^100+(x-3)^32+(x+2)^2+x-1-1/x+2/((x-1)^2)
          +1/((x-7)^15),3,4);
(%o13)  2/(x^2-2x+1)+( x+1)^100+x^2+5x-1/x+(x-3)^32+1/(x-7)^15+3
```

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

```
(%i14)  log(a*b);
(%o14)  log(a b)
```

```
(%i15)  sqrt(x*y)
(%o15)  sqrt(x y)
```

Cuando cambiamos su valor a `all`,

```
(%i16)  radexpand:all$ logexpand:all$
(%i17)  log(a*b);
(%o17)  log(a)+log(b)
(%i18)  sqrt(x*y)
(%o18)  sqrt(x)sqrt(y)
```

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

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

Factorización

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

factor

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

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

como con expresiones polinómicas como las anteriores

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

El número de variables que aparecen tampoco es un problema:

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

Evaluación de valores en expresiones

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

Ahora que hemos estado trabajando con expresiones polinómicas, para evaluar en un punto pode-
ev mos 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);
                          x^2+x-2
(%o28)                  ---------
                        x^2-6*x+9
(%i29)  partfrac(p/q,x);
                            7          10
(%o29)                    ----- + ---------- + 1
                          x - 3    x - 3^2
```

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

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

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

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

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

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

1.6.3 Expresiones trigonométricas

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

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

Por ejemplo,

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

```
(%o34)  -(cos(x)sin(y)-sin(x)cos(y))(cos(z)^2-sin(z)^2)+cos(x)sin(3y)+
        sin(x)cos(3y)
(%i35)  trigexpand(8*sin(2*x)^2*cos(x)^3);
(%o35)  32 cos(x)^5 sin(x)^2
```

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

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

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

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

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

$$(\%o39)\quad \frac{\cos(2x)+1}{2}+\cos(2x)$$

```
(%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);
```

$$(\%o42)\quad \cosh(y+x)+\frac{\cosh(2x)-1}{2}$$

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.

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

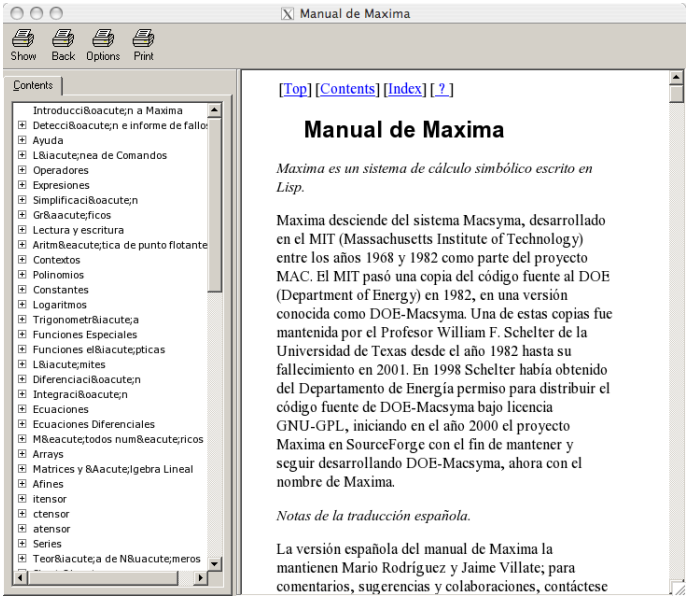


Figura 1.3 Ayuda de *wxMaxima*

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

describe

```
(%i43)  describe(dependencies);  - Variable del sistema: dependen-
        cies
        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 'de-
        pends' o
        'gradef' añade elementos adicionales.
        Véanse 'depends' y 'gradef'.
```

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

```
(%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 estadísti-
         cos)
         3: boxplot (Funciones y variables para gráficos estadísti-
         cos)
         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 esta-
         dísticos)
         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.

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

```
(%i46)   apropos("plot");
```

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

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

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

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

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

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

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

1.8 Ejercicios

Ejercicio 1.1. Calcula

a) Los 100 primeros decimales del número e,

b) el logaritmo en base 3 de 164232032682606581462314678007092552 89.

c) el arcocoseno hiperbólico de 1,

d) el seno y el coseno de i, y

e) el logaritmo de -2.

Ejercicio 1.2.

a) ¿Qué número es mayor 1000^{999} o 999^{1000}?

b) Ordena de mayor a menor los números π, $\frac{73231844868435875}{37631844868435563}$ y $\cosh(3)/3$.

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

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

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

Gráficos

2

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

Aunque sólo vamos a hablar de gráficos en dos dimensiones, hay que decir que se pueden realizar representaciones en tres dimensiones de manera análoga. En la ayuda de *Maxima* puedes encontrar todos los detalles.

2.1 Funciones

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

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

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

que se pueden utilizar como cualquier otra función.

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

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

```
(%i3)  g(x,y,z):=[2*x,3*cos(x+y)];
```

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

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

```
(%i5)   define(h(x,y,z),g(x,y,z)^2);
(%o5)   h(x,y,z):=[4x^2,9cos(y+x)^2]
```

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

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

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

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

pero teniendo cuidado de escribir el número correcto de variables

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

Esto plantea varias cuestiones muy relacionadas entre sí: cuando llevamos un rato trabajando y hemos definido varias funciones, ¿cómo sabemos cuales eran? y ¿cuál era su definición?. La lista de funciones que hemos definido se guarda en la variable `functions` a la que también puedes acceder desde el menú **Maxima→Mostrar funciones** de manera similar a como accedemos a la lista de variables. En el mismo menú, **Maxima→Borrar**

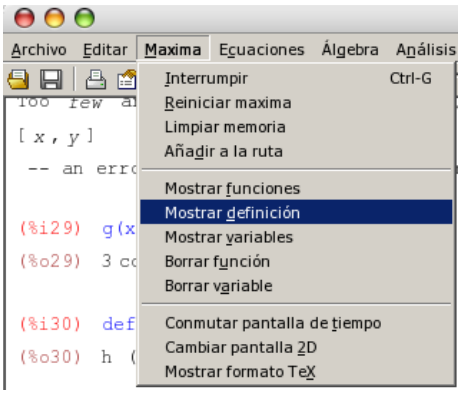


Figura 2.1 Desde el menú podemos consultar las funciones que tenemos definidas, cuál es su definición y borrar algunas o todas ellas

función tenemos la solución a cómo borrar una función (o todas). También podemos hacer esto con la orden `remfunction`. **remfunction**

```
(%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 **fundef**

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

que, si nos interesa, podemos borrar

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

o, simplemente, borrar todas las que tengamos definidas

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

Funciones definidas a trozos

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

a) definir una función para cada trozo con lo que tendremos que ocuparnos nosotros de ir escogiendo de elegir la función adecuada, o

b) utilizar una estructura `if-then-else` para definirla.[4]

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

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

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

la podemos definir de la siguiente forma utilizando el segundo método

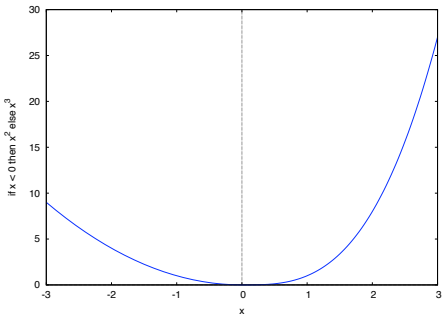
```
(%i13)  f(x):=if x< 0 then x^2 else x^3;
(%o13)  f(x):=if x< 0 then x^2 else x^3
```

y podemos evaluarla en un punto

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

o dibujarla

```
(%i16)  plot2d(f(x),[x,-3,3]);
(%o16)
```



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

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

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

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

y decidir nosotros cuál es la que tenemos que utilizar:

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

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

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

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

la podemos escribir como sigue anidando dos condicionales

```
(%i21)  g(x):=if x<=1 then x^2 else
          if x <= %pi then sin(x) else -x+1$
```

Comprobamos que la definición se comporta correctamente en un valor de cada intervalo

```
(%i22)  [g(-3),g(2),g(5)];
(%o22)  [9,sin(2),-4]
```

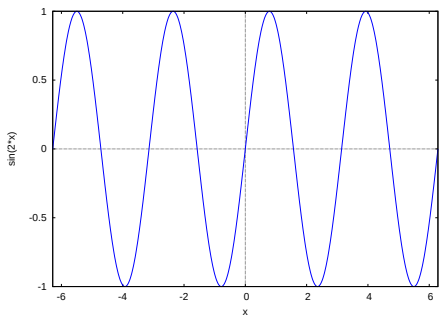
2.2 Gráficos en el plano con plot2d

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

`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 $[a,b]$

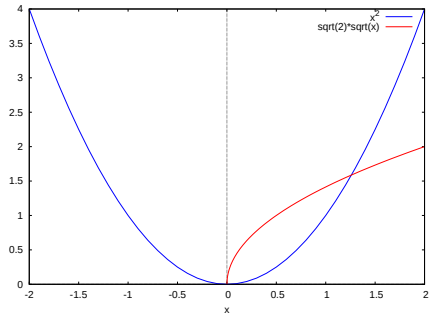
Podemos dibujar la gráfica de una función

```
(%i23)  plot2d(sin(2*x),[x,-2*%pi,2*%pi]);
(%o23)
```



o de varias

```
(%i24)  plot2d([x^2,sqrt(2*x)],[x,-2,2]);
(%o24)
```



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

Cuando accedemos a través del menú, aparece una ventana de diálogo con varios campos que podemos completar o modificar:

a) Expresión(es). La función o funciones que queramos dibujar. Por defecto, *wxMaxima* rellena este espacio con % para referirse a la salida anterior.

b) Variable *x*. Aquí establecemos el intervalo de la variable *x* donde queramos representar la función.

c) Variable *y*. Ídem para acotar el recorrido de los valores de la imagen.

d) Graduaciones. Nos permite regular el número de puntos en los que el programa evalúa una función para su representación.

Figura 2.2 Gráficos en 2D

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

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

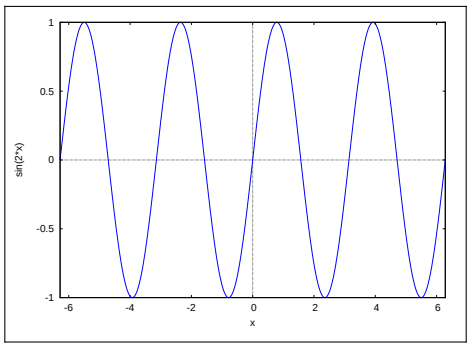
g) Gráfico al archivo. Guarda el gráfico en un archivo con formato Postscript.

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

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

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

(%t25)



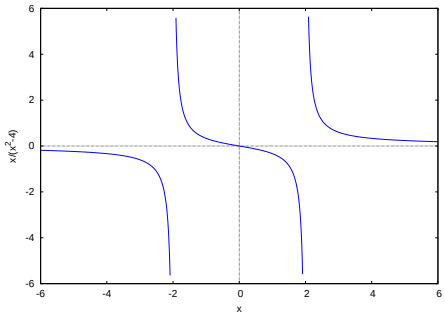
Es complicado representar una ventana separada en unas notas escritas así que, aunque no utilicemos `wxplot2d`, sí hemos representado todas las gráficas a continuación de la correspondiente orden.

wxplot2d

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

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

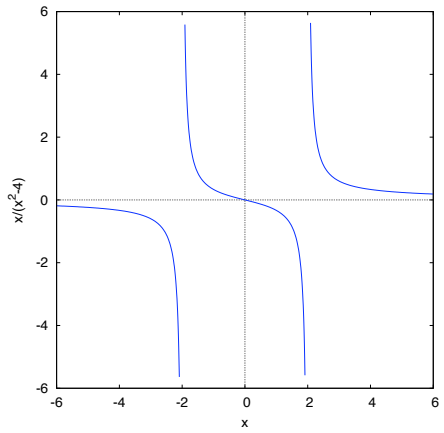
(%o26)



podemos cambiar la proporción entre ejes.

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

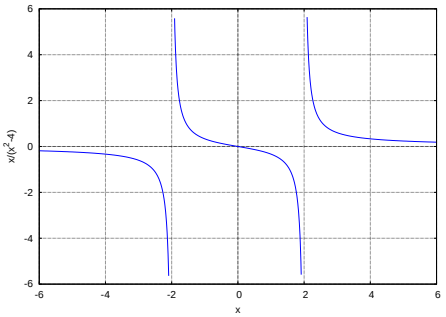
(%o27)



`set size ratio 1` dibuja ambos ejes con el mismo tamaño en pantalla, `set size ratio 2` o `set size ratio 0.5` dibuja el eje X el doble o la mitad de grande que el eje Y. O podemos añadir una malla que nos facilite la lectura de los valores de la función.

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

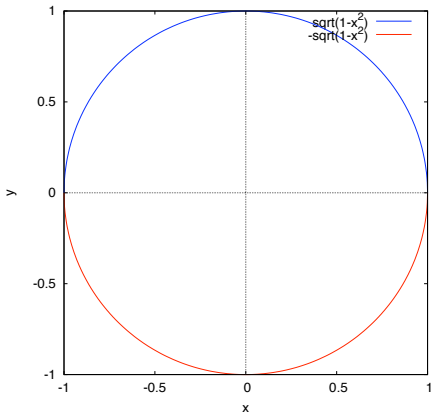
(%o28)



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

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

```
(%o29)
```



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

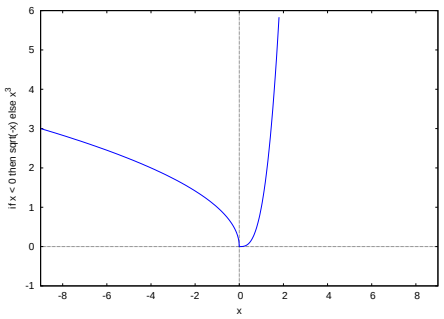
$$f(x) = \begin{cases} \sqrt{-x} & \text{si } x < 0 \\ x^3 & \text{si } x \geq 0. \end{cases}$$

Vamos, en primer lugar, a definirla:

```
(%i30)  f(x):= if x<0 then sqrt(-x) else x^3;
(%o30)  f(x):= if x<0 then √(-x) else x^3
```

y luego la representamos

```
(%i31)  plot2d(f(x),[x,-9,9],[y,-1,6],
          [gnuplot_preamble,"set zeroaxis;"])$
(%o31)
```



2.3 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. Este se hace de la siguiente forma

```
(%i32)  load(draw)$
```

`gr2d(`*opciones, objeto gráfico,...*`)`	gráfico dos dimensional
`draw(`*opciones, objeto gráfico,...*`)`	dibuja un gráfico
`draw2d(`*opciones, objeto gráfico,...*`)`	dibuja gráfico dos dimensional

El paquete `draw`, permite utilizar, entre otras, la orden `draw2d` para dibujar gráficos en dos dimensiones. 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)
    )
```

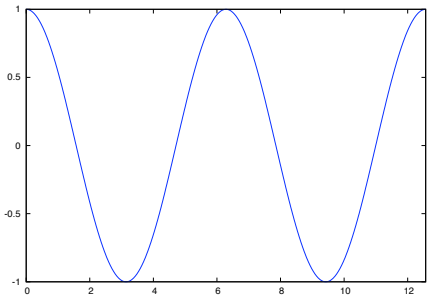
Las opciones son numerosas y permiten controlar prácticamente cualquier aspecto imaginable. Aquí comentaremos algunas de ellas pero la ayuda del programa es insustituible. En segundo lugar aparece el objeto gráfico. En este caso "`explicit(cos(t),t,0,2*%pi)`". Estos pueden ser de varios tipos aunque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

draw a) Si tenemos previamente definido el objeto, `draw(objeto)`, o bien,

draw2d b) `draw2d(definición del objeto)` si lo definimos en ese momento para dibujarlo.

Por ejemplo,

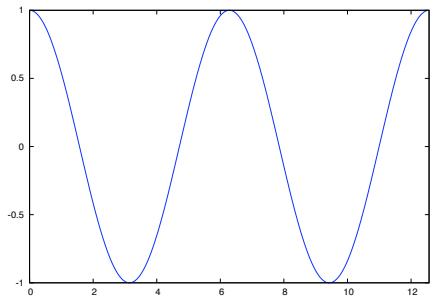
```
(%i33)  coseno:gr2d(
          color=blue,
          explicit(cos(x),x,0,4*%pi))$
(%i34)  draw(coseno);
(%o34)
```



da el mismo resultado que

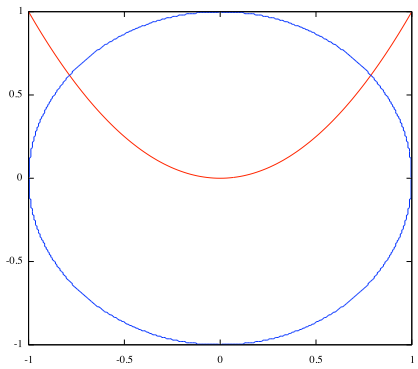
```
(%i35)  draw2d(
          color=blue,
          explicit(cos(x),x,0,4*%pi)
          );
```

```
(%o35)
```



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.

```
(%i36)  draw2d(
          color=red,
          explicit(x^2,x,-1,1),
          color=blue,nticks=60,
          implicit(x^2+y^2=1,x,-1,1,y,-1,1)
          );
(%o36)
```



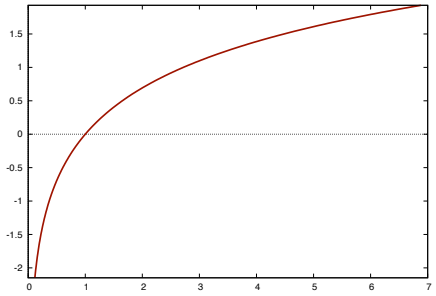
Vamos a comentar brevemente alguno de los objetos y de las opciones del módulo `draw`. Comenzamos con algunos de los objetos que podemos representar y, posteriormente, comentamos algunas opciones.

2.3.1 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`)`.

```
(%i37)  draw2d(
          color=dark-red,line_width=5,
          xaxis=true,yaxis=true,
          explicit(log(x),x,0,7));
```

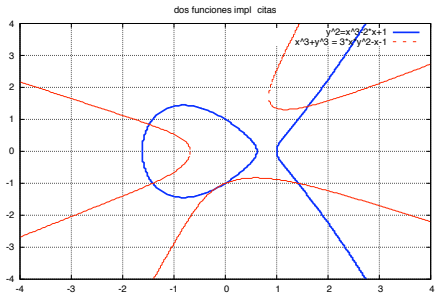
(%o37)



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

```
(%i38)  draw2d(
          grid=true,
          line_type=solid,
          color=blue,
          key="y^2=x^3-2x+1",
          implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4),
          line_type=dots,
          color=red,
          key="x^3+y^3 = 3xy^2-x-1",
          implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4),
          title="dos funciones implícitas");
```

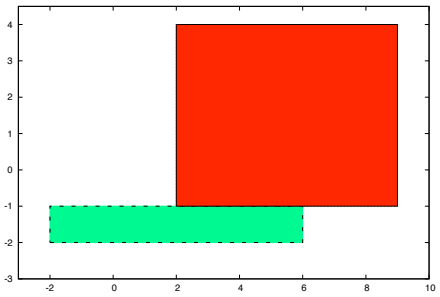
(%o38)



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

```
(%i39)  draw2d(line_width=6,
          line_type=dots,
          transparent=false,
          fill_color=spring-green,
          rectangle([-2,-2],[6,-1]),
          transparent=false,
          fill_color=red,
          line_type=solid,
          line_width=2,
          rectangle([9,4],[2,-1]),
          xrange=[-3,10],
          yrange=[-3,4.5]);
```

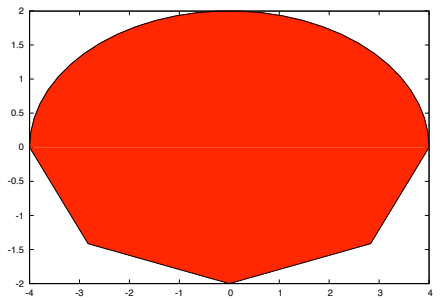
(%o39)



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.

```
(%i40)  draw2d(
          ellipse(0,0,4,2,0,180),
          nticks = 5,
          ellipse(0,0,4,2,180,360));
```

(%o40)

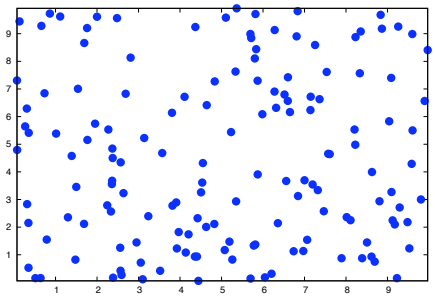


La parte superior de la elipse se ha dibujado utilizando 30 puntos y la inferior únicamente 5.

points: permite representar una lista de puntos en el plano. Las coordenadas de la lista las escribimos de la forma[5] `[[x1,y1],[x2,y2],...]`.

```
(%i41)  draw2d(
          color=blue,
          point_type=filled_circle,
          point_size=2,
          points(makelist([random(10.0),random(10.0)],k,1,150)));
```

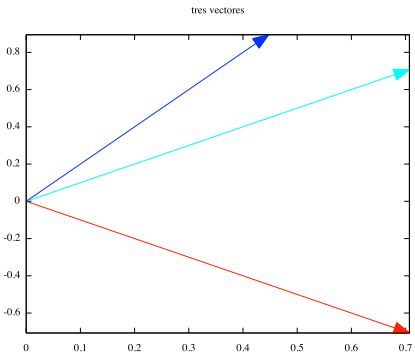
(%o41)



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

```
(%i42)  draw2d(
          head_length = 0.03, head_angle=20,
          color=cyan, vector([0,0],[1,1]/sqrt(2)),
          color=red, vector([0,0],[1,-1]/sqrt(2)),
          color=blue, vector([0,0],[1,2]/sqrt(5)),
          title="tres vectores");
```

(%o42)



[5] En el ejemplo usamos 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.

En la ayuda puedes encontrar varios opciones sobre el aspecto como se representan los vectores. Nosotros hemos usado `head_length` y `head_angle` para el tamaño de la punta de la fecha de los vectores.

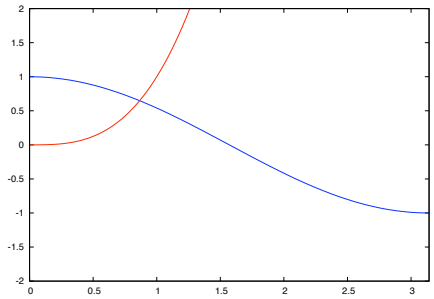
2.3.2 Opciones

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

Opciones globales

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

```
(%i43)  draw2d(color=blue,
          explicit(cos(x),x,0,4*%pi),
          color=red,
          explicit(x^3,x,-5,5),
          xrange=[0,%pi],yrange=[-2,2])$
(%o43)
```

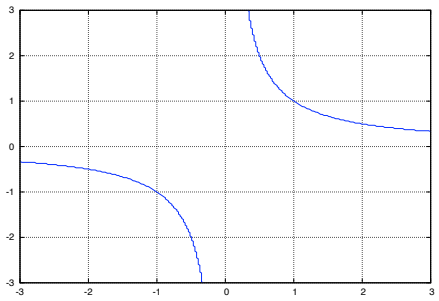


Si en el ejemplo anterior no limitamos el rango a representar, al menos en la coordenada y, es difícil poder ver a la vez la función coseno que toma valores entre 1 y -1 y la función x^3 que en 5 vale bastante más.

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

```
(%i44)  draw2d(
          color=blue,nticks=100,
          implicit(x*y=1,x,-3,3,y,-3,3),
          grid=true)$
```

(%o44)

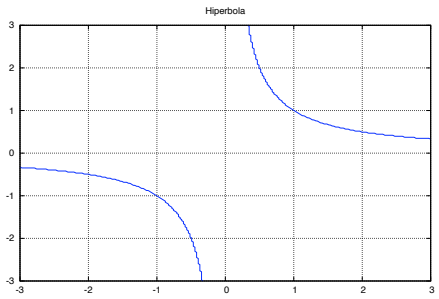


Acabamos de dibujar la hipérbola definida implícitamente por la ecuación $xy = 1$. La opción `grid` nos ayuda a hacernos una idea de los valores que estamos representando.

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

```
(%i45)  draw2d(
          color=blue,
          nticks=100,
          implicit(x*y=1,x,-3,3,y,-3,3),
          grid=true,
          title="Hiperbola"
        )$
```

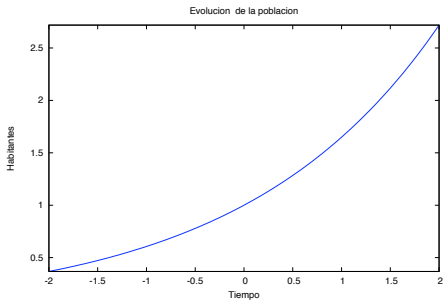
(%o45)



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

```
(%i46)  draw2d(color=blue,
          explicit(exp(x/2),x,-2,2),
          xlabel="Tiempo",
          ylabel="Habitantes",
          title="Evolucion de la poblacion");
```

(%o46)



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

Opciones locales

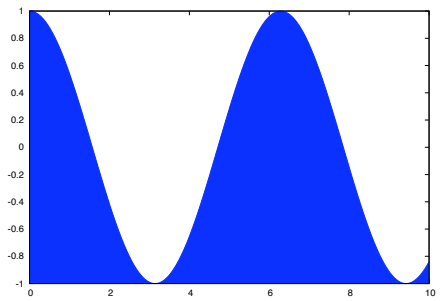
point_size: tamaño al que se dibujan los puntos. Su valor por defecto es 1. Afecta a los objetos de tipo `point`.

point_type: indica cómo se van a dibujar los puntos . El valor para esta opción puede ser un nombre o un número: none (-1), dot (0),plus (1), multiply (2), asterisk (3), square (4), filled_square (5), circle (6), filled_circle (7), up_triangle (8), filled_up_triangle (9), down_triangle (10), filled_down_triangle (11), diamant (12) y filled_diamant (13). Afecta a los objetos de tipo `point`.

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

```
(%i47)  draw2d(fill_color=blue,
          filled_func=true,
          explicit(cos(x),x,0,10)
        ):
```

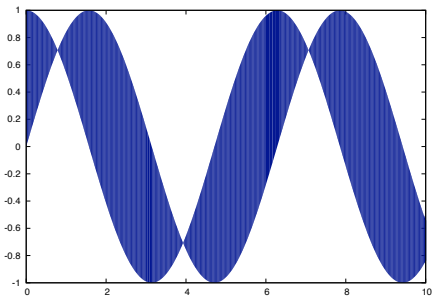
(%o47)



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

```
(%i48)  draw2d(
          filled_func=sin(x),
          fill_color=navy,
          explicit(cos(x),x,0,10));
```

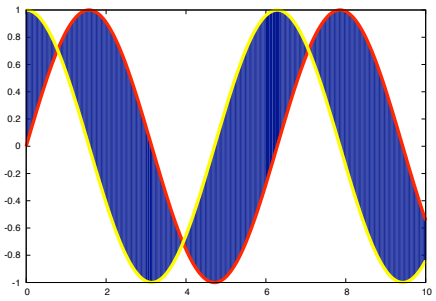
(%o48)



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:

```
(%i49)  draw2d(
          filled_func=sin(x),fill_color=navy,
          explicit(cos(x),x,0,10),
          filled_func=false,color=red,line_width=10,
          explicit(sin(x),x,0,10),
          color=yellow,line_width=10,
          explicit(cos(x),x,0,10));
```

(%o49)



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"’,
```

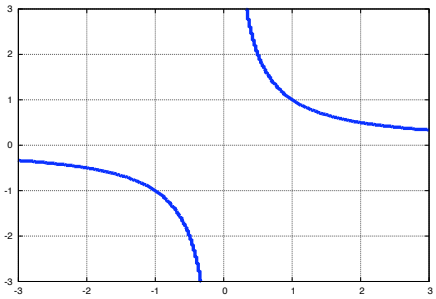
```
‘"light-cyan"’, ‘"dark-cyan"’, ‘"magenta"’, ‘"light-magenta"’,
‘"dark-magenta"’, ‘"turquoise"’, ‘"light-turquoise"’,
‘"dark-turquoise"’, ‘"pink"’, ‘"light-pink"’, ‘"dark-pink"’,
‘"coral"’, ‘"light-coral"’, ‘"orange-red"’, ‘"salmon"’,
‘"light-salmon"’, ‘"dark-salmon"’, ‘"aquamarine"’, ‘"khaki"’,
‘"dark-khaki"’, ‘"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.

```
(%i50)  draw2d(
          color=blue,line_width=10,nticks=100,
          implicit(x*y=1,x,-3,3,y,-3,3),
          grid=true,
          )$
```

(%o50)

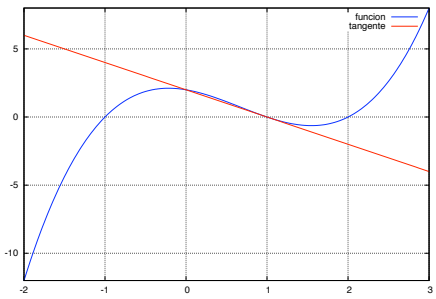


nticks: número de puntos que se utilizan para calcular los dibujos. Por defecto es 30. Un número mayor aumenta el detalle del dibujo aunque a costa de un mayor tiempo de cálculo y tamaño del fichero (si se guarda). Sólo afecta a los objetos de tipo `ellipse`, `explicit`, `parametric`, `polar` y `parametric`.

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

```
(%i51)  draw2d(color=blue,key="función",explicit(f(x),x,-2,3),
          color=red,key="tangente",explicit(tangente(x,1),x,-2,3),
          grid=true);
```

(%o51)



2.4 Animaciones gráficas

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

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

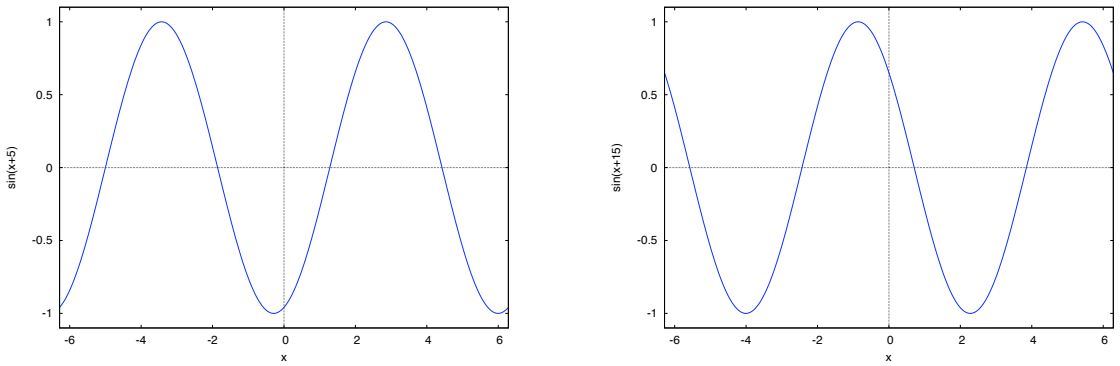
`with_slider`	animación de `plot2d`
`with_slider_draw`	animación de `draw2d`

Tenemos dos posibilidades para construir animaciones dependiendo de si queremos que *Maxima* utilice `plot2d` o `draw2d`. 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.

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

```
(%i52)  with_slider(n,
          makelist(i,i,1,20),
          sin(x+n),
          [x,-2*%pi,2*%pi],
          [y,-1.1,1.1]);
```

En la Figura 2.3 tienes la representación de algunos valores.



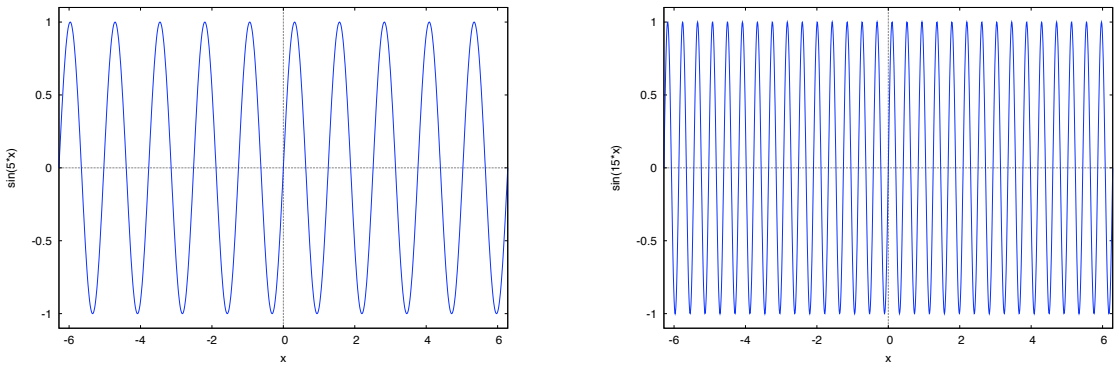
$n = 5$ $n = 15$

Figura 2.3 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.

```
(%i53)  with_slider(n,makelist(i,i,1,20),sin(x*n),
          [x,-2*%pi,2*%pi],[y,-1.1,1.1]);
```

Puedes ver en la Figura 2.4 puedes ver cómo aumenta la frecuencia con n.



$n = 5$ $n = 15$

Figura 2.4 sen($n\,x$)

Si en lugar de `plot2d`, utilizamos el módulo `draw` para diseñar los dibujos, tenemos que usar `with_slider_draw`. De nuevo, en primer lugar va el parámetro, después, una lista que indica los valores que tomará el parámetro y el resto debe ser algo aceptable por la orden `draw`. 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

```
(%i54)  with_slider_draw(
          t,makelist(%pi*i/10,i,1,20),
          parametric(cos(x),sin(x),x,0,t),
          xrange=[-1,1],
          yrange=[-1,1],
          user_preamble="set size ratio 1")$
```

En la Figura 2.5 tenemos representados algunos pasos intermedios

El tipo de objeto "parametric" no lo hemos comentado en los apartados anteriores. Nos permite representar la gráfica de una curva en el plano. En la ayuda de *Maxima* puedes encontrar más detalles.

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.

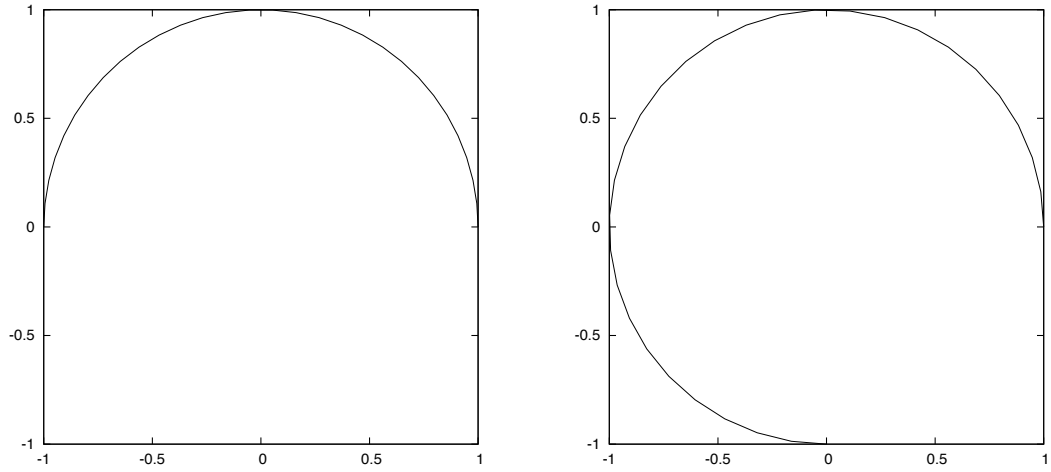


Figura 2.5 Construcción de una circunferencia en paramétricas

```
(%i55)  with_slider_draw(t,makelist(2*%pi*i/39,i,1,40),
          line_width=3, color=blue,
          parametric(cos(x),sin(x),x,0,t),
          color=light-red, key="seno",
          explicit(sin(x),x,0,t),
          color=dark-red, key="coseno",
          explicit(cos(x),x,0,t),
          line_type=dots, head_length=0.1,
          color=dark-red, key="",
          vector([0,0],[cos(t),0]),
          color=light-red, line_type=dots,
          head_length=0.1, key="",
          vector([0,0],[0,sin(t)]),
          line_type=dots, head_length=0.1, key="",
          vector([0,0],[cos(t),sin(t)]),
          xaxis=true,yaxis=true,
          title="Funciones seno y coseno",
          xrange=[-1,2*%pi],yrange=[-1,1]);
```

Para $t = 5$, el resultado lo puedes ver en la Figura 2.6

2.5 Ejercicios

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

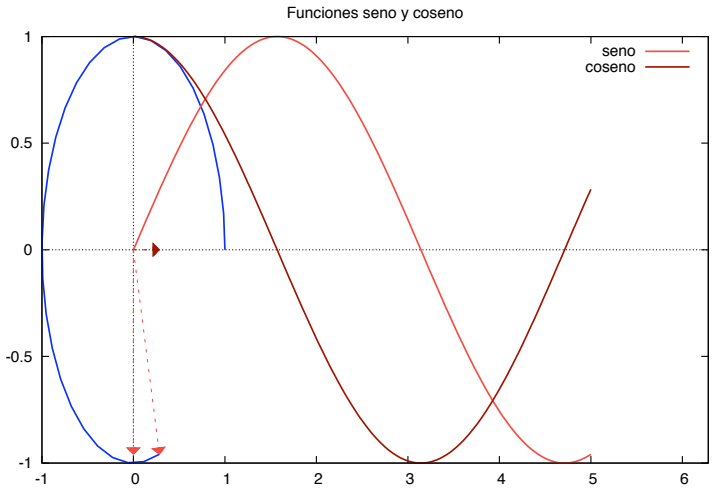


Figura 2.6 Las funciones seno y coseno

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

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

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

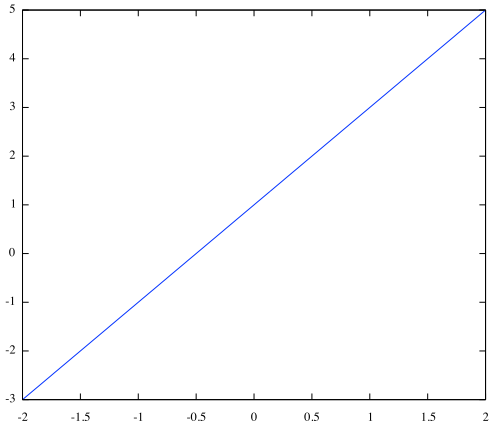
Ejercicio 2.5. Representa la curva $\cos(x)^2 - x\,\operatorname{sen}(x)^2$ en el intervalo $[-\pi, \pi]$ y sobre ella 5 puntos cuyo tamaño y color debes elegir tú. ¿Sabrías hacer lo mismo con 8 puntos elegidos aleatoriamente?[6]

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

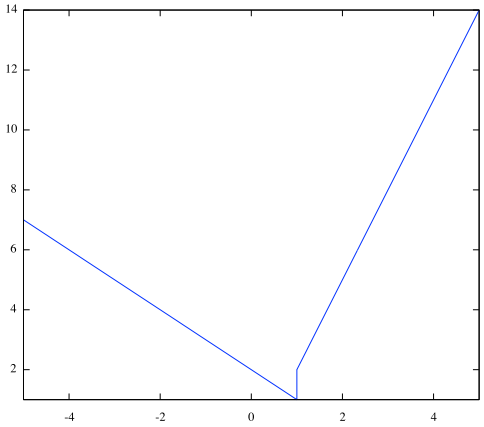
$$f(x) = \begin{cases} e^{3x+1}, & \text{si } 0 \leq x < 10, \\ \ln\left(x^2 + 1\right), & \text{si } x \geq 10. \end{cases}$$

Ejercicio 2.7. Encuentra las funciones cuyas gráficas corresponden a las siguientes curvas:

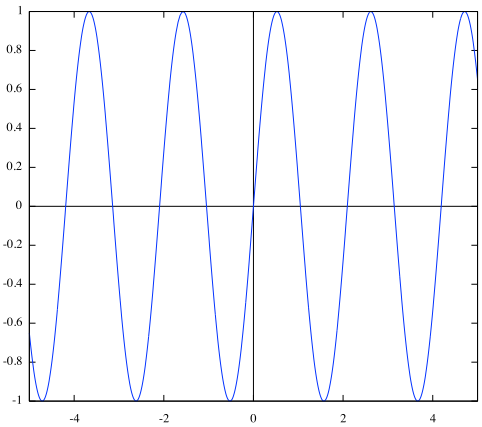
[6] En el siguiente capítulo puedes encontrar una explicación más detallada sobre como definir y operar con listas.


5
4
3
2
1
0
-1
-2
-3
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

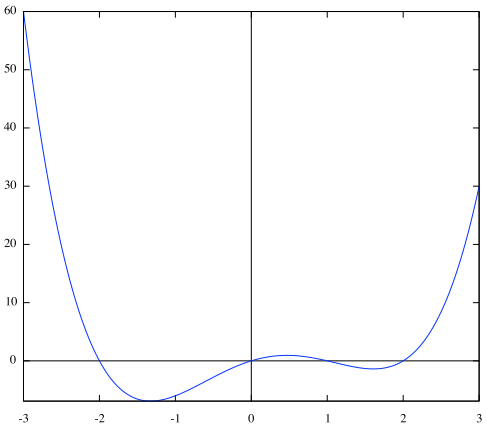
(a)


14
12
10
8
6
4
2
-4
-2
0
2
4

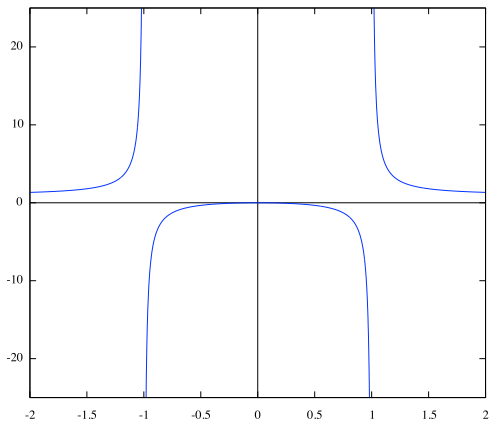
(b)


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-4
-2
0
2
4

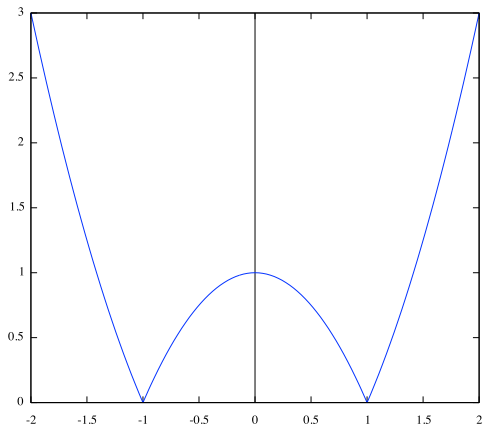
(c)


60
50
40
30
20
10
0
-3
-2
-1
0
1
2
3

(d)


20
10
0
-10
-20
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

(e)


3
2.5
2
1.5
1
0.5
0
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

(f)

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

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

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

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

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

```
(%i5)   first(lista);
(%o5)   [1,2]
(%i6)   second(lista);
(%o6)   1
```

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

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

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

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

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

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

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

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

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

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

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

```
(%i14)  unique(%);
(%o14)  [1,2,3,a]
```

Vectores

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

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

o multiplicarlos.

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

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

```
(%i19)  v1.v2;
```

```
(%o19)  -5
```

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

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

length o saber cuántos elementos tiene

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

3.1.1 Construir y operar con listas

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

makelist 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 sen(x), sen($2x$),..., sen($20x$). Seguro que no tienes ganas de escribir la lista completa. Este es el papel de la orden `makelist`. Para escribir esa lista necesitamos la regla, la fórmula que la define, un parámetro y entre qué dos valores se mueve dicho parámetro:

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

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

```
(%i23)  makelist(i,i,1,100);
```

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

o si sólo queremos los pares:

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

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

map

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

o si queremos la expresión decimal

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

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

apply

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

nos da la suma de los primeros 100 naturales.

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

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

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

```
(%i29)  apply("*",makelist(i,i,1,100))^(1/100);
(%o29)  17^(1/20) 19^(1/20) 23^(1/25) 37^(1/50) 41^(1/50) 43^(1/50) 47^(1/50) 2401^(1/25) 15625^(1/25) 531441^(1/25)
        63871474118205504453 0[30digits]99766363898994157944832^(1/100)
(%i30)  float(%);
(%o30)  37.9926893448343
```

Parece que la media geométrica es menor.

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

```
(%i31)  vector:[1,3,-7,8,1];
(%o31)  [1,3,-7,8,1]
(%i32)  sqrt(apply("+",vector^2));
(%o32)  2*sqrt(31)
```

También es posible calcular el módulo como la raíz cuadrada del producto escalar de un vector consigo mismo.

```
(%i33)  sqrt(vector.vector);
(%o33)  2*sqrt(31)
```

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

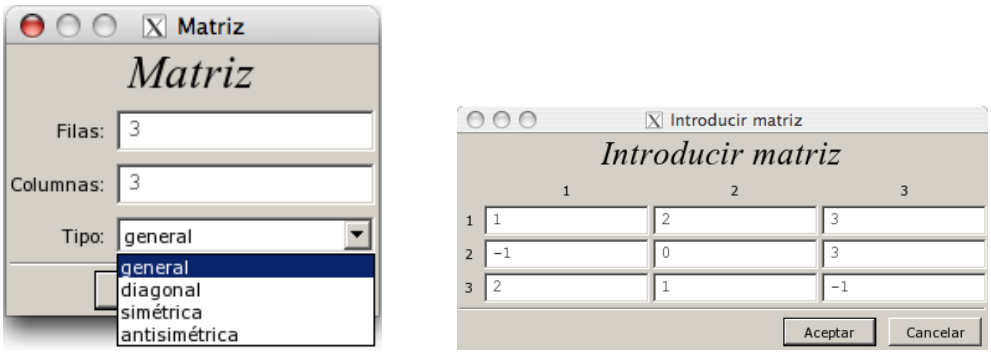
3.2 Matrices

matrix

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.

```
(%i34)  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];
(%o35)  matrix([1,2,3],[-1,0,3],[2,1,-1])
(%o36)  matrix([-1,1,1],[1,0,0],[-3,7,2])
(%o37)  [1,2,1]
(%o37)  [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.



Seleccionar tipo de matriz Introducir valores

Figura 3.1 Introducir matriz

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

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

```
(%i38)  matrix_size(A);
(%o38)  [3,3]
```

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

```
(%i39)  matrixp(a);
(%o39)  false
```

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

```
(%i40)  listp(a);
(%o40)  true
```

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

```
(%i41)  [1,2,3];
(%o41)  [1,2,3]
(%i42)  matrix([1,2,3]);
(%o42)  [1   2   3]
```

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

3.2.1 Operaciones elementales con matrices

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

```
(%i43)  A+B;
(%o43)  [ 0   3   4 ]
        [ 0   0   3 ]
        [-1   8   1 ]
(%i44)  A-B;
(%o44)  [  2   1   2 ]
        [ -2   0   3 ]
        [  5  -6  -3 ]
```

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.

```
(%i45)  A.B;
(%o45)  [ -8   22   7 ]
        [ -8   20   5 ]
        [  2   -5   0 ]
```

```
(%i46)  A*B;
(%o46)  [ -1  2  3 ]
        [ -1  0  0 ]
        [ -6  7 -2 ]
```

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

```
(%i47)  A^^2;
(%o47)  [  5  5  6 ]
        [  5  1 -6 ]
        [ -1  3 10 ]
```

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

```
(%i48)  A^2
(%o48)  [ 1  4  9 ]
        [ 1  0  9 ]
        [ 4  1  1 ]
```

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

```
(%i49)  A.a;
(%o49)  [ 8 ]
        [ 2 ]
        [ 3 ]
```

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

```
(%i50)  a.A;
(%o50)  [ 1  3  8 ]
```

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

```
(%i51)  2*A;
(%o51)  [  2  4  6 ]
        [ -2  0  6 ]
        [  4  2 -2 ]
```

3.2.2 Otras operaciones usuales

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

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

```
(%i52)  transpose(A);
(%o52)  [ 1  -1   2 ]
        [ 2   0   1 ]
        [ 3   3  -1 ]
```

determinant calcular el determinante,

```
(%i53)  determinant(A);
(%o53)  4
```

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

```
(%i54)  invert(A);
(%o54)  [ -3/4   5/4   3/2 ]
        [  5/4  -7/4  -3/2 ]
        [ -1/4   3/4   1/2 ]
```

rank 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`

```
(%i55)  m:matrix([1,3,0,-1],[3,-1,0,6],[5,-3,1,1])$
(%i56)  rank(m);
```

```
(%o56)  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: **triangularize**

```
(%i57)  triangularize(m);
                                [ 1   3    0    -1  ]
                                [                   ]
                                [ 0  -10   0     9  ]
(%o57)                          [                   ]
                                [ 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 **minor**

```
(%i58)  minor(A,2,1);
                                [ 2   3  ]
(%o58)                          [        ]
                                [ 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: **submatrix**

```
(%i59)  submatrix(2,m,1,4);
                                [ 3   0 ]
(%o59)                          [       ]
                                [ -3  1 ]
```

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

```
(%i60)  col(m,2);
                                [  3 ]
                                [    ]
                                [ -1 ]
(%o60)                          [    ]
                                [ -3 ]
```

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

```
(%i61)  row(m,1);
(%o61)  [1   3   0   -1]
(%i62)  matrixp(%);
(%o62)  true
```

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

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

nullspace 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:

```
(%i63)  nullspace(matrix([1,2,4],[-1,0,2]));
```

$$(\%o63)\quad \mathrm{span}\left(\begin{bmatrix} -4 \\ 6 \\ -2 \end{bmatrix}\right)$$

3.2.3 Más sobre escribir matrices

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

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

entermatrix 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:

```
(%i64)  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.
```

$$(\%o64)\quad \begin{bmatrix} 1 & 2 & 4 \\ -1 & 0 & 2 \end{bmatrix}$$

diagmatrix 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.

```
(%i65)  diagmatrix(4,1);
```

```
(%o65)  [1 0 0 0]
        [0 1 0 0]
        [0 0 1 0]
        [0 0 0 1]
```

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[7]

```
(%i66)  a[i,j]:=i*j;
(%o66)  a_ij:=ij
```

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

```
(%i67)  genmatrix(a,3,3);
(%o67)  [1 2 3]
        [2 4 6]
        [3 6 9]
```

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

```
(%i68)  matrixp(a);
(%o68)  false
```

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

```
(%i69)  c:genmatrix(a,4,5);
(%o69)  [1 2  3  4  5 ]
        [2 4  6  8  10]
        [3 6  9  12 15]
        [4 8  12 16 20]
```

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):

```
(%i70)  c[2,3];
(%o70)  6
```

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

3.2.4 Valores propios

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

charpoly

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.

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

$$(\%o71)\quad \begin{bmatrix} -\frac{11}{15} & -\frac{2}{15} & -\frac{4}{3} \\ -\frac{17}{15} & \frac{16}{15} & -\frac{1}{3} \\ -\frac{8}{15} & \frac{4}{15} & \frac{5}{3} \end{bmatrix}$$

```
(%i72)  charpoly(S,x);
```

$$(\%o72)\quad \left(\left(\frac{16}{15}-x\right)\left(\frac{5}{3}-x\right)+\frac{4}{45}\right)\left(-x-\frac{11}{15}\right)+\frac{2\left(-\frac{17\left(\frac{5}{3}-x\right)}{15}-\frac{8}{45}\right)}{15}-\frac{4\left(\frac{8\left(\frac{16}{15}-x\right)}{15}-\frac{68}{225}\right)}{3}$$

```
(%i73)  expand(%);
(%o73)  -x^3+2x^2+x-2
```

Por tanto, sus valores propios son

```
(%i74)  solve(%,x);
(%o74)  [x=2,x=-1,x=1]
```

eigenvalues

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

```
(%i75)  eigenvalues(S);
(%o75)  [[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

```
(%i76)  eigenvectors(S);
(%o76)  [[[2,-1,1],[1,1,1]],[1,-1/2,-2],[1,4/7,1/7],[1,7,-2]]
```

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

3.3 Ejercicios

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

a) $a + b$,

b) $3c + 2b$,

c) $c.d$, y

d) $b.d + 3a.c$.

Ejercicio 3.2. Consideremos las matrices

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

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

a) Calcular $A.B$, $A + B$, $D.C$.

b) Extraer la segunda fila de A, la tercera columna de C y el elemento $(3, 3)$ de D.

c) Calcular $\det(A)$, $\det(B)$ y $\det(C)$. Para las matrices cuyo determinante sea no nulo, calcular su inversa. Calcular sus valores propios.

d) Calcular el rango de las matrices A, B, C, D, $D.C$ y $A + B$.

e) Construye una matriz del orden 3×3, de forma que el elemento (i, j) sea $i * j + j - i$. Calcula el determinante, su inversa si la tiene, y su rango. ¿Cuáles son sus valores propios?

Ejercicio 3.3. Calcula el rango de la matriz

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

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

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

Ejercicio 3.5.
a) Genera una lista de 10 números aleatorios entre 5 y 25 y reordénala en orden decreciente.

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

a) Multiplica cada elemento de "listauno" por todos los elementos de "listados". El resultado será una lista con 20 elementos (que a su vez serán listas de 10 elementos), a la que llamarás "productos".

b) Calcula la suma de cada una de las listas que forman la lista "productos" (no te equivoques, comprueba el resultado). Obtendrás una lista con 20 números.

c) Calcula el producto de los elementos de la lista obtenida en el apartado anterior.

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

Ejercicio 3.8.
a) Calcula la suma de los números de la forma $\frac{(-1)^{k+1}}{\sqrt{k}}$ desde $k = 1$ hasta $k = 1000$.
b) Calcula el producto de los números de la forma $\left(1 + \frac{1}{k^2}\right)$ desde $k = 1$ hasta $k = 1000$.

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. Si tienes alguna duda de cómo operar con números complejos, en el Apéndice A tienes una breve introducción sobre su uso.

4.1 Ecuaciones y operaciones con ecuaciones

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

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

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

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

además podemos asignarle un nombre para poder referirnos a ella

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

y operar como con cualquier otra expresión

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

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

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

4.2 Resolución de ecuaciones

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

4.2.1 La orden `solve`

solve

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

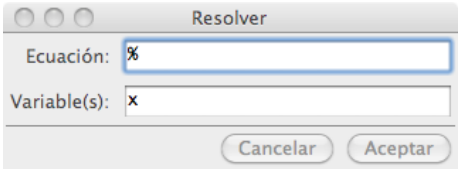


Figura 4.1 Resolver una ecuación

```
(%i5)  solve(x^2-3*x+1=0,x);
(%o5)  [x=-(sqrt(5)-3)/2,x=(sqrt(5)+3)/2]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(%i14)   solve(eq1,a);
```

```
(%o14)   [a=(x^2-x+2)/2]
```

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

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

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

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

```
(%i16)   eq:x+3=sqrt(x+1);
(%o16)   x+3=sqrt(x+1)
(%i17)   solve(eq,x);
(%o17)   [x=sqrt(x+1)-3]
(%i18)   solve(eq^2);
(%o18)   [x=-(sqrt(7)%i+5)/2,x=(sqrt(7)%i-5)/2]
```

Cómo hacer referencia a las soluciones

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

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

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

La primera solución es

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

```
(%o20)  3
```

y la segunda

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

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

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

Sistemas de ecuaciones

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

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

escribimos

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2.2 to_poly_solve

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

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

En cambio, tiene un poco más de éxito

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

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

`to_poly_solve(`*ecuación*`, `*variable*`)`	resuelve la ecuación
`to_poly_solve(`*expr*`, `*variable*`)`	resuelve la expresión igualada a cero

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

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

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

```
(%i32)  to_poly_solve(x*cos(x),x);
(%o32)  %union([x=0],[x=2π%z101−π/2],[x=2π%z103+π/2])
```

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

4.2.3 Sistemas de ecuaciones lineales

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

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

linsolve

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

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

4.2.4 Algsys

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

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

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

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

```
(%i37)  solve(eq:x^6+x+1);
(%o37)  [0=x^6+x+1]
(%i38)  algsys([eq],[x]);
(%o38)  [[x=-1.038380754458461%i-0.15473514449684],
        [x=1.038380754458461%i-0.15473514449684],
        [x=-0.30050692030955%i-0.79066718881442],
        [x=0.30050692030955%i-0.79066718881442],
        [x=0.94540233331126-0.61183669378101%i],
        [x=0.61183669378101%i+0.94540233331126]]
```

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

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

4.3 Ejercicios

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

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

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

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

a) la recta $x - y = 5$ y la parábola $(x - 1)^2 + y = 4$;

b) la hipérbola equilátera y la circunferencia de centro $(-1, 1)$ y radio 1;

c) las circunferencias de centro $(0, 0)$ y radio 2 y la de centro $(-1, 3)$ y radio 3.

Ejercicio 4.5. Resolver la ecuación logarítmica:

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

Métodos numéricos de resolución de ecuaciones

5

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

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

5.1 Introducción al análisis numérico

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

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

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

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

5.1.1 Números y precisión

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

```
(%i1)   sqrt(2);
(%o1)   √2
```

o podemos con la precisión por defecto

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

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

```
(%i3)   fpprec:20;
(%o3)   20
(%i4)   bfloat(sqrt(2));
(%o4)   1.4142135623730950488b0
```

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

Errores de redondeo

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

```
(%i5)   fpprec:5;
(%o5)   5
(%i6)   bfloat(7.12345);
(%o6)   7.1234b0
```

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

```
(%i7)   fpprec:20;
(%o7)   20
(%i8)   bfloat(0.1);
(%o8)   1.0000000000000000555b-1
```

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

```
(%i9)   bfloat(1/10);
```

```
(%o9)   1.0b-1
```

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

5.1.2 Aritmética de ordenador

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

Elemento neutro

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

```
(%i10)  h:2.22045*10^(-17);
(%o10)  2.22045 10^-17
```

Y si nos cuestionamos si h funciona como elemento neutro:

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

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

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

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

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

Propiedad asociativa de la suma

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

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

```
(%o15)   false
```

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

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

5.1.3 Cancelación de cifras significativas

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

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

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

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

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

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

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

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

en cambio,

```
(%i24)  b:1+a$
(%i25)  c:(a^2-1)/(a-1)$
(%i26)  is(b=c);
(%o26)  false
```

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

5.2 Resolución numérica de ecuaciones con *Maxima*

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

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

Estás órdenes nos dan todas las soluciones reales y complejas de un polinomio en una variable y son útiles en polinomios de grado alto cuando falla la orden `solve`. La primera de ellas, `allroots`, nos da las soluciones con la precisión por defecto

allroots

```
(%i27)  eq:x^9+x^7-x^4+x$
(%i28)  allroots(eq);
(%o28)  [x=0.0,x=0.30190507748312%i+0.8440677798278,
        x=0.8440677798278-0.30190507748312%i,
        x=0.8923132916888%i-0.32846441923834,
        x=-0.8923132916888%i-0.32846441923834,
        x=0.51104079208431%i-0.80986929589487,
        x=-0.51104079208431%i-0.80986929589487,
        x=1.189238256723466%i+0.29426593530541,
        x=0.29426593530541-1.189238256723466%i]
```

bfallroots Si queremos una precisión determinada, usamos la orden `bfallroots`.

```
(%i29)  fpprec:6$
        bfallroots(eq);
        [x=0.0b0,x=3.0191b-1%i+8.44063b-1,
        x=8.44063b-1-3.0191b-1%i,x=8.92279b-1%i-3.28481b-1,
(%o29)  x=-8.92279b-1%i-3.28481b-1,x=5.11037b-1%i-8.09838b-1,
        x=-5.11037b-1%i-8.09838b-1,x=1.18924b0%i+2.94256b-1,
        x=2.94256b-1-1.18924b0%i]
```

realroots Si sólo nos interesan las soluciones reales, la orden `realroots` calcula soluciones racionales aproximadas del polinomio.

```
(%i30)  eq1:x^4-3*x^3+x^2-4*x+12$
(%i31)  realroots(eq1);
(%o31)  [x=2,x=81497599/33554432]
```

Si comparas con la salida de `allroots`, comprobarás que 2 es solución, pero que $\frac{81497599}{33554432}$ sólo es una solución aproximada. La precisión con la que se realiza la aproximación se puede controlar con un segundo parámetro. Por ejemplo, si queremos que el error sea menor que 10^{-5}, escribimos lo siguiente.

```
(%i32)  realroots(eq1,10^(-5));
(%o32)  [x=2,x=636699/262144]
```

Recuerda que la variable `multiplicities` guarda la multiplicidad de cada una de las raíces de la última ecuación que has resuelto.

```
(%i33)  realroots((x-2)^2*eq1,10^(-5));
(%o33)  [x=2,x=636699/262144]
(%i34)  multiplicities;
(%o34)  [3,1]
```

Por último, comentar que es posible saber el número de raíces de un polinomio en una variable en un intervalo concreto. Incluso se admiten $\pm\infty$ como posibles extremos del intervalo.

```
(%i35)  nroots(eq1,0,2);
(%o35)  1
```

eso sí, ten cuidado porque se cuentan raíces con su multiplicidad

```
(%i36)  nroots((x-2)^2*eq1,0,2);
(%o36)  3
```

El teorema de los ceros de Bolzano

Uno de los primeros resultados que aprendemos sobre funciones continuas es que si cambian de signo tienen que valer cero en algún momento. Para que esto sea cierto nos falta añadir un ingrediente: la funciones tienen que estar definidas en intervalos. Este resultado se conoce como teorema de los ceros de Bolzano y es una variante del teorema del valor intermedio.

Teorema de los ceros de Bolzano

Teorema 5.1. *Sea $f:[a,b]\to\mathbb{R}$ una función continua verificando que $f(a)f(b)<0$, entonces existe $c\in]a,b[$ tal que $f(c)=0$.*

Ejemplo 5.2. Una de las utilidades más importantes del Teorema de los ceros de Bolzano es garantizar que una ecuación tiene solución. Por ejemplo, para comprobar que la ecuación $e^x+\log(x)=0$ tiene solución, estudiamos la función $f(x)=e^x+\log(x)$: es continua en $\mathbb{R}^+$ y se puede comprobar que $f(e^{-10})<0$ y $0<f(e^{10})$. Por tanto, la ecuación $e^x+\log(x)=0$ tiene al menos una solución entre e^{-10} y e^{10}. En particular, tiene solución en $\mathbb{R}^+$.

`find_root(f(x),x,a,b)`	solución de f en $[a,b]$

find_root

El comando `find_root` encuentra una solución de una función (ecuación) continua que cambia de signo por el método de bisección, esto es, dividiendo el intervalo por la mitad y quedándose con aquella mitad en la que la función sigue cambiando de signo. En realidad el método que utiliza *Maxima* es algo más elaborado pero no vamos a entrar en más detalles.

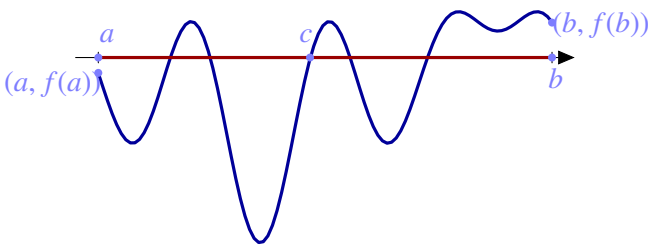


Figura 5.1 Teorema de los ceros de Bolzano

```
(%i37)  f(x):=exp(x)+log(x);
(%o37)  f(x):=exp(x)+log(x)
```

Buscamos un par de puntos donde cambie de signo

```
(%i38)  f(1);
(%o38)  %e
(%i39)  f(exp(-3));
(%o39)  %e^(%e^-3)+3
```

¿Ese número es negativo?

```
(%i40)  is(f(exp(-3))<0);
(%o40)  true
```

o bien,

```
(%i41)  f(exp(-3)),numer;
(%o41)  -1.948952728663784
```

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

```
(%i42)  find_root(f(x),x,exp(-3),1);
(%o42)  0.26987413757345
```

⚠ **Observación 5.3.** Este método encuentra *una* solución pero no nos dice cuántas soluciones hay. Para eso tendremos que echar mano de otras herramientas adicionales como, por ejemplo, el estudio de la monotonía de la función.

5.2.1 Ejercicios

Ejercicio 5.1. Calcula las soluciones de $8\,\mathrm{sen}(x) + 1 - \frac{x^2}{3} = 0$.

Ejercicio 5.2. Encuentra una solución de la ecuación $\tan(x) = \frac{1}{x}$ en el intervalo $\left]0, \frac{\pi}{2}\right[$.

Ejercicio 5.3. ¿Cuantas soluciones tiene la ecuación $\frac{e^x}{2} - 2\,\mathrm{sen}(x) = 1$ en el intervalo $[-3, 3]$?

5.3 Breves conceptos de programación

Hemos visto cómo resolver ecuaciones y sistemas de ecuaciones con *Maxima* mediante la orden `solve` o `algsys`. La resolución de ecuaciones y sistemas de ecuaciones de manera exacta está limitada a aquellas para los que es posible aplicar un método algebraico sencillo. En estas condiciones, nos damos cuenta de la necesidad de encontrar o aproximar soluciones para ecuaciones del tipo $f(x) = 0$, donde, en principio, podemos considerar como f cualquier función real de una variable. Nuestro siguiente objetivo es aprender a "programar" algoritmos con *Maxima* para aproximar la solución de estas ecuaciones.

Lo primero que tenemos que tener en cuenta es que no existe ningún método general para resolver todo este tipo de ecuaciones en un número finito de pasos. Lo que sí tendremos es condiciones para poder asegurar, bajo ciertas hipótesis sobre la función f, que un determinado valor es una aproximación de la solución de la ecuación con un error prefijado.

El principal resultado para asegurar la existencia de solución para la ecuación $f(x) = 0$ en un intervalo $[a, b]$, es el Teorema de Bolzano que hemos recordado más arriba. Dicho teorema asegura que si f es continua en $[a, b]$ y cambia de signo en el intervalo, entonces existe al menos una solución de la ecuación en el intervalo $[a, b]$.

Vamos a ver dos métodos que se basan en este resultado. Ambos métodos nos proporcionan un algoritmo para calcular una sucesión de aproximaciones, y condiciones sobre la función f para poder asegurar que la sucesión que obtenemos converge a la solución del problema. Una vez asegurada esta convergencia, bastará tomar alguno de los términos de la sucesión que se aproxime a la sucesión con la exactitud que deseemos.

5.3.1 Bucles

Antes de introducirnos en el método teórico de resolución, vamos a presentar algunas estructuras sencillas de programación que necesitaremos más adelante.

La primera de las órdenes que vamos a ver es el comando `for`, usada para realizar bucles. Un bucle es un proceso repetitivo que se realiza un cierto número de veces. Un ejemplo de bucle puede ser el siguiente: supongamos que queremos obtener los múltiplos de siete comprendidos entre 7 y 70; para ello, multiplicamos 7 por cada uno de los números naturales comprendidos entre 1 y 10, es decir, repetimos 10 veces la misma operación: multiplicar por 7.

`for` *var:valor1* `step` *valor2* `thru` *valor3* `do` *expr*	bucle `for`
`for` *var:valor1* `step` *valor2* `while` *cond* `do` *expr*	bucle `for`
`for` *var:valor1* `step` *valor2* `unless` *cond* `do` *expr*	bucle `for`

En un bucle `for` nos pueden aparecer los siguientes elementos (no necesariamente todos)

a) *var:valor1* nos sitúa en las condiciones de comienzo del bucle.
b) *cond* dirá a *Maxima* el momento de detener el proceso.
c) *step valor2* expresará la forma de aumentar la condición inicial.
d) *expr* dirá a *Maxima* lo que tiene que realizar en cada paso; *expr* puede estar compuesta de varias sentencias separadas mediante punto y coma.

En los casos en que el paso es 1, no es necesario indicarlo.

`for` *var:valor1* `thru` *valor3* `do` *expr*	bucle `for` con paso 1
`for` *var:valor1* `while` *cond* `do` *expr*	bucle `for` con paso 1
`for` *var:valor1* `unless` *cond* `do` *expr*	bucle `for` con paso 1

Para comprender mejor el funcionamiento de esta orden vamos a ver algunos ejemplos sencillos.

En primer lugar, generemos los múltiplos de 7 hasta 70:

```
(%i43)  for i:1 step 1 thru 10 do print(7*i);
        7
        14
        21
        28
        35
        42
        49
        56
        63
        70
```

```
(%o43)  done
```

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

```
(%i44)  for i:5 step 5 thru 25 do print(i);
        5
        10
        15
        20
        25
(%o44)  done
```

Ejemplo 5.4. Podemos utilizar un bucle para sumar una lista de números pero nos hace falta una variable adicional en la que ir guardando las sumas parciales que vamos obteniendo. Por ejemplo, el siguiente código suma los cuadrados de los 100 primeros naturales.

```
(%i45)  suma:0$
        for i:1 thru 100 do suma:suma+i^2$
        print("la suma de los cuadrados de los 100 primeros
          naturales vale ",suma);
(%o45)  la suma de los cuadrados de los 100 primeros
        naturales vale 338350
```

`print(`*expr1*`,`*expr2*`,...)`	escribe las expresiones en pantalla

print 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.

```
(%i46)  i:0;
(%o46)  0
(%i47)  while cos(i)>0 do (print([i,cos(i)]),i:i+0.3);
        [0, 1]
        [0.3, 0.95533648560273]
        [0.6, 0.82533560144755]
        [0.9, 0.62160994025671]
        [1.2, 0.36235771003359]
        [1.5, 0.070737142212368]
(%o47)  done
```

5.3.2 Condicionales

La segunda sentencia es la orden condicional `if`. Esta sentencia comprueba si se verifica una condición, después, si la condición es verdadera *Maxima* ejecutará una *expresión1*, y si es falsa ejecutará otra *expresión2*.

`if` *condición* `then` *expr1* `else` *expr2*	condicional if-then-else
`if` *condición* `then` *expr*	condicional if-then

La expresiones 1 y 2 pueden estar formadas por varias órdenes separadas por comas. Como siempre en estos casos, quizá un ejemplo es la mejor explicación:

```
(%i48)  if log(2)<0 then x:5 else 3;
(%o48)  3
```

Observa que la estructura if-then-else devuelve la expresión correspondiente y que esta expresión puede ser una asignación, algo más complicado o algo tan simple como "3".

La última sentencia de programación que vamos a ver es la orden `return(var)` cuya única finalidad es la de interrumpir un bucle en el momento que se ejecuta y devolver un valor. En el siguiente ejemplo se puede comprender rápidamente el uso de esta orden.

```
(%i49)  for i:1 thru 10 do
          (
          if log(i)<2 then print("el logaritmo de",i,"es menor
            que 2")   else return(x:i)
          )$
        print("el logaritmo de ",x," es mayor que 2")$
        el logaritmo de 1 es menor que 2
        el logaritmo de 2 es menor que 2
        el logaritmo de 3 es menor que 2
        el logaritmo de 4 es menor que 2
        el logaritmo de 5 es menor que 2
        el logaritmo de 6 es menor que 2
        el logaritmo de 7 es menor que 2
        el logaritmo de 8 es mayor que 2
```

Observación 5.5. La variable que se utiliza como contador, `i` en el caso anterior, es siempre local al bucle. No tiene ningún valor asignado fuera de él. Es por esto que hemos guardado su valor en una variable auxiliar, `x`, para poder usarla fuera del bucle.

5.3.3 Ejercicios

Ejercicio 5.4. Usa el comando `for` en los siguientes ejemplos:
a) Sumar los números naturales entre 400 y 450.
b) Calcula la media de los cuadrados de los primeros 1000 naturales.

Ejercicio 5.5. Dado un número positivo x, se puede conseguir que la suma

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

sea mayor que x tomando un número n suficientemente grande. Encuentra la forma de calcular dicho número de manera general. ¿Cuál es el valor para $x = 10$, 11 y 13?

Ejercicio 5.6. Calcula las medias geométricas de los n primeros naturales y averigua cuál es el primer natural para el que dicha media es mayor que 20.

Ejercicio 5.7. Calcula la lista de los divisores de un número natural n.

5.4 El método de bisección

El método de bisección es una de las formas más elementales de buscar una solución de una ecuación. Ya sabemos que si una función continua cambia de signo en un intervalo, entonces se anula en algún punto. ¿Cómo buscamos dicho punto?

Comencemos con una función $f\colon [a,b] \to \mathbb{R}$ continua y verificando que tiene signos distintos en los extremos del intervalo. La idea básica es ir dividiendo el intervalo por la mitad en dos subintervalos, $[a, \frac{1}{2}(a+b)]$ y $[\frac{1}{2}(a+b), b]$, y elegir aquel en el que la función siga cambiando de signo. Si repetimos este proceso, obtenemos un intervalo cada vez más pequeño donde se encuentra la raíz.

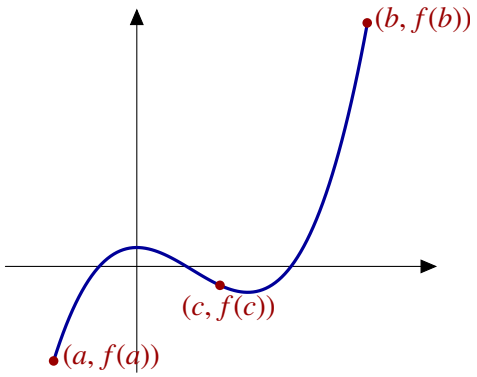


Figura 5.2 Método de bisección

Más concretamente el proceso sería,

Datos iniciales: función f, números a, b
Se verifica que: $f(a)f(b) < 0$
bucle
- ▷ Calculamos el punto medio $c = (a + b)/2$,
- ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
- ▷ y elegimos aquél donde la función cambie de signo

final bucle

5.4.1 Un ejemplo concreto

Vamos a aplicar este método a la función $f(x) = x^6 + x - 5$ en el intervalo $[0, 2]$.

Definiremos en primer lugar la función y el intervalo y luego un bucle que nos calcula el intervalo donde, después de los pasos que digamos, se encuentra localizado el cero de la función.

```
(%i50)  f(x):=x^6+x-5;
        a:0.0;
        b:2.0;
(%o51)  x^6+x-5
(%o52)  0.0
(%o53)  2.0
```

Observa que hemos declarado `a` y `b` como valores numéricos. Comprobamos que la función cambia de signo.

```
(%i54)  f(a)*f(b);
(%o54)  -305.0
```

Ahora el bucle,[8]

[8] Los símbolos /* y */ indican el principio y el fin de un comentario. No se ejecutan.

```
(%i55)  for i:1 thru 10 do
          (
          c:(a+b)/2, /* calculamos el punto medio */
          if f(a)*f(c)<0 /* ¿cambia de signo en [a,c]? */
            then b:c /* elegimos [a,c] */
            else a:c, /* elegimos [c,b] */
          print(a,b) /* escribimos los resultados por pantalla */
          )$
          1.0 2.0
          1.0 1.5
          1.0 1.25
          1.125 1.25
          1.1875 1.25
(%o55)    1.21875 1.25
          1.234375 1.25
          1.2421875 1.25
          1.24609375 1.25
          1.24609375 1.248046875
```

Fíjate que ya sabemos que la solución es aproximadamente 1.24. No podemos estar seguros todavía del tercer decimal. Quizá sería mejor repetir el bucle más de diez veces, pero, ¿cuantas? Podemos establecer como control que la distancia entre los extremos sea pequeña. Eso sí, habría que añadir un tope al número de pasos para asegurarnos de que el bucle termina.

⚠ **Observación 5.6.** Si no se quiere ralentizar mucho la ejecución de este bucle y del resto de programas en el resto del tema, es conveniente trabajar en modo númerico. Recuerda que este comportamiento se controla con la variable `numer`. Puedes cambiarlo en el menú **Numérico→Conmutar salida numérica** o directamente estableciendo el valor de `numer` en verdadero.

```
(%i56)  numer:true;
(%o56)  true
```

Control del error

En este método es fácil acotar el error que estamos cometiendo. Sabemos que la función f se anula entre a y b. ¿Cuál es la mejor elección sin tener más datos? Si elegimos a la solución, teóricamente, podría ser b. El error sería en este caso $b-a$. ¿Hay alguna elección mejor? Sí, el punto medio $c=(a+b)/2$. ¿Cuál es el error ahora? Lo peor que podría pasar sería que la solución fuera alguno de los extremos del intervalo. Por tanto, el error sería como mucho $\frac{b-a}{2}$. En cada paso que damos dividimos el intervalo por la mitad y al mismo tiempo también el error cometido que en el paso n-ésimo es menor o igual que $\frac{b-a}{2^n}$.

A partir de aquí, podemos deducir el número de iteraciones necesarias para obtener una aproximación con un error o exactitud prefijados. Si notamos por "`err`" a la exactitud prefijada, entonces para conseguir dicha precisión, el número "n" de iteraciones necesarias deberá satisfacer

$$\frac{b-a}{2^n} \leq err$$

así,

$$n \geq \log_2\left(\frac{b-a}{err}\right) = \frac{\log\left(\frac{b-a}{err}\right)}{\log(2)}.$$

`ceiling(a)`	menor entero mayor o igual que `a`

La orden `ceiling(x)` nos da el menor entero mayor o igual que x. Bueno, ya sabemos cuántos pasos tenemos que dar. Reescribimos nuestro algoritmo con esta nueva información: **ceiling**

Datos iniciales: función f, números a, b, error err, contador i
Se verifica que: $f(a)f(b) < 0$
▷ Calculamos el número de pasos
para i:1 hasta número de pasos **hacer**
▷ Calculamos el punto medio $c = (a+b)/2$,
▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
▷ y elegimos aquél donde la función cambie de signo
final bucle

Volviendo a nuestro ejemplo, nos quedaría algo así.

```
(%i57)  f(x):=x^6+x-5;
        a:0$
        b:2$
        err:10^(-6)$
        log2(x):=log(x)/log(2)$ /* hay que definir el logaritmo en
        base 2 */
        pasos:ceiling(log2((b-a)/err))$
        for i:1 thru pasos do
          (
          c:(a+b)/2,
          if f(a)*f(c)<0
            then b:c
            else a:c,
          print(a,b) /* escribimos los resultados por pantalla */
          )$
```

¿Y si hay suerte?

Si encontramos la solución en un paso intermedio no habría que hacer más iteraciones. Deberíamos parar y presentar la solución encontrada. En cada paso, tenemos que ir comprobando que $f(c)$ vale o no vale cero. Podríamos comprobarlo con una orden del tipo `is(f(c)=0)`, pero recuerda que con valores numéricos esto puede dar problemas. Mejor comprobemos que es "suficientemente" pequeño.

Datos iniciales: función f, números a, b, error err, contador i, precisión pr
Se verifica que: $f(a)f(b) < 0$
▷ Calculamos el número de pasos
para i:1 hasta número de pasos **hacer**
  ▷ Calculamos el punto medio $c = (a + b)/2$,
  si f(c)<pr **entonces**
    ▷ La solución es c
  en otro caso
    ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
    ▷ y elegimos aquél donde la función cambie de signo
  final si
  ▷ La solución aproximada es c
final del bucle

En nuestro ejemplo, tendríamos lo siguiente

```
(%i58)  f(x):=x^6+x-5;
        a:0$
        b:2$
        err:10^(-6)$
        pr:10^(-5)$
        log2(x):=log(x)/log(2)$
        pasos:ceiling(log2((b-a)/err))$
        for i:1 thru pasos do
          (
          c:(a+b)/2,
          if abs(f(c))<pr
            then (print("La solucion es exacta"), return(c))
            else if f(a)*f(c)<0
              then b:c
              else a:c
          )$
        print("la solucion es ",c)$ /* aproximada o exacta, es la
        solución */
```

¿Se te ocurren algunas mejoras del algoritmo? Algunas ideas más:

a) el cálculo de $f(a)f(c)$ en cada paso no es necesario: si sabemos el signo de $f(a)$, sólo necesitamos saber el signo de $f(c)$ y no el signo del producto,

b) habría que comprobar que $f(a)$ y $f(b)$ no son cero (eso ya lo hemos hecho) ni están cerca de cero como hemos hecho con c.

c) Si queremos trabajar con una precisión mayor de 16 dígitos, sería conveniente utilizar numéros en coma flotante grandes.

5.4.2 Funciones y bloques

Una vez que tenemos más o menos completo el método de bisección, sería interesante tener una forma cómoda de cambiar los parámetros iniciales: la función, la precisión, los extremos, etc. Un bloque es la estructura diseñada para esto: permite evaluar varias expresiones y devuelve el último resultado salvo petición expresa.

La forma más elemental de "programa" en *Maxima* es lo que hemos hecho dentro del cuerpo del bucle anterior: entre paréntesis y separados por comas se incluyen comandos que se ejecutan sucesivamente y devuelve como salida la respuesta de la última sentencia.

```
(%i59)   (a:3,b:2,a+b);
(%o59)   5
```

Variables y funciones locales

Es conveniente tener la precaución de que las variables que se utilicen sean locales a dicho programa y que no afecten al resto de la sesión. Esto se consigue agrupando estas órdenes en un bloque

```
(%i60)   a:1;
(%o60)   1
(%i61)   block([a,b],a:2,b:3,a+b);
(%o61)   5
(%i62)   a;
(%o62)   1
```

Como puedes ver, la variable `a` global sigue valiendo uno y no cambia su valor a pesar de las asignaciones dentro del bloque. Esto no ocurre con las funciones que definamos dentro de un bloque. Su valor es global a menos que lo declaremos local explícitamente con la sentencia `local`. **local**
Observa la diferencia entre las funcione f y g.

```
(%i63)   block([a,b],
           local(g),g(x):=x^3,
           f(x):=x^2,
           a:2,b:3,g(a+b));
(%o63)   125
```

Si preguntamos por el valor de f o de g fuera del bloque, f tiene un valor concreto y g no:

```
(%i64)   f(x);
(%o64)   x^2
```

```
(%i65)  g(x);
(%o65)  g(x)
```

`local(`*`funciones`*`)`	declara funciones locales a un bloque
`return(`*`expr`*`)`	detiene la ejecución de un bloque y devuelve *expr*
`block([`*`var1,var2,..`*`],`*`expr1,expr2,..`*`)`	evalúa *expr1, expr2,...* y devuelve la última expresión evaluada

El último paso suele ser definir una función que permite reutilizar el bloque. Por ejemplo, el factorial de un número natural n se define recursivamente como

$$1! = 1, \quad (n+1)! = (n+1)\cdot n!.$$

Podemos calcular el factorial de un natural usando un bucle: usaremos la variable f para ir acumulando los productos sucesivos y multiplicamos todos los naturales hasta llegar al pedido.

```
(%i66)  fact(n):=block([f:1], for k:1 thru n do f:f*k,f);
(%o66)  fact(n):=block([f:1],for k thru n do f:f*k,f)
(%i67)  fact(5);
(%o67)  120
```

¿Y si quiero acabar antes?

Si queremos salir de un bloque y devolver un resultado antes de llegar a la última expresión, podemos usar la orden `return`. Por ejemplo, recuerda la definición que hicimos en el primer tema de la función logaritmo con base arbitraria.

```
(%i68)  loga(x):=log(x)/log(a)$
```

Esto podemos mejorarlo algo utilizando dos variables:

```
(%i69)  loga(x,a):=log(x)/log(a)$
```

pero deberíamos tener en cuenta si a es un número que se puede tomar como base para los logaritmos. Sólo nos valen los números positivos distintos de 1. Vamos a utilizar un bloque y un condicional.

```
(%i70)  loga(x,a):=block(
          if a<0 then print("La base es negativa"),
          if a=1 then print("La base es 1"),
          log(x)/log(a)
        )$
```

Si probamos con números positivos

```
(%i71)  loga(3,4);
                          log(3)
(%o71)                    ------
                          log(4)
```

Funciona. ¿Y si la base no es válida?

```
(%i72)  log(3,-1);
La base es negativa
                          log(3)
(%o72)                    ------
                          log(-1)
```

Fíjate que no hemos puesto ninguna condición de salida en el caso de que la base no sea válida. Por tanto, *Maxima* evalúa una tras otra cada una de las sentencias y devuelve la última. Vamos a arreglarlo.

```
(%i73)  loga(x,a):=block(
          if a<0 then
            (print("La base es negativa"),return()),
          if a=1 then
            (print("La base es 1"),return()),
          log(x)/log(a)
        )$
```

Parámetros opcionales

Para redondear la definición de la función logaritmo con base cualquiera, podría ser interesante que la función "loga" calcule el logaritmo neperiano si sólo ponemos una variable y el logaritmo en base a si tenemos dos variables.

Las entradas opcionales se pasan a la definición de una función entre corchetes. Por ejemplo, la función

```
(%i74)  f(a,[b]):=block(print(a),print(b))$
```

da por pantalla la variable `a` y el parámetro o parámetros adicionales que sean. Si solo escribimos una coordenada

```
(%i75)  f(2);
2
[]
(%o75)  []
```

nos devuelve la primera entrada y la segunda obviamente vacía en este caso. Pero si añadimos una entrada más

```
(%i76)  f(2,3);
        2
        [3]
(%o76)  [3]
```

o varias

```
(%i77)  f(2,3,4,5);
        2
        [3,4,5]
(%o77)  [3,4,5]
```

Como puedes ver, "`[b]`" en este caso representa una lista en la que incluimos todos los parámetros opcionales que necesitemos. Ahora sólo es cuestión de utilizar las sentencias que nos permiten manejar los elementos de una lista para definir la función logaritmo tal y como queríamos.

```
(%i78)  loga(x,[a]):=block(   if length(a)=0
            then return(log(x))
            else (
              if a[1]<0 then (print("La base es negativa"),return()),
              if a[1]=1 then (print("La base es 1"),return()),
              log(x)/log(a[1])
            )
        )$
```

Observación 5.7. Se puede salir del bloque de definición de la función usando la sentencia `error(`*`mensaje`*`)` en los casos en que la base no sea la adecuada.

```
(%i79)  loga(x,[a]):=block(   if length(a)=0
            then return(log(x))
            else (
              if a[1]<0 then error("Cambia la base"),
              if a[1]=1 then error("La base es 1"),
              log(x)/log(a[1])
            )
        )$
```

5.4.3 De nuevo bisección

Si unimos todo lo que hemos aprendido, podemos definir una función que utilice el método de bisección. Hemos usado la sentencia `subst` para definir la función a la que aplicamos bisección dentro del bloque. La orden `subst(a,b,c)` sustituye `a` por `b` en `c`.

```
biseccion(expr,var,ext_inf,ext_sup):=
    block(
        [a,b,c,k,err:10^(-8),prec:10^(-9)],
        /* extremos del intervalo */
        a:ext_inf,
        b:ext_sup,

        /* número de pasos */
        local(log2,f),
        define(log2(x),log(x)/log(2)),
        define(f(x),subst(x,var,expr)),
        pasos:ceiling(log2((b-a)/err)),

       /* comprobamos las condiciones iniciales */
        if f(a)*f(b)>0 then error("Error: no hay cambio de signo"),

        /* ¿se alcanza la solución en los extremos? */
        if abs(f(a)) < prec then return(a),
        if abs(f(b)) < prec then return(b),

        for k:1 thru pasos do
           (
           c:(a+b)/2,
           if abs(f(c))< prec then return (c),
           if f(a)*f(c)< 0 then b:c else a:c
           ),
        c
       );
```

A partir de este momento, podemos utilizarlo usando

```
(%i80)   biseccion(x^2-2,x,0.0,3.0);
(%o80)   1.414213562384248
```

Observa que las cotas del error y la precisión la hemos fijado dentro del bloque. Prueba a añadirlo como valores opcionales.

5.4.4 Ejercicios

Ejercicio 5.8. 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}.

Ejercicio 5.9. El método de trisección para el cálculo de ceros de funciones difiere del método de bisección en que los intervalos se dividen en tres trozos de igual longitud en vez de hacerlo en dos.
Escribe un programa que desarrolle el método de trisección y que tenga como entradas: la función, el intervalo y el error permitido.

5.5 Métodos de iteración funcional

En esta sección tratamos de encontrar una solución aproximada de una ecuación de la forma $x = f(x)$. Es usual referirse a dichas soluciones como *puntos fijos* de la función f. Los puntos fijos de una función f no son más los puntos de intersección de las gráficas de la función f y de la identidad. Por ejemplo, la función de la Figura 5.3 tiene tres puntos fijos.

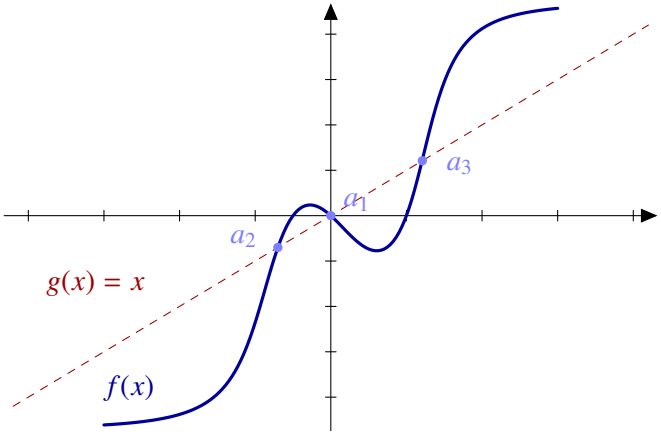


Figura 5.3 Puntos fijos de una función

Hay algunas condiciones sencillas que nos garantizan que una función tiene un único punto fijo.

Teorema 5.8. *Sea* $f : [a, b] \to [a, b]$.

a) Si f *es continua,* f *tiene al menos un punto fijo.*

b) Si f *es derivable y* $|f'(x)| \leq L < 1$, $\forall\, x \in [a, b]$, *entonces tiene un* único *punto fijo.*

Iteración funcional

Para buscar un punto fijo de una función, se elige un punto inicial $x_1 \in [a, b]$ cualquiera y aplicamos la función repetidamente. En otras palabras, consideramos la sucesión definida por recurrencia como

$$x_{n+1} = f(x_n)$$

para $n \geq 1$. Si la sucesión es convergente y llamamos s a su límite, entonces

$$s = \lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} f(x_n) = f(s),$$

o, lo que es lo mismo, s es un punto fijo de la función f. Más aún, en algunos casos es posible controlar el error.

Teorema 5.9. *Sea $f : [a,b] \to [a,b]$ derivable verificando que $|f'(x)| \leq L < 1$, $\forall\, x \in [a,b]$. Sea s el único punto fijo de la función f. Sea $x_1 \in [a,b]$ cualquiera y $x_{n+1} = f(x_n)$, entonces*

$$|x_n - s| \leq \frac{L}{1-L}\,|x_n - x_{n-1}| \leq \frac{L^{n-1}}{1-L}|x_2 - x_1|\,.$$

El método de construcción de la sucesión es lo que se conoce como un método de iteración funcional.

Ejemplo 5.10. Consideremos la función $f(x) = \frac{1}{4}\left(\cos(x) + x^2\right)$ con $x \in [0,1]$. Acotemos la derivada,

$$\left|f'(x)\right| = \left|\frac{-\operatorname{sen}(x) + 2x}{4}\right| \leq \left|\frac{\operatorname{sen}(x)}{4}\right| + \left|\frac{2x}{4}\right| \leq \frac{1}{4} + \frac{2}{4} = \frac{3}{4} < 1.$$

Por tanto, la función f tiene un único punto fijo en el intervalo $[0,1]$. Podemos encontrar el punto fijo resolviendo la correspondiente ecuación.

```
(%i81)  find_root((cos(x)+x^2)/4-x,x,0,1);
(%o81)  .2583921443715997
```

También podemos calcular las iteraciones comenzando en un punto inicial dado de manera sencilla utilizando un bucle

```
(%i82)  define(f(x),(cos(x)+x^2)/4)$
(%i83)  x0:0;
        for i:1 thru 10 do(
          x1:f(x0),print("Iteración ",i," vale ", x1),x0:x1
        );
        0
        Iteración 1 vale 0.25
        Iteración 2 vale .2578531054276612
        Iteración 3 vale .2583569748525884
        Iteración 4 vale .2583898474528139
(%o83)  Iteración 5 vale .2583919943502456
        Iteración 6 vale .2583921345730372
        Iteración 7 vale .2583921437316118
        Iteración 8 vale .2583921443297992
        Iteración 9 vale .2583921443688695
        Iteración 10 vale .2583921443714213
```

Observación 5.11. Existen muchas formas de cambiar una ecuación de la forma $f(x) = 0$ en un problema de puntos fijos de la forma $g(x) = x$. Por ejemplo, consideremos la ecuación $x^2 - 5x + 2 = 0$.

a) Sumando x en los dos miembros

$$x^2 - 5x + 2 = 0 \iff x^2 - 4x + 2 = x,$$

y las soluciones de f son los puntos fijos de $g_1(x) = x^2 - 4x + 2$ (si los tiene).

b) Si despejamos x,

$$x^2 - 5x + 2 = 0 \iff x = \frac{x^2 + 2}{5}$$

y, en este caso, los puntos fijos de la función $g_2(x) = \frac{x^2+2}{5}$ son las soluciones buscadas.

c) También podemos despejar x^2 y extraer raíces cuadradas

$$x^2 - 5x + 2 = 0 \iff x^2 = 5x - 2 \iff x = \sqrt{5x - 2}\,.$$

En este caso, nos interesan los puntos fijos de la función $g_3(x) = \sqrt{5x - 2}$.

Como puedes ver, la transformación en un problema de puntos fijos no es única. Evidentemente, algunas de las transformaciones mencionadas antes dependen de que x sea distinto de cero, mayor o menor que $2/5$, etc. Además de eso las funciones g_i pueden tener mejores o peores propiedades, algunas verificarán las condiciones del teorema anterior y otras no.

5.5.1 Ejercicios

Ejercicio 5.10. Escribe un programa que dada una función, un punto inicial y un número de iteraciones, devuelva la última de ellas.

Ejercicio 5.11. Utiliza el método de iteración con las 3 funciones anteriores empezando en cada uno de los puntos 0.5, 1.5 y 6. ¿En cuáles obtienes convergencia a un punto fijo? ¿Es siempre el mismo?

5.5.2 Representación gráfica con el paquete *dynamics*

Para representar gráficamente los puntos de la sucesión, comenzamos con el primer punto de la sucesión $(x_1, f(x_1))$ y, a partir de ese momento, nos vamos moviendo horizontalmente hasta cruzar la bisectriz y verticalmente hasta encontrar de nuevo la gráfica de la función. Más concretamente,

a) comenzamos con $(x_1, f(x_1))$;

b) nos movemos horizontalmente hasta cortar la bisectriz. El punto de corte será $(f(x_1), f(x_1))$;

c) nos movemos verticalmente hasta cortar a la gráfica de f o, lo que es lo mismo, tomamos $x_2 = f(x_1)$ y le calculamos su imagen. El punto de corte será esta vez $(x_2, f(x_2))$.

d) Repetimos.

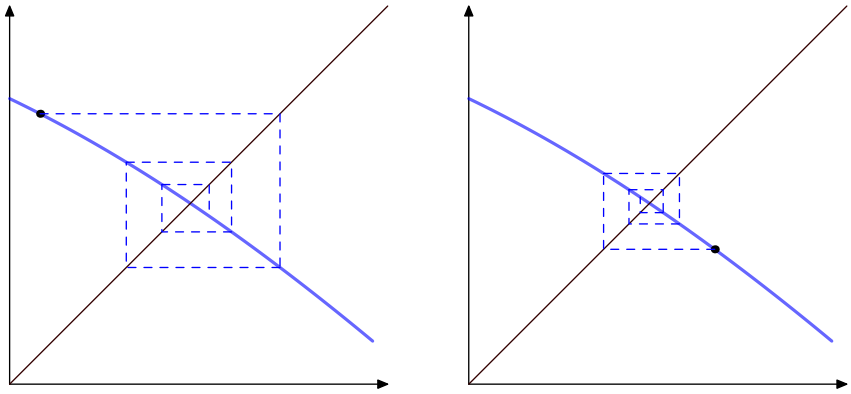


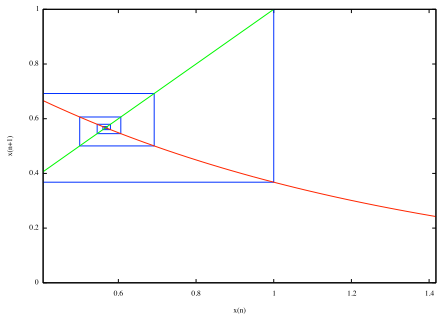
Figura 5.4 Método de iteración funcional

El paquete *dynamics* permite hacer estas representaciones de forma muy sencilla. Primero lo cargamos

```
(%i84)  load(dynamics)$
```

y luego podemos usar los comandos `evolution` y `staircase` indicando el punto inicial y el número de iteraciones. Por ejemplo, el diagrama de escalera para la función e^{-x} tomando como punto inicial 1 y 10 pasos

```
(%i85)  staircase(exp(-x),1,10,[y,0,1]);
(%o85)
```



Observa que hemos añadido `[y,0,1]` para indicar un rango más apropiado que el se dibuja por defecto.

`evolution(`*`func,pto1,pasos,opciones`*`)`	Gráfico de las iteraciones de *func*
`staircase(`*`func,pto1,pasos,opciones`*`)`	Gráfico de escalera de las iteraciones de *func*

5.5.3 Criterios de parada

Suele cuando trabajamos con métodos iterativos que tenemos una sucesión que sabemos que es convergente, pero no conocemos cuál es el valor exacto de su límite. En estos casos lo que podemos hacer es sustituir el valor desconocido del límite por uno de los términos de la sucesión

que haría el papel de una aproximación de dicho límite. Por ejemplo, si consideramos el término general de una sucesión $\{a_n\}_{n\in\mathbb{N}}$ dada, con la ayuda del ordenador podemos calcular un número finito de términos. La idea es pararse en los cálculos en un determinado elemento a_{k_0} para que haga el papel del límite. Se impone entonces un *criterio de parada* para que dicho valor sea una buena aproximación del límite de la sucesión.

Tolerancia

Una forma de establecer un criterio de parada es considerar un número pequeño, al que llamaremos *tolerancia* y denotaremos por T, y parar el desarrollo de la sucesión cuando se de una de las dos circunstancias siguientes:

a) $|a_n - a_{n-1}| < T$,

b) $\dfrac{|\,a_n - a_{n-1}\,|}{|a_n|} < T.$

La primera es el error absoluto y la segunda el error relativo. Suele ser mejor utilizar esta última.

5.5.4 Ejercicios

Ejercicio 5.12. Añade una condición de parada al método de iteración.

5.5.5 El método de Newton-Raphson

El método de Newton-Raphson nos proporciona un algoritmo para obtener una sucesión de puntos que aproxima un cero de una función dada.

La forma de construir los términos de la sucesión de aproximaciones es sencilla. Una vez fijado un valor inicial x_1, el término x_2 se obtiene como el punto de corte de la recta tangente a f en x_1 con el eje OX. De la misma forma, obtenemos x_{n+1} como el punto de corte de la recta tangente a f en el punto x_n con el eje OX. De lo dicho hasta aquí se deduce:

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

Como observarás se trata de un método de iteración funcional. Para comprender el algoritmo observa la Figura 5.5 donde se ve cómo se generan los valores de las aproximaciones.

Para asegurar la convergencia de la sucesión (hacia la solución de la ecuación) usaremos el siguiente resultado.

Teorema de Newton-Raphson

Teorema 5.12. *Sea f una función de clase dos en el intervalo $[a,b]$ que verifica:*
a) $f(a)f(b) < 0$,
b) $f'(x) \neq 0$, para todo $x \in [a,b]$,
c) $f''(x)$ no cambia de signo en $[a,b]$.
Entonces, tomando como primera aproximación el extremo del intervalo $[a,b]$ donde f y f'' tienen el mismo signo, la sucesión de valores x_n del método de Newton-Raphson es convergente hacia la única solución de la ecuación $f(x) = 0$ en $[a,b]$.

Una vez que tenemos asegurada la convergencia de la sucesión hacia la solución de la ecuación, deberíamos decidir la precisión. Sin embargo, veremos que el método es tan rápido en su convergencia que por defecto haremos siempre 10 iteraciones. Otra posibilidad sería detener el cálculo 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).

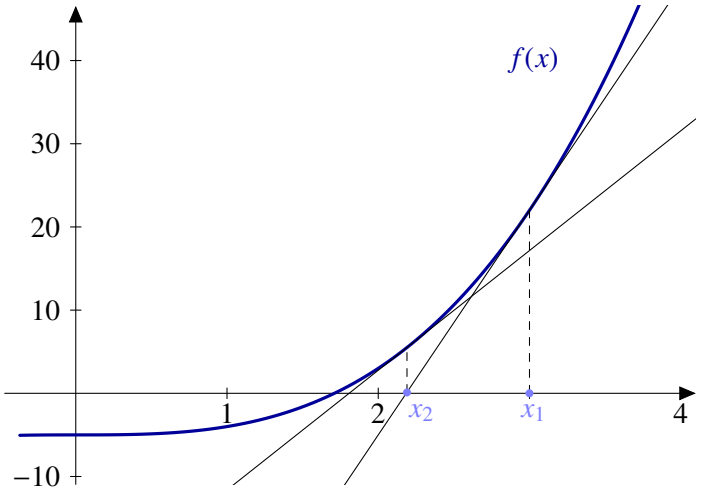
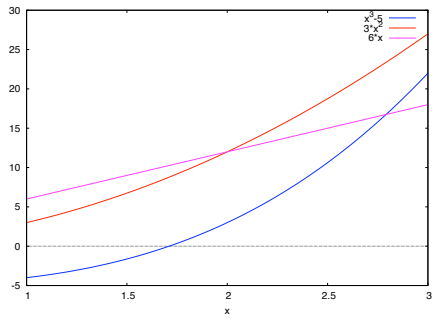


Figura 5.5 Método de Newton-Raphson

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.

```
(%i86)  f(x):=x^3-5$
(%i87)  define(df(x),diff(f(x),x))$
(%i88)  define(df2(x),diff(f(x),x,2))$
(%i89)  plot2d([f(x),df(x),df2(x)],[x,1,3]);
(%o89)
```



A continuación, generaremos los términos de la sucesión de aproximaciones mediante el siguiente algoritmo. Comenzaremos por definir la función f y el valor de la primera aproximación. Inmediatamente después definimos el algoritmo del método de Newton-Raphson, e iremos visualizando las sucesivas aproximaciones. Como dijimos, pondremos un límite de 10 iteraciones, aunque usando mayor precisión decimal puedes probar con un número mayor de iteraciones.

```
(%i90)  y:3.0$
        for i:1 thru 10 do
          (y1:y-f(y)/df(y),
          print(i,"- aproximación",y1),
          y:y1
          );
        1 - aproximación 2.185185185185185
        2 - aproximación 1.80582775632091
        3 - aproximación 1.714973662124988
        4 - aproximación 1.709990496694424
        5 - aproximación 1.7099759468005
        6 - aproximación 1.709975946676697
        7 - aproximación 1.709975946676697
        8 - aproximación 1.709975946676697
        9 - aproximación 1.709975946676697
        10 - aproximación 1.709975946676697
```

Observarás al ejecutar este grupo de comandos que ya en la séptima iteración se han "estabilizado" diez cifras decimales. Como puedes ver, la velocidad de convergencia de este método es muy alta.

El módulo mnewton

El método que acabamos de ver se encuentra implementado en *Maxima* en el módulo `mnewton` de forma mucho más completa. Esta versión se puede aplicar tanto a funciones de varias variables, en otras palabras, también sirve para resolver sistemas de ecuaciones.

Primero cargamos el módulo

```
(%i91)  load(mnewton)$
```

y luego podemos buscar una solución indicando función, variable y punto inicial

```
(%i92)  mnewton(x^3-5,x,3);
(%o92)  [[x=1.709975946676697]]
```

5.5.6 Ejercicios

Ejercicio 5.13.

El método de regula falsi o de la falsa posición es muy parecido al método de bisección. La única diferencia es que se cambia el punto medio por el punto de corte del segmento que une los puntos $(a, f(a))$ y $(b, f(b))$ con el eje de abscisas.
Escribe un programa que utilice este método. Para la función $f(x) = x^2 - 5$ en el intervalo $[0, 4]$, compara los resultados obtenidos. ¿Cuál es mejor?

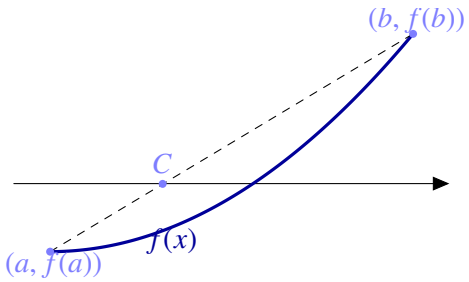


Figura 5.6 Método de regula falsi

Ejercicio 5.14. Reescribe el método de Newton-Raphson añadiendo una condición de salida (cuándo el error relativo o absoluto sea menor que una cierta cantidad) y que compruebe que la primera derivada está "lejos" de cero en cada paso.

Ejercicio 5.15.
El método de la secante evita calcular la derivada de una función utilizando recta secantes. Partiendo de dos puntos iniciales x_0 y x_1, el siguiente es el punto de corte de la recta que pasa por $(x_0, f(x_0))$ y $(x_1, f(x_1))$ y el eje de abscisas. Se repite el proceso tomando ahora los puntos x_1 y x_2 y así sucesivamente.
La convergencia de este método no está garantizada, pero si los dos puntos iniciales están próximos a la raíz no suele haber problemas. Su convergencia es más lenta que el método de Newton-Raphson aunque a cambio los cálculos son más simples.

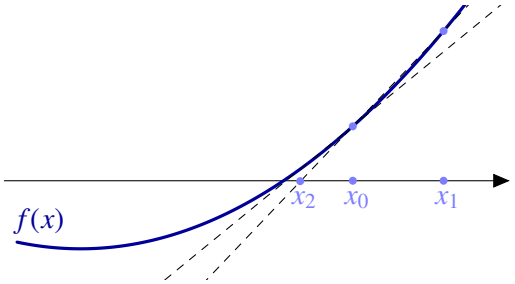


Figura 5.7 Método de la secante

Escribe un programa que utilice este método. Para la función $f(x) = x^2 - 5$, compara los resultados obtenidos con el método de Newton-Raphson. ¿Cuál es mejor?

Ejercicio 5.16. Resuelve las ecuaciones

a) $e^{-x} + x^2 - 3\,\mathrm{sen}(x) = 0$,

b) $e^{|x|} = \arctan(x)$,

c) $x^{15} - 2 = 0$

utilizando los métodos que hemos estudiado. Compara cómo se comportan y decide en cuál la convergencia es más rápida.

Ejercicio 5.17.

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.

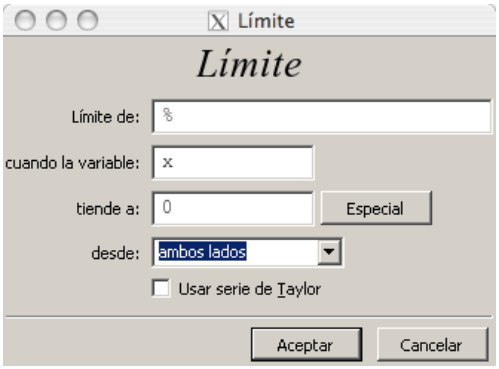
Límites y continuidad

6

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 esto.

6.1 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 π, e o infinito. Además de esto, también podemos marcar si queremos calcular únicamente el límite por la derecha o por la izquierda en lugar de la opción por defecto que es por ambos lados.



`limit (expr,x,a)`	$\lim_{x\to a} expr$
`limit (expr,x,a,plus)`	$\lim_{x\to a^+} expr$
`limit (expr,x,a,minus)`	$\lim_{x\to a^-} expr$
`inf`	$+\infty$
`minf`	$-\infty$
`und`	indefinido
`ind`	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);
(%o2)  -∞
```

aplicar las reglas de L'Hôpital,

```
(%i3)  limit(sin(x)/x,x,0);
(%o3)  1
```

Incluso es capaz de dar alguna información en el caso de que no exista el límite. Por ejemplo, sabemos que las funciones periódicas, salvo las constantes, no tienen límite en ∞. La respuesta de *Maxima* cuando calculamos el límite de la función coseno en ∞ es

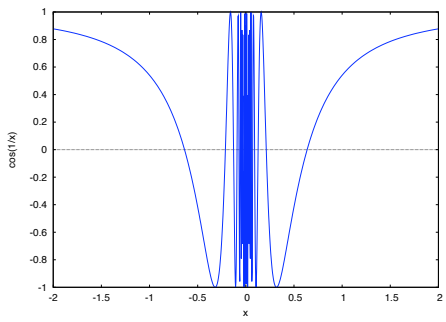
```
(%i4)  limit(cos(x),x,inf);
(%o4)  ind
```

Indeterminado. Este límite es equivalente a

```
(%i5)  limit(cos(1/x),x,0);
(%o5)  ind
```

La función $\cos\left(\frac{1}{x}\right)$ oscila cada vez más rápidamente cuando nos acercamos al origen. Observa su gráfica.

```
(%i6)  plot2d([cos(1/x)], [x,-2,2]);
(%o6)
```



ind **und** *Maxima* tiene dos formas de indicar indeterminación. Una es `ind`, para indicar que está acotado, y la otra es `und`, para indicar indeterminación a secas. Ahora bien, mucho cuidado con pensar que si la respuesta es `und` entonces la función es no acotada. Esto puede ser cierto o no

```
(%i7)  limit(abs(x)/x,x,0);
(%o7)  und
```

Observación 6.1. 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

```
(%i8)  limit(abs(x)/x,x,0,plus);
(%o8)  1
(%i9)  limit(abs(x)/x,x,0,minus);
(%o9)  -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.

¿Infinito o infinitos?

Maxima diferencia entre "infinitos reales" e "infinitos complejos". ¿Qué quiere decir esto? Veamos un ejemplo. Si calculamos el límite de la función $1/x$ en 0 inmediatamente pensamos que el resultado depende de si calculamos el límite por la izquierda o por la derecha. En efecto,

```
(%i10)  limit(1/x,x,0,plus);
(%o10)  ∞
(%i11)  limit(1/x,x,0,minus);
(%o11)  -∞
```

Pero, ¿qué ocurre si no estudiamos límites laterales?

```
(%i12)  limit(1/x,x,0);
(%o12)  infinity
```

La constante *infinity* representa "infinito complejo". Esto quiere decir que en módulo el límite es infinito. **infinity**

6.2 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

```
(%i13)  f(n):=(1+1/n)^n;
(%o13)  f(n):=(1+1/n)^n
```

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

```
(%i14)  limit(f(n), n, inf);
(%o14)  %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:

```
(%i15)  c[1]:1;
(%o15)  1
(%i16)  c[n]:=c[n-1]/(1+c[n-1]);
(%o16)  c_n:=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:

```
(%i17)  limit(c[n],n,inf);
        Maxima encountered a Lisp error:
          Error in PROGN [or a callee]: Bind stack overflow.
        Automatically continuing.
        To reenable 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*

```
(%i18)  solve(L/(1+L)=L,L);
(%o18)  [L=0]
```

con lo que tendríamos demostrado que el límite es 0.

Observación 6.2. 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.

```
(%i19)  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=(n+%k_1+1)/(n+%k_1)-1
```

que, simplificando, nos queda

```
(%i3)  ratsimp(%);
(%o3)  c_n=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**
`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,

```
(%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(1+sqrt(1+sqrt(1+sqrt(1+sqrt(1+sqrt(1+sqrt(1+sqrt(2)))))))+1+1+1+1+1+1+1)
(%i7)  %,numer;
(%o7)  1.618016542231488
```

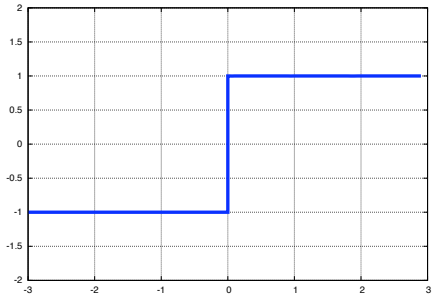
pero seremos nosotros los que tendremos que demostrar la convergencia estudiando la monotonía y la acotación de la sucesión.

6.3 Continuidad

El estudio de la continuidad de una función es inmediato una vez que sabemos calcular límites. Una función $f : A \subset \mathbb{R} \to \mathbb{R}$ es continua en $a \in A$ si $\lim_{x\to a} f(x) = f(a)$. 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.

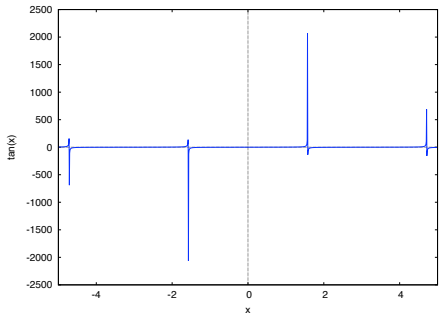
Con respecto a las funciones discontinuas, la gráfica puede darnos una idea del tipo de discontinuidad. Si la discontinuidad es evitable, es difícil apreciar un único pixel en la gráfica. Una discontinuidad de salto es fácilmente apreciable. Por ejemplo, la función signo, esto es, $\frac{|x|}{x}$, tiene 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);
(%o9)
```



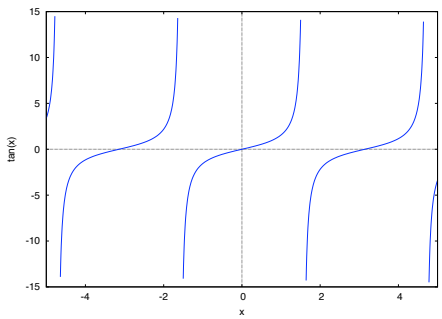
Cuando el salto es infinito o, lo que es lo mismo, cuando la función tiene una asíntota vertical, la primera dificultad que se encuentra *Maxima* es escoger un rango adecuado para representarla:

```
(%i10)  plot2d(tan(x),[x,-5,5]);
(%o10)
```



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

```
(%i11)  plot2d(tan(x),[x,-5,5],[y,-15,15]);
(%o11)
```



6.4 Ejercicios

Ejercicio 6.1. Estudia la continuidad de la función $f\colon \mathbb{R} \to \mathbb{R}$ definida como $f(x) = x * \ln|x|$ si $x \neq 0$ y $f(0) = 0$.

Ejercicio 6.2. 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 6.3. 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 6.4.

a) Dibuja una función continua cuya imagen no sea un intervalo.

b) Dibuja una función definida en un intervalo cuya imagen sea un intervalo y que no sea continua.

c) 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 6.5. Consideremos la función $f\colon [0, 1] \to \mathbb{R}$ definida como $f(x) = \frac{1}{2}(\cos(x) + \operatorname{sen}(x))$.

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

7

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.

7.1 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**

`diff(expr,variable)`	derivada de *expr*
`diff(expr,variable,n)`	derivada *n*-ésima de *expr*

A este comando también podemos acceder a través del menú **Análisis→Derivar** o, también, a través del botón Derivar si hemos activado el correspondiente panel. Haciéndolo de cualquiera de estas dos formas, aparece una ventana de diálogo con varios datos a rellenar; a saber:

Figura 7.1 Introducir derivada

a) 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.

b) respecto la variable. Se refiere a la variable respecto a la cual vamos a derivar.

c) 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)
```

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
```

7.1.1 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 7.1.** 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 ? , es decir, son dos apóstrofos ,'+', y no hay que confundirla con las dobles comillas de la tecla 2 .

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)  d/dx (sin(x^2)+x^4)=2 x cos(x^2)+4 x^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 7.2. 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.

7.2 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,

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

límite que denotamos $f'(a)$.

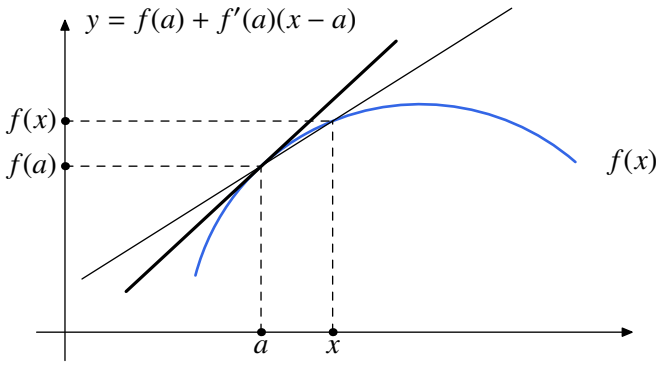


Figura 7.2 Recta tangente

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,

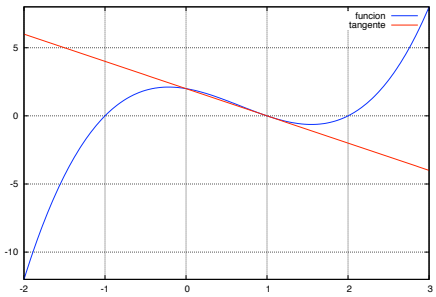
```
(%i19)  f(x):=x^3-2*x^2-x+2;
(%o19)  f(x):=x^3-2x^2-x+2
(%i20)  define(df(x),diff(f(x),x));
(%o20)  df(x):=3x^2-4x-1
```

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);
(%o23)
```



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

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

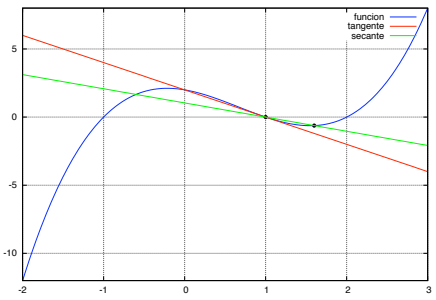
La definimos en *Maxima*:

```
(%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 la 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)$
(%o25)
```



Si te fijas, en la gráfica anterior hemos añadido también el par de puntos que definen la recta sectante.

En la Figura 7.3 puedes ver el resultado para $h = 0.2, 0.4, 0.6$, y 0.8.

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 $h = 0.1, 0.2, \ldots, 2$:

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

7.2.1 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.

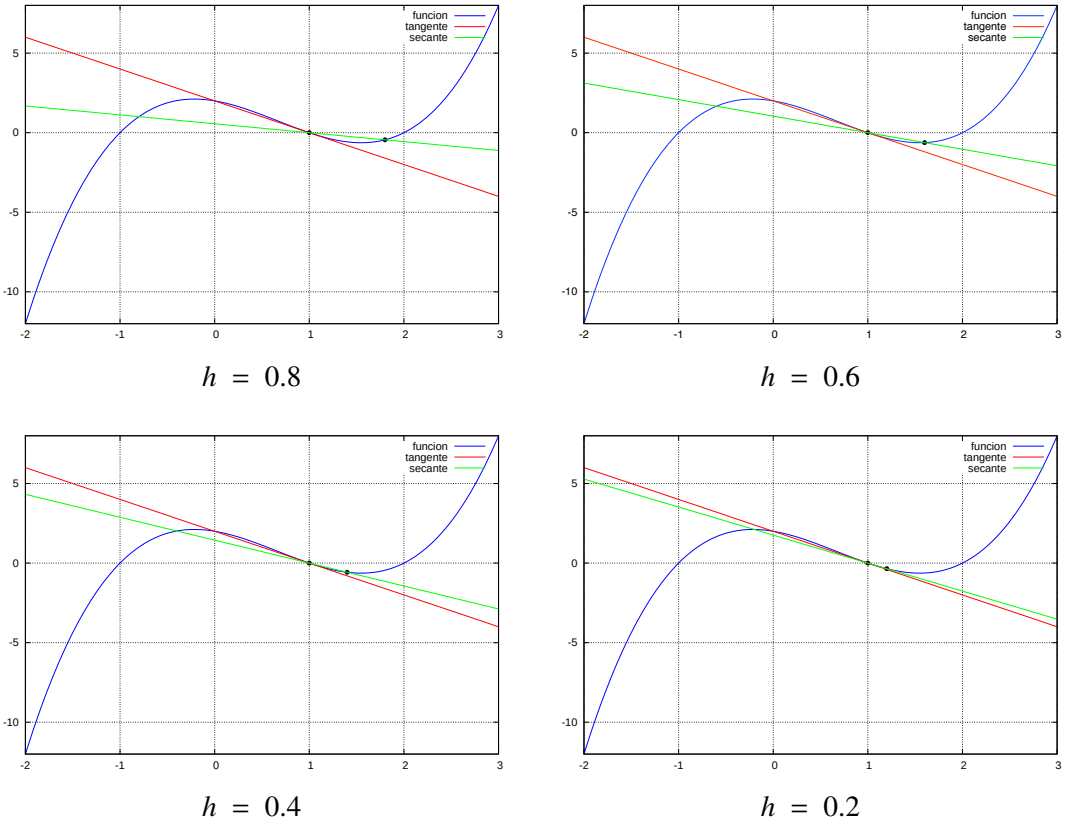
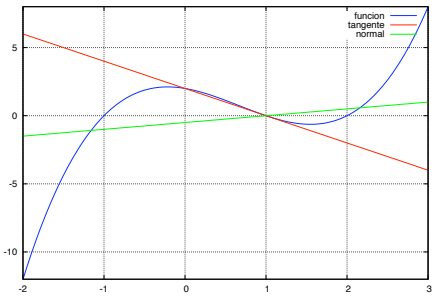


Figura 7.3 Rectas secantes y tangente

```
(%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)$
```

(%o28)



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.

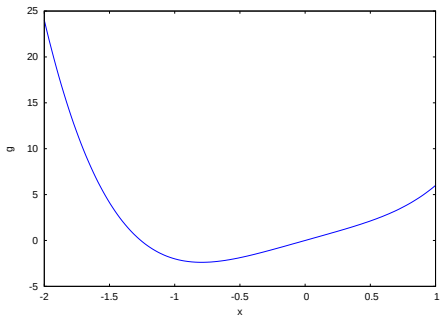
Prueba a cambiar la función f, el punto a. ¡Ahora es tu turno! Por ejemplo, haz esto mismo para una función que no sea derivable como $f(x) = \sqrt{|x|}$ en el origen.

7.3 Máximos y mínimos relativos

En esta sección vamos a aprender a localizar extemos 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 7.3. 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,-2,1]);
(%o30)
```



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

```
(%i31)  define(d1f(x),diff(f(x),x))$
(%i32)  puntosf:solve(d1f(x),x);
(%o32)  [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.

```
(%i33)  last(puntosf);
(%o33)  x=-1/2^(1/3)
```

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

```
(%i34)  a:rhs(%)$
(%i35)  define(d2f(x),diff(f(x),x,2))$
(%i36)  d2f(a);
```

```
(%o36)   24/2^(2/3)
```

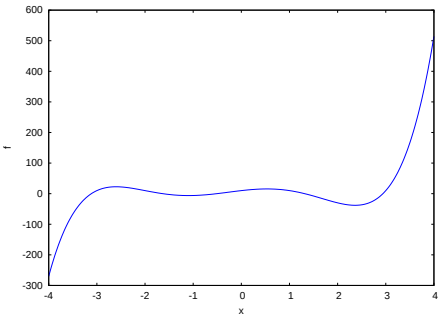
Por tanto la función f tiene un mínimo relativo en dicho punto a. ¿Puede haber otro extremo más? ¿Cómo podemos asegurarnos?

Ejemplo 7.4. 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.

```
(%i37)   f(x):=x^5+x^4-11*x^3-9*x^2+18*x+10$
(%i38)   plot2d(f(x),[x,-4,4])$
(%o38)
```



A simple vista observamos que hay:

a) un máximo relativo entre -3 y -2,

b) un mínimo relativo entre -2 y -1,

c) un máximo relativo entre 0 y 1, y

d) 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`.

```
(%i39)   define(d1f(x),diff(f(x),x));
(%o39)   d1f(x):=5x^4+4x^3-33x^2-18x+18
(%i40)   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.

```
(%i41)  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)];
(%o41)  [-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:

```
(%i42)  define(d2f(x),diff(f(x),x,2));
(%o42)  d2f(x):=20x^3+12x^2-66x-18
(%i43)  map(d2f,puntosf);
(%o43)  [-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.

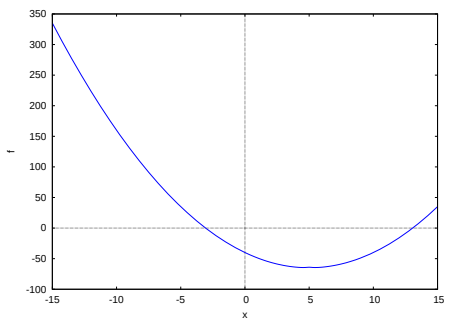
A continuación veremos un caso en el que la gráfica diseñada por el programa nos puede llevar a engaño. Descubriremos el error gracias al "test de la segunda derivada".

Ejemplo 7.5. 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 $[-15, 15]$

```
(%i44)  f(x):=x^2-10*x-40+1/(10*x^2-100*x+251)$
(%i45)  plot2d(f(x),[x,-15,15]);
(%o45)
```

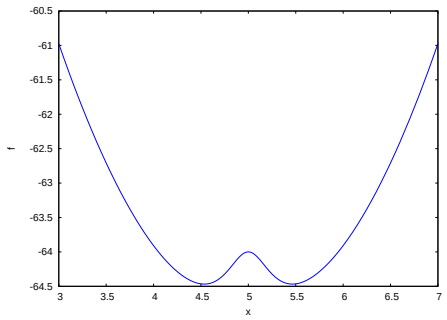


Aparentemente hay un mínimo cerca de 5. De hecho, si aplicamos el comando `solve` a la primera derivada de f obtenemos que precisamente la derivada de f se anula en $x = 5$. A la vista de la gráfica parece que efectivamente en $x = 5$ hay un mínimo de la función f. Pero nada más lejos de la realidad, 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:

```
(%i46)  define(d1f(x),diff(f(x),x))$
(%i47)  d1f(5);
(%o47)  0
(%i48)  define(d2f(x),diff(f(x),x,2))$
(%i49)  d2f(5);
(%o49)  -18
```

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

```
(%i50)  plot2d(f(x),[x,3,7]);
(%o50)
```



En resumen, ten cuidado con los resultados que obtengas y, siempre, comprueba si se ajustan a los contenidos teóricos que has visto en clase.

En el siguiente ejemplo, resolvemos un problema de optimización.

Ejemplo 7.6. 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:

```
(%i51)  altura:solve(x+2*h+(%pi*x)/2=A,h)$
(%o51)  [h=(2 A+(-π-2) x)/4]
```

La luminosidad viene dada por

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

Definimos entonces la función f y calculamos sus puntos críticos:

```
(%i52)  f(x):=A*x-(1/8)*(8+3*%pi)*x^2$
(%i53)  define(d1f(x),diff(f(x),x))$
(%i54)  solve(diff(f(x),x),x);
(%o54)  [x=4A/(3π+8)]
(%i55)  %[1];
(%o55)  x=4A/(3π+8)
```

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

```
(%i56)  a:rhs(%)$
(%i57)  define(d2f(x),diff(f(x),x,2))$
(%i58)  d2f(a);
(%o58)  -(3π+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 7.7. 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:

```
(%i59)  f(x):=log(x+1)-x/(x+1)$
(%i60)  define(d1f(x),diff(f(x),x))$
(%i61)  solve(d1f(x),x);
(%o61)  [x=0]
```

Por tanto sólo tiene un punto crítico, y además:

```
(%i62)  define(d2f(x),diff(f(x),x,2))$
(%i63)  d2f(0);
(%o63)  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.

7.4 Ejercicios

Ejercicio 7.1.

a) Calcula las tangentes a la hipérbola $xy = 1$ que pasan por el punto $(-1, 1)$. Haz una representación gráfica.

b) Calcula las tangentes a la gráfica de $y = 2x^3 + 13x^2 + 5x + 9$ que pasan por el origen. Haz una representación gráfica.

Ejercicio 7.2. Dibuja la semicircunferencia superior centrada en el origen y de radio 2. Dibuja el punto en el que la recta tangente a dicha circunferencia pasa por el punto $(3, 7)$, así como dicha recta tangente.

Ejercicio 7.3. 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 7.4. 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 7.5. Halla dos números no negativos tales que la suma de uno más el doble del otro sea 12 y su producto sea máximo.

Ejercicio 7.6. 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?

Integración

8

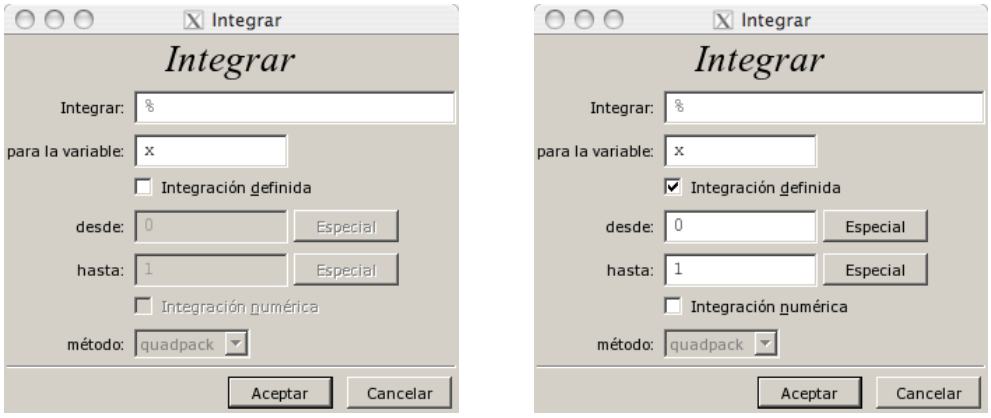
En este capítulo vamos a aprender a calcular integrales en una variable, así como aplicar estas integrales al cálculo de áreas de recintos en el plano limitados por varias curvas, longitudes de curvas, volúmenes de cuerpos de revolución y áreas de superficies de revolución.

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.

`integrate(f(x),x)`	primitiva de la función $f(x)$
`integrate(f(x),x,a,b)`	$\int_a^b f(x)\,dx$

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.



Integral indefinida Integral definida

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);
(%o2)  -log(x+1)/4 - atan(x)/2 + log(x-1)/4
```

irracionales,

```
(%i3)  integrate(sqrt(x^2+1),x);
(%o3)  asinh(x)/2+x*sqrt(x^2+1)/2
```

ya sea aplicando integración por partes

```
(%i4)  integrate(x^3*%e^x,x);
(%o4)  (x^3-3*x^2+6*x-6)*%e^x
```

o el método que considere necesario

```
(%i5)  integrate(%e^(-x^2),x);
(%o5)  sqrt(%pi)*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);
(%o8)  -log(6)/2+log(4)/2+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)  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)  integrate(%e^(x^3+x),x)
```

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)  π
(%i13)  integrate(%e(-x^2),x,0,inf);
(%o13)  √π/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

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

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

⚠ **Observación 8.1.** Hay que tener cierto cuidado con la integración de funciones que no son continuas. Por ejemplo, la función $f(x) = \frac{1}{x}$ no es impropiamente integrable en $[-1, 1]$. Si así fuera, también lo sería en el intervalo $[0, 1]$ y esto es claramente falso.
Dependiendo de las versiones, la respuesta de *Maxima* puede variar, pero siempre involucra el llamado *valor principal* de la integral. Por ejemplo, con la versión 5.17 de *Maxima* el resultado es el siguiente

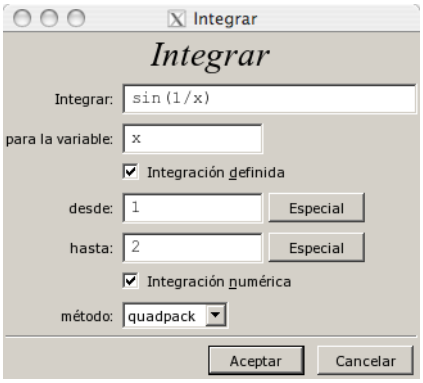
```
(%i16)  integrate(1/x,x,-1,1);
Principal value
(%o16)  log(-1)+%iπ
```

8.1.2 Integración númerica

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.

`quad_quags(f(x),x,a,b)`	aproximación numérica de $\int_a^b f(x)\,dx$
`quad_qagi(f(x),x,a,b)`	aproximación numérica de $\int_a^b f(x)\,dx$
`romberg(f(x),x,a,b)`	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 $\pi - 1$

```
(%i17)  quad_qags(x*sin(x),x,%pi/2,%pi);
(%o17)  [2.141592653589794,2.3788064330872769 10^-14,21,0]
(%i18)  romberg(x*sin(x),x,%pi/2,%pi);
(%o18)  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:

a) la aproximación de la integral
b) el error absoluto estimado de la aproximación
c) el número de evaluaciones del integrando
d) 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 $f(x) = e^{-x^2}$.

```
(%i19)  integrate(exp(-x^2),x);
(%o19)  sqrt(%pi) erf(x)
        ----------------
                2
(%i20)  integrate(exp(-x^2),x,0,1);
(%o20)  sqrt(%pi) erf(1)
        ----------------
                2
```

Si ahora queremos, podemos calcular numéricamente este valor,

```
(%i21)  float(%);
(%o21)  0.74682413281243
```

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

```
(%i22)  quad_qags(exp(-x^2),x,0,1);
(%o22)  [0.74682413281243,8.2954620176770253 10^-15,21,0]
(%i23)  romberg(exp(-x^2),x,0,1);
(%o23)  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

```
(%i24)  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
(%o24)  [-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 $\int_0^1 \frac{\text{sen}(1/x)}{x}$.

```
(%i25)  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
(%o25)  [-1.050233246377689,0.20398634967385,8379,5]
(%i26)  romberg((1/x)*sin(1/x),x,0,1);
(%o26)  romberg(sin(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 `quadqags` 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 $f : [a,b] \to \mathbb{R}$ 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 $P = \{a = x_0 < x_1 < \ldots < x_n = b\}$ del intervalo. Luego calculábamos el supremo y el ínfimo de f en cada uno de los intervalos $I_i = [x_{i-i}, x_i]$ 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.

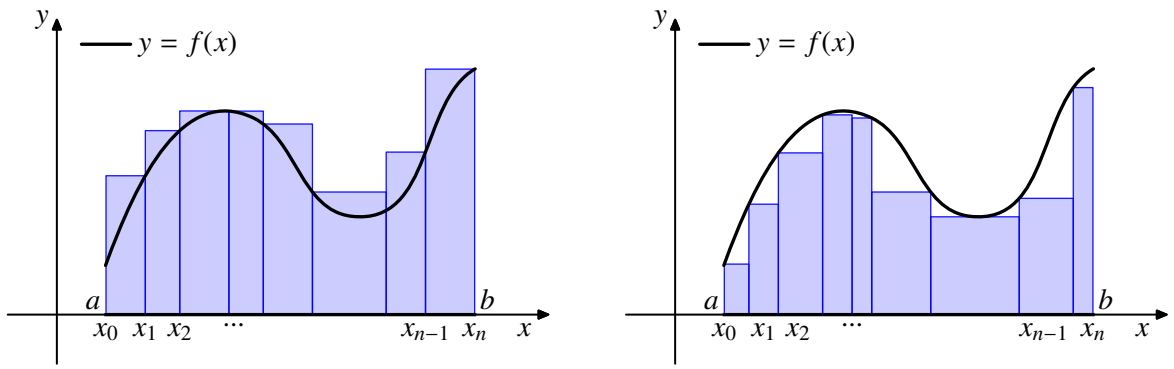


Figura 8.2 Sumas superiores e inferiores

Las dos áreas que tienes marcadas en la Figura 8.2 son

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

donde $\ell\left(I_i\right)$ 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: $\left[a,a+\frac{b-a}{3}\right]$. 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,

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

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:

```
(%i27)  a:-2;b:5;n:10
(%o28)  -2
(%o29)  5
(%o29)  10
(%i30)  f(x):=(x-1)*x*(x-3);
(%o30)  f(x):=(x-1)x(x-3)
```

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

```
(%i31)  makelist(a+i*(b-a)/n,i,0,n);
(%o31)  [-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 8.2. *Sea $f : [a, b] \to \mathbb{R}$ integrable. Sea P_n la partición*

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

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

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

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.

```
(%i32)  puntos:makelist(a+(i-1)*(b-a)/n+random(1.0)*(b-a)/n,i,1,n);
(%o32)  [-1.974524678870055,-0.86229173716074,0.095612020508637,
        0.12911247230811,1.242837772556621,1.915206527061883,
        2.658260436280155,3.229497471823565,3.636346487565163,
        4.30162951626272]
```

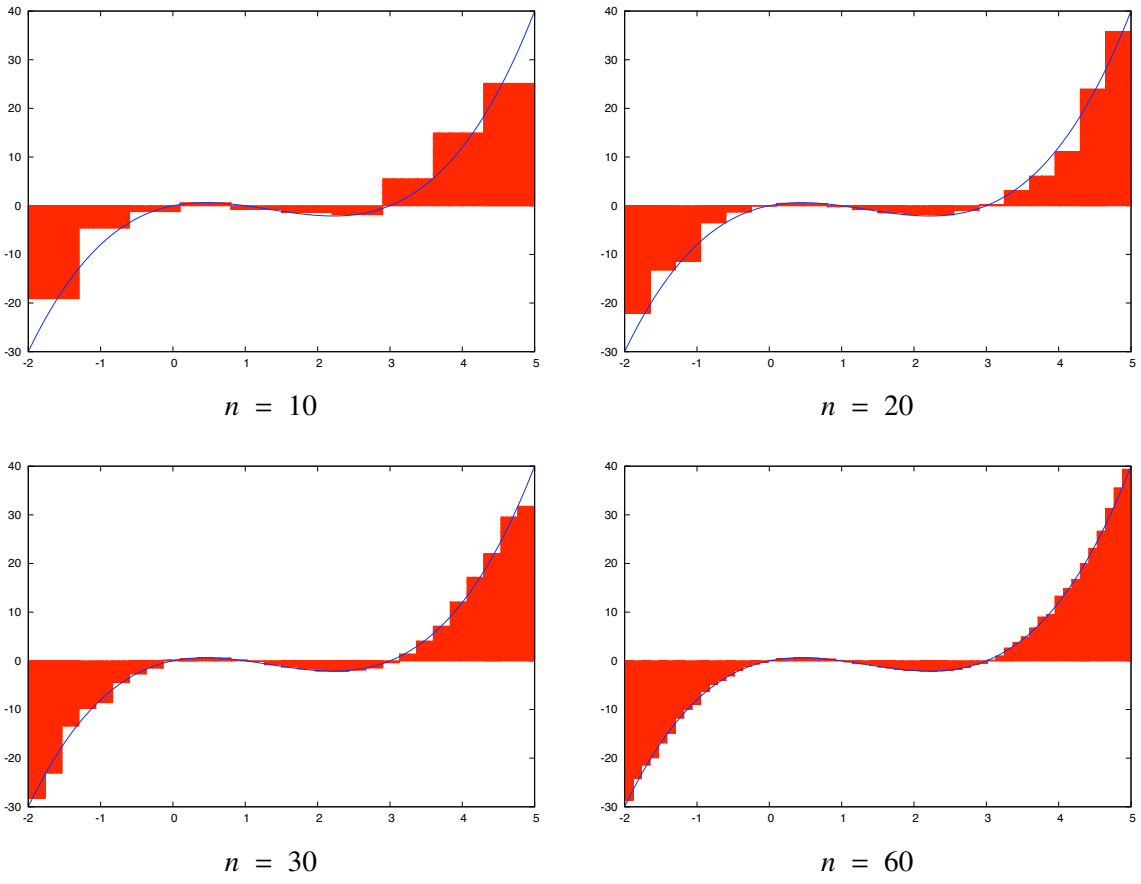


Figura 8.3 Convergencia de las sumas de Riemann

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`:

```
(%i33)  sum((b-a)/n*(f(a+(i-1)*(b-a)/n+random(1.0)*(b-a)/n)),i,1,n);
(%o33)  5.859818716400653
```

que si lo comparamos con el valor de dicha integral

```
(%i34)  integrate(f(x),x,a,b);
(%o34)  77/12
(%i35)  %,numer
(%o35)  6.416666666666667
```

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

```
(%i36)  n:100;
(%o36)  100
(%i37)  sum((b-a)/n*(f(a+(i-1)*(b-a)/n+random(1.0)*(b-a)/n)),i,1,n);
(%o37)  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

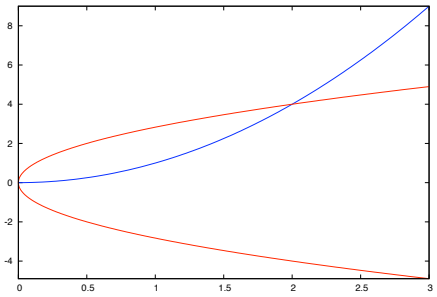
$$\int_a^b |f(x)-g(x)|\,dx$$

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 8.3. 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.

```
(%i38)  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)
        )
```

(%o38)



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

```
(%i39)  solve([y=x^2,y^2=8*x],[x,y]);
(%o39)  [[x=2,y=4],[x=-√3 %i-1,y=2√3 %i-2],[x=√3 %i-1,y=-2√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

```
(%i40)  integrate(sqrt(8*x)-x^2,x,0,2);
(%o40)  8/3
```

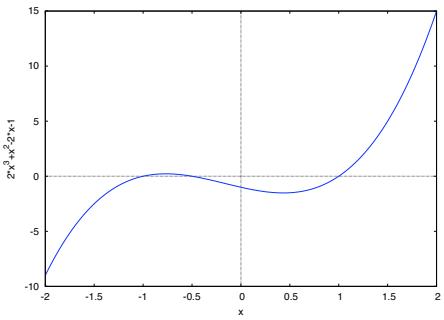
Observación 8.4. La fórmula que define el cálculo del área entre dos funciones tiene un valor absoluto que hace difícil calcular una primitiva directamente. Ese es el motivo por el que dividimos el intervalo en trozos: quitar ese valor absoluto. Si sólo pretendemos calcular el valor numéricamente, entonces el valor absoluto no es un impedimento y podemos calcular directamente la integral olvidándonos de puntos de corte o de qué función es mayor:

```
(%i41)  quad_qags(abs(x^2-sqrt(8*x)),x,0,2);
(%o41)  [2.666666666666668,2.9605947323337525 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 8.5. 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

```
(%i42)  f(x):=2x^3+x^2-2x-1$
(%i43)  plot2d(f(x),[x,-2,2]);
(%o43)
```



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:

```
(%i44)  solve(f(x)=0,x);
(%o44)  [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 $\left]-1,-\frac{1}{2}\right[$ 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 $\left]-1,-\frac{1}{2}\right[$ 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 $\left]-1,-\frac{1}{2}\right[$

```
(%i45)  f(-0.75);
(%o45)  0.21874999813735
```

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

```
(%i46)  f(0);
(%o46)  -1
```

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

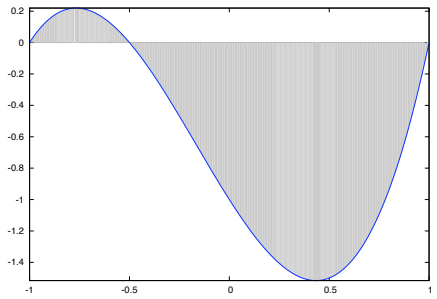
$$\int_{-1}^{1} |f(x)|\,dx = \int_{-1}^{-\frac{1}{2}} f(x)\,dx - \int_{-\frac{1}{2}}^{1} f(x)\,dx,$$

o, lo que es lo mismo,

```
(%i47)  integrate(f(x),x,-1,-1/2)+integrate(-f(x),x,-1/2,1);
(%o47)  71/48
```

Observación 8.6. 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:

```
(%i48)  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);
(%o48)
```



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.

```
(%i49)  f(x):=sqrt(1-x^2);
(%o49)  f(x):=sqrt(1-x^2)
(%i50)  diff(f(x),x);
```

```
(%o50)  -x/sqrt(1-x^2)
(%i51)  integrate(sqrt(1+diff(f(x),x)^2),x,-1,1);
(%o51)  π
```

Observa que hemos calculado la longitud de media circunferencia, ya que la longitud de la circunferencia completa de radio 1 es 2π.

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

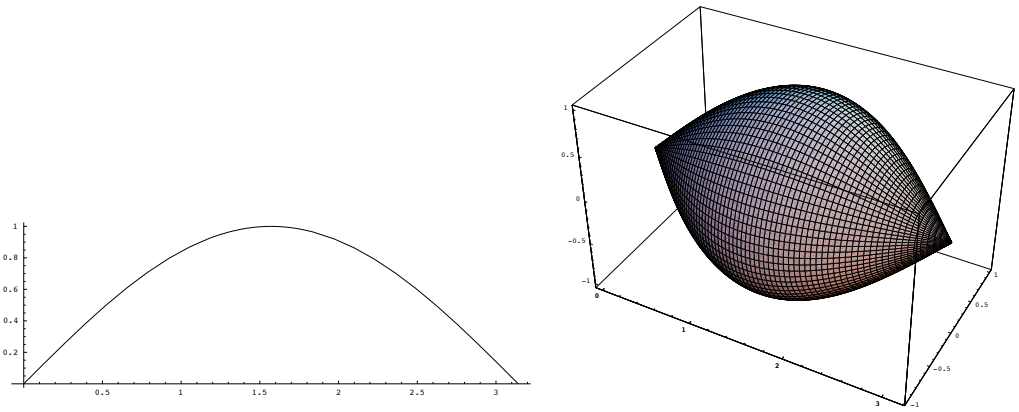
$$\text{longitud} = \int_a^b \sqrt{f'(x)^2 + g'(x)^2}\, dx$$

Por ejemplo, la circunferencia unidad la podemos parametrizar como $t \mapsto (\cos(t), \text{sen}(t))$ con $0 \leq t \leq 2\pi$. Por tanto, la longitud de dicha circunferencia es

```
(%i52)  integrate(sqrt(sin(t)^2+cos(t)^2),x,0,2*%pi);
(%o52)  2π
```

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 π.



Función seno Función seno girada

Figura 8.4

Giro de una curva $y = f(x)$ respecto al eje X

Sea $f : [a, b] \to \mathbb{R}$ 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 Ω con volumen igual a

$$\text{Vol}(\Omega) = \pi \int_a^b f(x)^2\, dx$$

Veamos algunos ejemplos.

Ejemplo 8.7. 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 :

```
(%i53)  %pi*integrate(R^2-x^2,x,-R,R);
(%o53)  4πR^3/3
```

Ejemplo 8.8. 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

```
(%i54)  %pi*integrate(R^2*x^2/h^2,x,0,h);
(%o54)  πhR^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.

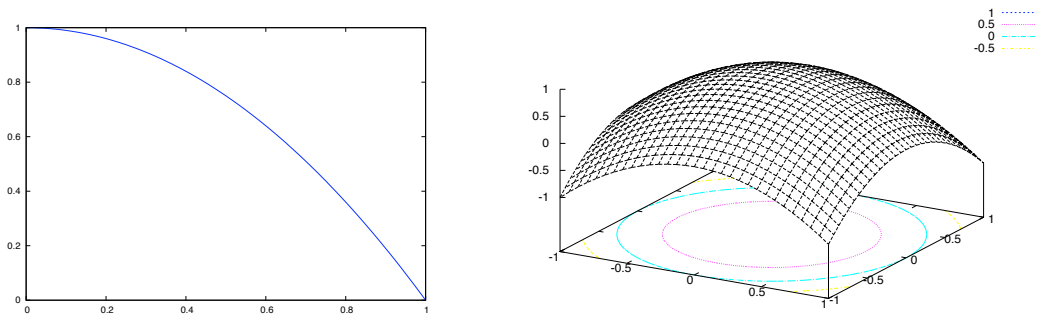


Figura 8.5 Función $1 - x^2$ girada respecto al eje OY

Pues bien, el volumen de la región así obtenida, Ω, viene dado por

$$\text{Volumen}(\Omega) = 2\pi \int_a^b x\, f(x)\, dx$$

Ejemplo 8.9.

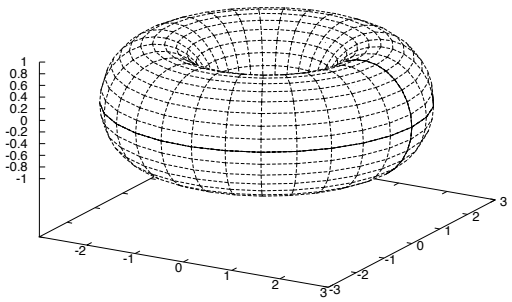


Figura 8.6 Toro

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

$$y = \sqrt{R^2 - (x-a)^2},$$

con $a - R \le x \le a + R$ 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*:

```
(%i55)  4*%pi*integrate(x*sqrt(R^2-(x-a)^2),x,a-R,a+R);
(%o55)  2π²aR²
```

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 $(\cos(t), \text{sen}(t))$, pero que tenemos que trasladar dos unidades. Ése es el motivo de sumar 2 en la siguiente representación en coordenadas paramétricas.

```
(%i56)  with_slider_draw3d(
        n,makelist(0.1*%pi*i,i,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.

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 Ω obtenida al girar respecto al eje OX una función f. El área al girar f en el intervalo $[a, b]$ es

$$\text{área}(\Omega) = 2\pi \int_a^4 f(x)\sqrt{1 + f'(x)^2}\, dx$$

Por ejemplo, una esfera de radio 1 la podemos obtener girando respecto del eje OX la función $\sqrt{1 - x^2}$.

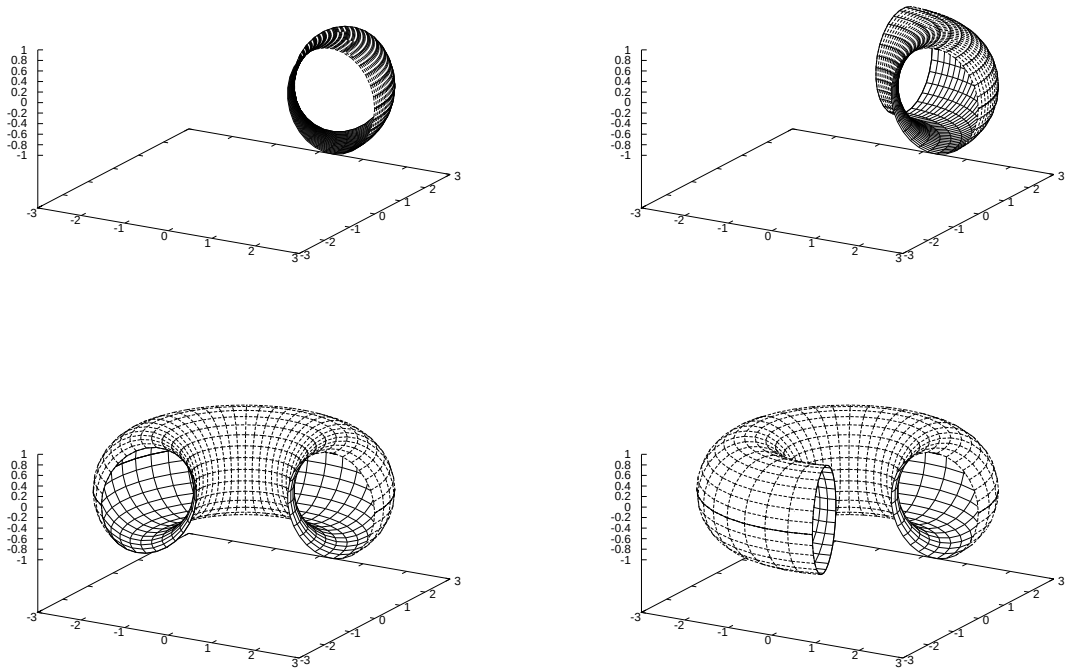


Figura 8.7 Cómo conseguir un toro girando una circunferencia

```
(%i57)  integrate(f(x)*sqrt(1+diff(f(x),x)^2), x, -1, 1);
(%o57)  2 * π^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 centrados en cero de la función coseno.

Teorema fundamental del Cálculo

Ejercicio 8.3. Calcula la derivada de la función $f(x) = \int_{\sqrt{x}}^{x^2+1} \operatorname{sen}(t)\, dt$.

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:

a) área limitada por $y = xe^{-x^2}$, el eje OX, la ordenada en el punto $x = 0$ y la ordenada en el máximo.
b) área de la figura limitada por la curva $y = x^3 - x^2$ y el eje OX.
c) área comprendida entre la curva $y = \tan(x)$, el eje OX y la recta $x = \pi/3$.
d) área del recinto limitado por las rectas $x = 0$, $x = 1$, $y = 0$ y la gráfica de la función $f : \mathbb{R} \to \mathbb{R}$ definida por $f(x) = \frac{1}{(1+x^2)^2}$.
e) las dos áreas en los que la función $f(x) = |x| - x\operatorname{sen}(x)e^x$ divide a la bola unidad $x^2 + y^2 = 1$.

Ejercicio 8.6. Calcula el área entre las curvas:

a) $y = -x^2 - 2x$, $y = x^2 - 4$, para $-3 \leq x \leq 1$.
b) $y^2 = x$, $x^2 + y^2 = 8$.
c) $x = 12y^2 - 12y^3$, $x = 2y^2 - 2y$.
d) $y = \sec^2(x)$, $y = \tan^2(x)$, $-\pi/4 \leq x \leq \pi/4$.
e) $(y - x)^2 = x - 3$, $x = 7$.
f) $y = x^4 + x^3 + 16x - 4$, $y = x^4 + 6x^2 + 8x - 4$.
g) $y = xe^{-x}$, $y = x^2e^{-}x$, $x \geq 0$.

Ejercicio 8.7. Divide un círculo de radio 1 en tres trozos del mismo área usando dos líneas verticales, simétricas respecto del eje OY.

Ejercicio 8.8. Una cabra está atada en el borde de un prado circular de 10 metros de diámetro. Calcula la longitud mínima de cuerda para que la cabra tenga acceso a la mitad de la superficie.

Ejercicio 8.9. Se considera la función $f(x) = x \operatorname{sen} x \cos x$.

a) Calcula dos puntos de su gráfica, $(a, f(a))$ y $(b, f(b))$, por la condición de que la tangente en cada uno de ellos pase por $(4, 0)$.

b) Representa juntas gráficamente a la función y las dos rectas tangentes antes calculadas. Debes elegir un intervalo y opciones de color y grosor de línea apropiados.

c) Calcula el área de la región limitada superiormente por la gráfica de f e inferiormente por las rectas tangentes cuando $a \leq x \leq b$, donde a y b son los valores calculados en el primer apartado.

Longitud de curvas

Ejercicio 8.10.

a) Calcula la longitud del arco de la cicloide $x = t - \operatorname{sen}(t)$, $y = 1 - \cos(t)$, $(0 \leq t \leq 2\pi)$.
b) Calcula la longitud del arco de curva $y = x^2 + 4$ entre $x = 0$ y $x = 3$.

Ejercicio 8.11. Calcula la longitud de una circunferencia de radio arbitrario R.

Ejercicio 8.12. 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 8.13. Calcular el volumen del sólido generado al rotar respecto al eje OX las siguiente curvas:

a) $y = \sec(x),\ x \in [0, \frac{\pi}{3}]$
b) $y = \sqrt{\cos(x)},\ x \in [0, \frac{\pi}{2}]$
c) $y = 9 - x^2$
d) $y = e^x,\ x \in [0, \ln(3)]$

Ejercicio 8.14. Calcular el volumen del sólido generado al rotar respecto al eje OY las siguiente curvas:

a) $y = 1/x,\ x \in [1, 3]$
b) $y = \frac{1}{1+x^2},\ x \in [0, 1]$
c) $y = e^{x^2},\ x \in [1, \sqrt{3}]$

Ejercicio 8.15.
a) Calcular la superficie de una esfera de radio R.
b) Calcular la superficie de la figura que se obtiene al girar la función $y = \tan(x),\ x \in [0, \pi/4]$ alrededor del eje OX.

Ejercicio 8.16.
a) Calcular el volumen del toro engendrado al girar el círculo de centro $(2, 0)$ y radio 3 alrededor del eje OY.
b) Calcular el volumen del sólido engendrado al girar alrededor del eje OY la región limitada por la parábolas $y^2 = x$, $x^2 = y$.

Ejercicio 8.17. Sea $f(x) = x^5 + 4x^3 + 2x^2 + 8$. Calcula el volumen al girar dicha función alrededor del eje OX entre los valores donde f alcanza su máximo y su mínimo relativos.

Área de superficies de revolución

Ejercicio 8.18.
a) Calcula el área de una esfera de radio R.
b) Calcula el área de la figura obtenida al girar la parábola $y^2 = 4x$ y la recta $x = 5$ alrededor del eje OX.
c) Calcula el área lateral de un cono de radio R y altura h.

Ejercicio 8.19. Calcular:
a) La integral de $f(x) = \frac{1}{x^2}$ con $x \in [1, +\infty]$.
b) El volumen y la superficie lateral del sólido obtenido al girar la gráfica de la anterior función respecto del eje OX.
c) Ídem a los dos anteriores con $g(x) = \frac{1}{x}$ con $x \in [1, +\infty]$.

Series numéricas y series de Taylor

9

9.1 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.

`sum(expr,n,a,b)`	$\sum_{n=a}^{b} expr$
`sum(expr,n,a,b),simpsum`	simplifica la suma
`nusum (expr,x,a,plus)`	$\sum_{n=a}^{b} expr$

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**

```
(%i58)  sum(i,i,0,100);
(%o58)  5050
```

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

```
(%i59)  sum(i,i,0,n);
(%o59)  sum(i, i, 0, n)
(%i60)  sum(i,i,0,n),simpsum;
(%o60)  (n^2+n)/2
```

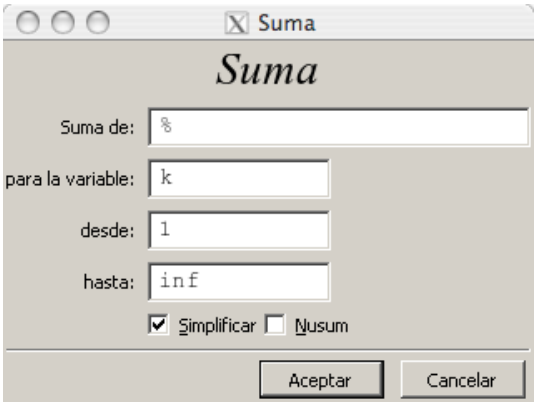


Figura 9.1 Ventana para calcular sumas

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**

```
(%i61)  nusum(i,i,0,n);
```

```
(%o61)   n(n+1)
         ------
           2
```

A cualquiera de estas dos órdenes se accede desde el menú **Análisis→Calcular suma**. Aparece la ventana de la Figura 9.1 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,

```
(%i62)   sum(1/n^2,n,1,inf),simpsum;
(%o62)   %pi^2
         ----
          6
```

o la serie armónica

```
(%i63)   sum(1/n,n,1,inf),simpsum;
(%o63)   ∞
```

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

```
(%i64)   sum(x^n,n,1,inf),simpsum;
         Is |x|-1 positive, negative or zero? negative;
(%o64)   x
         ---
         1-x
```

En cambio no suma algunas otras como

```
(%i65)   sum(1/n!,n,0,inf),simpsum;
(%o65)   inf
         ====
         \     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 $\sum_{n\geq 1} \frac{1}{n^n}$

```
(%i66)   a[n]:=1/n^n;
(%o66)   a    := 1
           n     n
                n
(%i67)   sum(a[n],n,1,inf),simpsum;
```

```
(%o67)  sum(1/n^n,n,1,inf)
```

$$\sum_{n=1}^{\infty} \frac{1}{n^n}$$

no sabemos sumarla, pero el criterio del cociente

```
(%i68)  limit(a[n+1]/a[n],n,inf);
(%o68)  0
```

nos dice que es convergente.

9.2 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?

```
(%i69)  taylor(cos(x),x,0,4);
(%o69)  1-x^2/2+x^4/24+...
```

¿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.

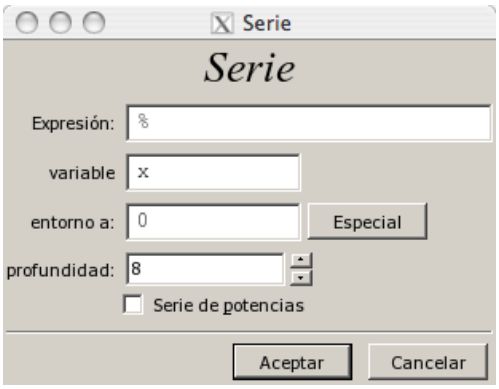


Figura 9.2 Ventana para el calculo de la serie de Taylor

`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

```
(%i70)  niceindices(powerseries(cos(x), x, 0));
(%o70)  sum((-1)^i*x^(2*i)/(2*i)!,i,0,inf)
```

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

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":

powerseries
niceindices

```
(%i71)  powerseries(cos(x),x,0);
```

```
(%o71)  sum((-1)^i16*x^(2*i16)/(2*i16)!,i16,0,inf)
```

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

```
(%i72)  niceindices(powerseries(atan(x),x,0));
(%o72)  sum((-1)^i*x^(2*i+1)/(2*i+1),i,0,inf)
```

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:

```
(%i73)  a[i]:=(-1)^i/(2*i+1);
(%o73)  a_i := (-1)^i/(2*i+1)
(%i74)  limit(abs(a[i+1]/a[i]),i,inf);
(%o74)  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 9.1. Se pueden realizar algunas operaciones con series de potencias. Por ejemplo, intenta derivarlas:

```
(%i75)  f(x):=niceindices(powerseries(log(1+x^2),x,0))$
(%i76)  diff(f(x),x);
(%o76)  -2*sum((-1)^i*x^(2*i-1),i,1,inf)
```

9.3 Ejercicios

Ejercicio 9.1. Estudiar la convergencia de las series que tienen el siguiente término general.

a) $a_n = \dfrac{n^2 - n + 3}{3^n + 2^{-n} + n - 1}$,

b) $a_n = \ln\left(\dfrac{n^3 + 1}{n^3 + 2n^2 - 3}\right)$,

c) $a_n = \dfrac{1}{2^n - n}$,

d) $a_n = \dfrac{n!}{n^n}$,

e) $a_n = \dfrac{1}{\ln(n)^2}$.

Ejercicio 9.2. ¿Cuántos términos de la serie

$$\sum_{n\geq 2} \frac{1}{n\log(n)}$$

hay que sumar para que la suma parcial sea mayor que 3? ¿Es convergente?

Ejercicio 9.3. 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 9.4. 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) $f(x) = \sin^2(x)$, $a = 0$,
c) $f(x) = \log(1 + x^2)$, $a = 0$,
d) $f(x) = \log(5 + x^2)$, $a = 0$,
e) $f(x) = \log(5 + x^2)$, $a = 1$,
f) $f(x) = x^4 + 3x^2 - 2x + 3$, $a = 1$,
g) $f(x) = |x|$, $a = 1$,
h) $f(x) = |x|$, $a = 5$.

Interpolación polinómica

10

10.1 Interpolación polinómica

Dados dos puntos del plano (x_1, y_1), (x_2, y_2), sabemos que hay una recta que pasa por ellos. Dicha recta es la gráfica de un polinomio de grado 1, en este caso la función es

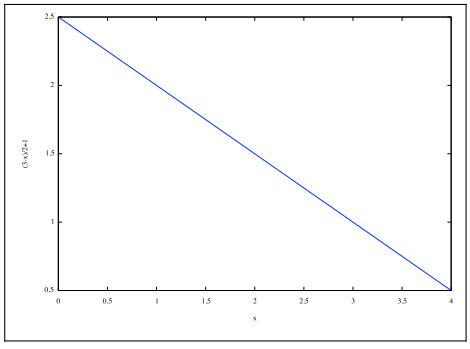
$$f(x) = f(x_2) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_2).$$

Por ejemplo, la recta que pasa por los puntos $(1, 2)$ y $(3, 1)$ es

```
(%i1)  recta(x1,y1,x2,y2):=y2+(x-x2)*(y2-y1)/(x2-x1)$
(%i2)  recta(1,2,3,1);
(%o2)  (3-x)/2+1
```

y la gráfica

```
(%i3)  wxplot2d(recta(1,2,3,1),[x,0,4]);
(%t3)
```



El problema general es cómo se busca una función que tome unos valores en unos puntos concretos. También se puede exigir que las derivadas de algún orden tengan un valor predeterminado.

10.2 Interpolación de Lagrange

El problema más clásico de interpolación es la *interpolación de Lagrange:*

Dados $n+1$ pares de puntos (x_0, y_0), (x_1, y_1),..., (x_n, y_n), encuéntrese el polinomio P de grado menor o igual que n tal que $P(x_i) = y_i$, $i = 0, 1, ..., n$.

Los puntos x_0, x_1,..., x_n se llaman *nodos de interpolación.*

10.2.1 Dos o tres nodos

Comencemos con un caso sencillo. Dada una lista de un par de puntos y un par de valores, ¿cuál es el polinomio que pasa por esos puntos?

```
(%i4)  nodos:[1,2];
(%o4)  [1,2]
(%i5)  valor:[3,7];
(%o5)  [3,7]
```

Necesitamos un polinomio de grado uno:

```
(%i6)  define(f(x),a*x+b);
(%o6)  f(x):=ax+b
```

que debe verificar que $f(1) = 3$, $f(2) = 7$. Con estas dos condiciones planteamos un sistema de ecuaciones que nos permite calcular a y b:

```
(%i7)  solve([f(nodos[1])=valor[1],f(nodos[2])=valor[2]],[a,b]);
(%o7)  [[a=4,b=-1]]
```

Podemos aplicar la misma técnica para encontrar el polinomio, de grado 2 en este caso, que pasa por los puntos $(1, 3)$, $(2, 7)$ y $(3, 1)$:

```
(%i8)  nodos:[1,2,3]$
(%i9)  valor:[3,7,1]$
(%i10)  define(f(x),a*x^2+b*x+c)$
(%i11)  solve([f(nodos[1])=valor[1],f(nodos[2])=valor[2],
          f(nodos[3])=valor[3]],[a,b,c]);
(%o11)  [[a=-5,b=19,c=-11]]
```

Vamos a resolver este mismo problema de otra forma. Busquemos tres polinomios de orden 2, `L1`, `L2` y `L3` verificando que valen 1 en uno de los nodos y cero en el resto. En concreto, buscamos `L1`,

L2 y L3 tales que L1(1)=1, L2(2)=1, L3(3)=1 y que valen cero en los otros puntos. Si encontramos estos polinomios, entonces la solución a nuestro problema es

```
3  L1+ 7  L2 + L3.
```

Comencemos con L1: el polinomio $(x-2)(x-3)$ se anula en 2 y en 3, pero su valor en 1, $(1-2)(1-3)$, no es 1. Si dividimos por dicha cantidad

$$\texttt{L1}(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)}$$

ya hemos encontrado el primer polinomio que estábamos buscando. Análogamente

$$\texttt{L2}(x) = \frac{(x-1)(x-3)}{(2-1)(2-3)}$$
$$\texttt{L3}(x) = \frac{(x-1)(x-2)}{(3-1)(3-2)}$$

Con estos tres polinomios ya podemos interpolar cualesquiera tres valores en 1, 2 y 3.

10.2.2 Caso general

Si ahora tenemos $n+1$ nodos distintos, podemos hacer una construcción análoga.

Definición 10.1. Dados x_i, $i = 0, 1, 2, ..., n$ números reales distintos, los polinomios

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(x-x_j)}{(x_i-x_j)}, \quad i = 0, 1, 2, \ldots, n$$

se llaman *polinomios de Lagrange* de grado n en los puntos $x_0, x_1, \ldots, x_n$.

Polinomio de Lagrange

¿Qué propiedades tienen? Fíjate que son muy fáciles de evaluar en los puntos x_i. Si evaluamos en el mismo índice

$$L_i(x_i) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(x_i-x_j)}{(x_i-x_j)} = 1,$$

ya que numerador y denominador coinciden. Si evaluamos en x_k con $k \neq i$, entonces

$$L_i(x_k) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{(x_k-x_j)}{(x_i-x_j)} = 0,$$

ya que uno de los factores del numerador, en concreto cuando $j = k$, se anula. Resumiendo, los polinomios de Lagrange valen

$$L_i(x_k) = \begin{cases} 1, & \text{si } i = k, \\ 0, & \text{si } i \neq k. \end{cases}$$

para $i, k = 0, 1, \ldots, n$. Observa que hemos dado el valor de los polinomios de Lagrange en $n+1$ puntos diferentes y, por tanto, estos valores los determinan completamente.

Con los polinomios L_i ahora podemos calcular fácilmente un polinomio que tome valores y_i en los nodos x_i. En efecto, el polinomio

$$P(x) = y_0L_0(x) + y_1L_1(x) + \cdots + y_nL_n(x)$$

cumple que $P(x_i) = y_i$ para $i = 0, 1,...,n$. El siguiente teorema recoge toda la información que hemos presentado.

Teorema 10.2. *Sean $x_0, x_1,..., x_n$ números reales distintos. Entonces dados $y_0, y_1,..., y_n$ existe un único polinomio $P_n(x)$ de grado menor o igual que n verificando que*

$$P_n(x_i) = y_i,\ i = 0, 1, \ldots, n.$$

Fórmula de Lagrange del polinomio de interpolación

Dicho polinomio viene dado por

$$P(x) = y_0L_0(x) + y_1L_1(x) + \cdots + y_nL_n(x), \tag{10.1}$$

donde

$$L_i(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{i-1})(x - x_{i+1})\cdots(x - x_n)}{(x_i - x_0)(x_0 - x_1)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_n)}$$

para $i = 0, 1,..., n$.

La identidad (10.1) se llama *fórmula de Lagrange del polinomio de interpolación.*

Ventajas e inconvenientes

Los polinomios de Lagrange son muy fáciles de calcular. Es por ello que se utilizan como uno de los primeros ejemplos de polinomios interpoladores. Su interés práctico es limitado y suelen presentarse más bien como ejemplo teórico de interpolación.

Su principal inconveniente se presenta cuando el conjunto de nodos es muy grande. En ese caso el grado del polinomio también es muy grande. Esto implica dificultades para el cálculo y, además, hay una alta tendencia a que el polinomio oscile mucho entre dos nodos.

10.2.3 El paquete *interpol*

Puedes pensar en alguna forma de calcular los polinomios de Lagrange en un conjunto de nodos, pero en *Maxima* disponemos del paquete *interpol* que calcula el polinomio interpolador de Lagrange. En primer lugar cargamos el módulo

```
(%i12)  load(interpol)$
```

y podemos usar la orden `lagrange` para calcular el polinomio que interpola una lista de pares (nodo, valor)

```
(%i13)  lagrange([[1,3],[2,1],[3,4]]);
(%o13)  2(x-2)(x-1)-(x-3)(x-1)-3(2-x)(x-3)/2
(%i14)  expand(%);
(%o14)  5x^2/2 - 19x/2+10
```

o, en el caso de que los nodos sea 1, 2, 3, 4, etc., simplemente dando la lista de valores

```
(%i15)  expand(lagrange([3,1,4]));
(%o15)  5x^2/2 - 19x/2+10
```

`lagrange([[nodo1,valor1],[nodo2,valor2],...])`	polinomio de Lagrange
`lagrange([valor1,valor2,...])`	polinomio de Lagrange

10.2.4 Ejercicios

Ejercicio 10.1. ¿Cuál es el error cuando aproximamos $\sqrt{102}$ utilizando el valor de la función raíz cuadrada en 81, 100 y 121? Representa las gráficas de la función raíz cuadrada y compárala con la gráfica del polinomio.

Ejercicio 10.2. Utiliza los valores de la función raíz cuadrada en $n = 2, 3, \ldots, 10$ puntos, elegidos por tí, para calcular el polinomio de interpolación y aproximar el valor en 102. Haz una animación que represente la función raíz cuadrada y el polinomio de interpolación de Lagrange en función de su grado.

Ejercicio 10.3. Calcula la fórmula de la suma de los cubos de los primeros n naturales sabiendo que es un polinomio de grado cuatro.

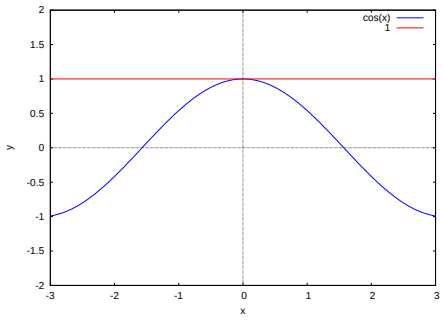
Ejercicio 10.4. Se considera la función $f(x) = e^{-x}\,\mathrm{sen}(2x)$ con $x \in [2, 5]$. Calcula el polinomio de interpolación de Lagrange de la función en los puntos 2, 3 y 4. Calcula los ceros de la función y de dicho polinomio en ese intervalo.

10.3 Polinomio de Taylor

En el Capítulo 7 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

```
(%i16)  f(x):=cos(x);
(%o16)  f(x):=cos(x)
(%i17)  t(x):=1;
(%o17)  t(x):=1
(%i18)  plot2d([f(x),t(x)],[x,-3,3],[y,-2,2]);
```

```
(%o18)
```



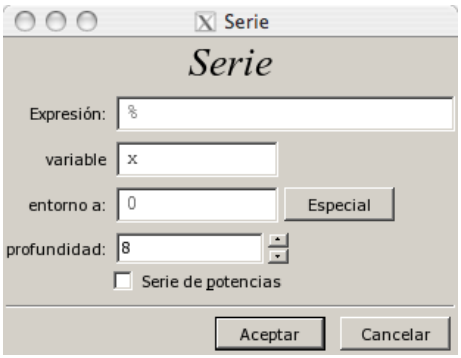


Figura 10.1 Ventana para el cálculo del polinomio de Taylor

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:

$$\begin{aligned} T(f,a,n)(x) &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 \\ &\quad + \frac{f'''(a)}{3!}(x-a)^3 + \cdots + \frac{f^{n)}(a)}{n!}(x-a)^n \\ &= \sum_{k=1}^{n} \frac{f^{k)}(a)}{k!}(x-a)^k \end{aligned}$$

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. Como en otras ventanas similares, si marcamos la casilla de `Especial`, podemos elegir π o e como centro para el cálculo del desarrollo.

`taylor(f(x),x,a,n)`	polinomio de Taylor de la función f en el punto a y de orden n
`trunc(polinomio)`	convierte polinomio de Taylor en un polinomio
`taylorp(polinomio)`	devuelve `true` si el polinomio es un polinomio de Taylor

Veamos un ejemplo.

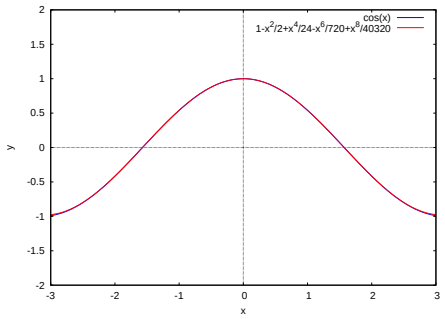
```
(%i19)  taylor(cos(x),x,0,5);
```

```
(%o19)
```
$$1-\frac{x^2}{2}+\frac{x^4}{24}+\ldots$$

```
(%i20) taylor(log(x),x,1,7);
```
$$(\%o20)\quad x-1-\frac{(x-1)^2}{2}+\frac{(x-1)^3}{3}-\frac{(x-1)^4}{4}+\frac{(x-1)^5}{5}-\frac{(x-1)^6}{6}+\frac{(x-1)^7}{7}+\ldots$$

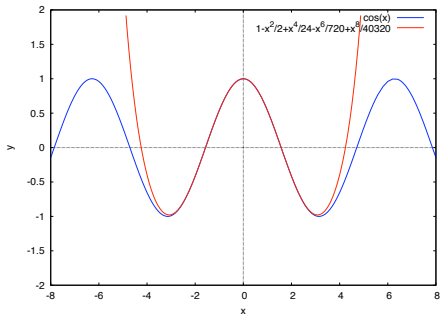
En teoría, un polinomio de Taylor de orden más alto debería aproximar mejor a la función. Ya hemos visto cómo aproxima la recta tangente a la función coseno. Vamos ahora a dibujar las gráficas de la función $f(x) = \cos(x)$ y de su polinomio de Taylor de orden 8 en el cero para comprobar que la aproximación es más exacta.

```
(%i21) plot2d([f(x),taylor(f(x),x,0,8)],[x,-4,4],[y,-2,2]);
(%o21)
```



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

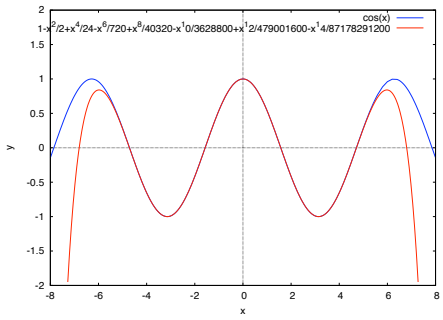
```
(%i22) plot2d([f(x),taylor(f(x),x,0,8)],[x,-8,8],[y,-2,2]);
(%o22)
```



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:

```
(%i23) plot2d([f(x),taylor(f(x),x,0,14)],[x,-8,8],[y,-2,2]);
```

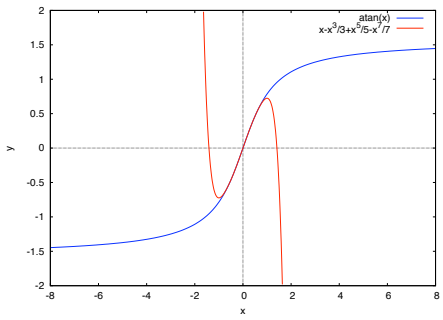
(%o23)



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:

```
(%i24)  g(x):=atan(x);
(%o24)  g(x):=atan(x)
(%i25)  plot2d([g(x),taylor(g(x),x,0,8)],[x,-8,8],[y,-2,2]);
```

(%o25)



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

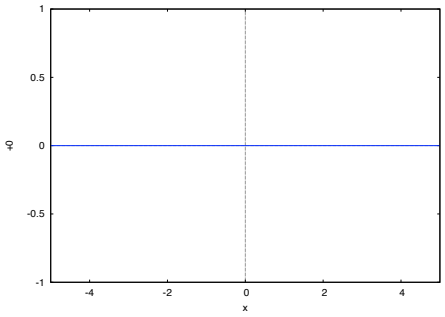
Observación 10.3. *Maxima* tiene dos formas de representar internamente los polinomios. Sin entrar en detalles, no se guardan de la misma forma un polinomio de Taylor y un polinomio cualquiera. Esto puede dar lugar a algunas sorpresas. Por ejemplo, hemos visto cómo el polinomio de Taylor nos sirve para aproximar una función, pero, en lugar de representar la función y dicho polinomio, podríamos representar la diferencia. Veamos que ocurre[9]. Definimos las funciones,

```
(%i26)  f(x):=cos(x);
(%o26)  f(x):=cos(x)
(%i27)  define(g(x),taylor(f(x),x,0,5));`
(%o27)  g(x):=1-x^2/2+x^4/24+...`
```

y dibujamos la diferencia

[9] Este ejemplo está hecho con la versión 5.18 de *Maxima*. Es posible que el resultado sea distinto en otras versiones.

```
(%i28)  plot2d(f(x)-g(x),[x,-5,5]);
(%o28)
```



¿Cómo puede salir 0? ¿Es que no hay diferencia? Sí la hay. Ya lo sabemos: si evaluamos en algún punto podemos ver que el resultado no es cero.

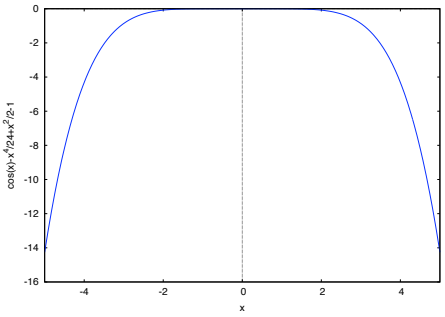
```
(%i29)  f(2)-g(2);
(%o29)  (3*cos(2)+1)/3
```

Como hemos avisado antes, *Maxima* maneja de forma diferente un polinomio de Taylor y un polinomio "normal". Puedes comprobarlo preguntando a *Maxima* si $g(x)$ es un polinomio de Taylor o no.

```
(%i30)  taylorp(g(x));
(%o30)  true
(%i31)  taylorp(1+x^2);
(%o31)  false
```

La orden `trunc(`*`polinomio de Taylor`*`)` nos permite pasar de polinomio de Taylor a polinomio "normal". Con estos ya no tenemos este problema. **trunc**

```
(%i32)  plot2d(f(x)-trunc(g(x)),[x,-5,5]);
(%o32)
```



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:

```
(%i33)  f(x):=sin(x)+cos(x);
(%o33)  f(x):=sin(x)+cos(x)
(%i34)  tay(n,x):=block([ts:taylor(f(z),z,0,n)],subst(z=x,ts));
(%o34)  tay(n,x):=block([ts:taylor(f(z),z,0,n)],subst(z=x,ts))
(%i35)  with_slider(n,makelist(i,i,1,20),
          [f(x),'(tay(n,x))],[x,-10,10],[y,-3,3]);
```

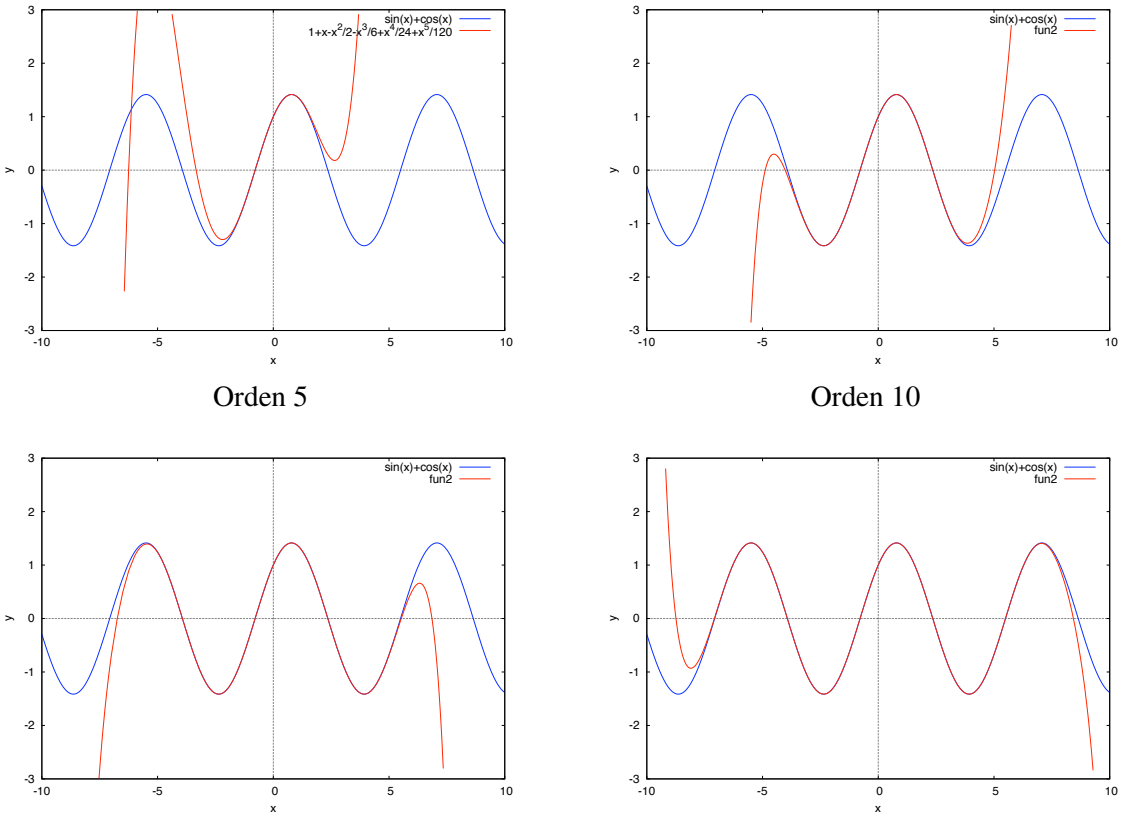
nos permite dibujar los primeros 20 polinomios de Taylor de la función f. En la Figura 10.2 tienes algunos pasos intermedios representados.

Observación 10.4. 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.

10.3.1 Ejercicios

Ejercicio 10.5. ¿Es cierto o falso que el polinomio de Taylor de una función al cuadrado es el cuadrado del polinomio?

Ejercicio 10.6. Estudia los extremos relativos del polinomio de orden 5 centrado en el origen de la función $f(x) = \cos(x) + e^x$.



Orden 5 Orden 10

Orden 15 Orden 20

Figura 10.2 Función $\text{sen}(x) + \cos(x)$ y sus polinomios de Taylor

Derivación e integración numérica

11

El cálculo de la derivada de o de la integral de una función no siempre es fácil. En este capítulo vamos a ver cómo podemos aproximarlos. Una de las herramientas claves es la interpolación. Para polinomios sí es factible calcular derivadas e integrales. Aprovecharemos tanto la interpolación de Lagrange como la de Taylor para aproximar una función por un polinomio.

11.1 Derivación numérica

A veces ocurre que calcular la derivada de una función f en un punto a del interior de dominio no es fácil, ya sea por la complejidad de la función dada, ya sea porque sólo dispongamos de una tabla de valores de f. En esta sección vamos a establecer métodos para calcular $f'(a)$. No es que vayamos a calcular la función derivada primera de f, sino que vamos a aproximar los valores de ésta en un punto dado a.

Hemos visto en este capítulo que la derivada de una función $f : I \to \mathbb{R}$ en un punto $a \in I$ es

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a} = \lim_{h \to 0} \frac{(a + h) - f(a)}{h} .$$

Si consideramos un valor de h suficientemente pequeño, podemos dar una primera aproximación

$$f'(a) \approx \frac{f(a + h) - f(a)}{h} .$$

11.1.1 Fórmulas de derivación numérica

En las fórmulas que vamos a estudiar en este apartado aparecen dos valores: el que aproxima $f'(a)$ y el error cometido. Aunque este último no se calcula explícitamente, sí se puede dar una acotación del mismo. Notemos que dicho error se obtiene gracias al desarrollo de Taylor de f centrado en a.

En lo que sigue, el parámetro h se suele tomar positivo y "pequeño".

a) Fórmula de dos puntos:

$$f'(a) = \frac{1}{h} \left(f(a) - f(a - h)\right) + \frac{h}{2} f''(\psi) , \quad \psi \in]a - h, a[$$

Esta fórmula se llama *regresiva* porque utiliza información de f en a y en $a - h$.

$$f'(a) = \frac{1}{h} \left(f(a + h) - f(a)\right) - \frac{h}{2} f''(\psi) , \quad \psi \in]a, a + h[$$

Esta fórmula se llama *progresiva* porque utiliza información de f en a y en $a + h$.

El primer sumando de ambas fórmulas nos da una aproximación de $f'(a)$, y el segundo nos indica el error cometido.

b) Fórmula de tres puntos:

$$f'(a) = \frac{1}{2h}\left(f(a+h) - f(a-h)\right) - \frac{h^2}{6} f'''(\psi)\,, \quad \psi \in]a-h, a+h[$$

Esta fórmula se llama *central* porque utiliza información de f en $a-h$ y en $a+h$.

$$f'(a) = \frac{1}{2h}\left(-3f(a) + 4f(a+h) - f(a+2h)\right) + \frac{h^2}{3} f'''(\psi)\,, \quad \psi \in]a, a+2h[$$

Esta fórmula es progresiva.

c) Fómula de cinco puntos:

$$f'(a) = \frac{1}{12h}\left(f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)\right) - \frac{h^4}{30} f^{(5)}(\psi)\,, \quad \psi \in]a-2h, a+2h[$$

Ésta es central.

$$f'(a) = \frac{1}{12h}\left(-25f(a) + 48f(a+h) - 36f(a+2h) + 16f(a+3h) - 3f(a+4h)\right) + \frac{h^4}{5} f^{(5)}(\psi)\,, \quad \psi \in]a, a+4h[$$

Esta es progresiva.

Unas observaciones sobre el término del error:

a) Cuanto mayor es el exponente de h en la fórmula del error, mejor es la aproximación.

b) Cuanto menor es la constante que aparece en la fórmula del error, mejor es la aproximación.

c) Cuidado con los errores de redondeo cuando trabajemos con h excesivamente pequeño. Puede ocurrir que por la aritmética del ordenador, la aproximación numérica sea peor cuanto más pequño sea h.

Ejemplo 11.1. FALTA (PARA PRÁCTICAS)

11.2 Integración numérica

11.2.1 Introducción

El cálculo de la integral de una función en un intervalo $[a, b]$ a través de la regla de Barrow puede ser en algunos casos no sólo complicado, sino prácticamente imposible por varias razones, principalmente dos.

La primera razón es que nos sea imposible calcular una primitiva elemental de la función a integrar, este caso es muy usual. Por ejemplo la función$f(x) = (1 + x^2)^{1/3}$ no somos capaces de calcularle una primitiva expresable con funciones elementales. Otro ejemplo, muy usual éste, es la función de densidad $f(x) = e^{-x^2}$ que tantas veces aparece.

La segunda razón para no poder aplicar la regla de Barrow es que no conozcamos la expresión de la función que queremos integrar o simplemente que no conozcamos el valor de la función en

todos los puntos del intervalo donde está definida (le llamaremos $[a, b]$) sino solamente en algunos puntos de dicho intervalo

Por todo lo anterior, a veces hay que recurrir a métodos de aproximación para calcular el valor de la integral de una función en un intervalo. Ya que la propia definición de la integral viene dada por un límite (de sumas superiores y sumas inferiores), una aproximación a dicho límite podría considerarse un primer método de aproximación numérica de la integral.

Los métodos de aproximación del cálculo de la integral que veremos se llaman *métodos o reglas de integración numérica* y consisten, básicamente, en dada $f : [a, b] \to \mathbb{R}$ continua, aproximamos dicha función por un polinomio adecuado. Así, la integral de dicho polinomio será una aproximación de la integral de f. Estos métodos son los llamados *métodos simples*, que, si bien son muy sencillos de enunciar, la aproximación que dan deja mucho que desear en muchos casos. Un análisis posterior nos dará los *métodos compuestos* que producen una aproximación más buena. Primero veremos los simples.

11.3 Métodos simples

11.3.1 Método del trapecio simple

El método del trapecio simple consiste en considerar como polinomio de aproximación de la función f en el intervalo $[a, b]$ el polinomio de interpolación de Lagrange de la función f en $x_0 = a$ y $x_1 = b$ y se toma $\int_a^b P_1(x)dx$ como aproximación de $\int_a^b f(x)dx$. El polinomio de interpolación de Lagrange es

$$P_1(x) = f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a},$$

lo que nos proporciona

$$\int_a^b f(x)dx \approx \int_a^b P_1(x)dx = \frac{b-a}{2}(f(a) + f(b)).$$

El nombre de método o regla del trapecio está claro si observamos en el siguiente dibujo que aproximación estamos dando de la integral: Si tenemos una función $f : [a, b] \to \mathbb{R}$ y suponemos que es positiva, la aproximación dada por el método del trapecio es el área del trapecio formado por los puntos $(a, 0)$, $(b, 0)$, $(b, f(b))$ y $(a, f(a))$.

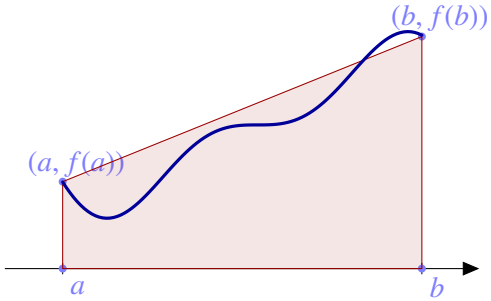


Figura 11.1 Método del trapecio simple

Esta aproximación solo es exacta cuando la función f es un polinomio de grado 1. En cualquier caso tenemos que el error que se comete al aplicar esta regla viene dado por

$$\int_a^b f(x)dx - \frac{b-a}{2}\left(f(a)+f(b)\right) = -\frac{1}{12}(b-a)^3 f''(\xi)$$

para conveniente $\xi \in]a,b[$.

11.3.2 Regla del punto medio simple

Para construir esta aproximación de la integral de la función f en el intervalo $[a,b]$ se considera P_0 el polinomio de aproximación de Lagrange de la función en el punto $x_0 = \frac{a+b}{2}$ y se toma $\int_a^b P_0(x)dx$ como aproximación de $\int_a^b f(x)dx$. Como $P_0(x) = f(\frac{a+b}{2})$, entonces la aproximación queda

$$\int_a^b f(x)dx \approx \int_a^b P_0(x)dx = (b-a)f(\frac{a+b}{2})$$

que, si f es positiva en el intervalo $[a,b]$, nos da como aproximación del área que queda entre el eje de abscisas y la gráfica de f entre a y b, el área del rectángulo de base $b-a$ (la longitud de intervalo $[a,b]$) y altura $f(\frac{a+b}{2})$.

Claramente esta fórmula es exacta cuando la función f es constante en el intervalo $[a,b]$, y, en general, el error cometido al tomar esta aproximación es

$$\int_a^b f(x)dx - (b-a)f(\frac{a+b}{2}) = \frac{1}{24}(b-a)^3 f''(\xi),$$

para algún $\xi \in]a,b[$.

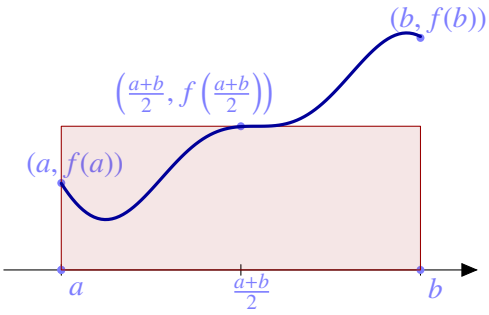


Figura 11.2 Método del punto medio simple

11.3.3 Regla de Simpson simple

En este caso se toma como aproximación de la función P_2, el polinomio de interpolación de Lagrange de la función f en los puntos $x_0 = a$, $x_1 = \frac{a+b}{2}$ y $x_2 = b$. Como aproximación de $\int_a^b f(x)dx$ se toma $\int_a^b P_2(x)dx$. Así el método de Simpson simple nos da

$$\int_a^b f(x)dx \approx \frac{(b-a)}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

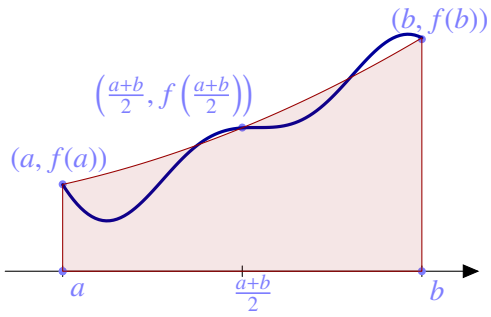


Figura 11.3 Regla de Simpson simple

Esté método es exacto cuando la función f es un polinomio de grado menor o igual que 3. El error cometido al sustituir el valor exacto de la integral por la aproximación que nos da la regla de Simpson vale

$$\int_a^b f(x)dx - \frac{(b-a)}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) = -\frac{1}{90\cdot 2^6}(b-a)^5 f^{4)}(\xi),$$

para algún $\xi \in]a,b[$.

Está claro que, al ser P_2 una mejor aproximación, en general, que P_0 y P_1, la regla de Simpson es mejor aproximación a la integral que la regla del trapecio y la del punto medio. La siguiente tabla muestra las aproximaciones dadas por las tres reglas que hemos presentado.

ahora viene una tabla que tengo que hacer.

En vista de las fórmulas de los errores cometidos al tomar como aproximación cualquiera de las fórmulas anteriores, estos errores dependen mucho de la longitud del intervalo $[a,b]$. En todos las fórmulas de los errores aparece un término $(b-a)$ elevado a una potencia. Está claro que si $(b-a)$ es muy grande la aproximación no es muy buena. Este hecho además responde a la intuición: no podemos pretender que si el intervalo de definición de la función es muy grande al tomarnos como aproximación de la función un polinomio de grado 0 (e incluso 1 o 2) obtengamos una muy buena aproximación.

Si en el ejemplo de la tabla anterior aumentamos la longitud del intervalo veamos qué ocurre:

Otra tabla con el mismo ejemplo con un intervalo más grande y comentar los errores que se cometen ahora.

11.4 Métodos de aproximación compuestos

Una forma de resolver este problema consiste en dividir el intervalo $[a,b]$ en n subintervalos de idéntica longitud y aplicar un método de aproximación simple en cada uno de los subintervalos obtenidos y después sumar las aproximaciones obtenidas. La aditividad de la integral respecto al intervalo nos garantiza que la suma obtenida es una aproximación de la integral de la función en el intervalo $[a,b]$. Estos métodos de aproximación se llaman métodos de aproximación compuesta y vamos a comentar brevemente los tres correspondientes a los métodos de aproximación simple que acabamos de estudiar.

Lo primero que tenemos que hacer, en los tres métodos, es dividir el intervalo en n subintervalos de igual longitud. Para tal fin consideramos los puntos

$$x_k = a + k\frac{b-a}{n}, \quad k = 0, 1, \ldots n, \tag{11.1}$$

donde n es un número natural. Estos puntos x_k, para $k = 0, 1, \ldots, n$ reciben el nombre de nodos. Si ahora aplicamos cada uno de los métodos de aproximación simples a la función en cada uno de los intervalos $[x_k, x_{k+1}]$ obtenemos en cada caso las siguientes fórmulas:

Método del trapecio compuesto

Sean $x_0, x_1, \ldots, x_n$ los nodos dados por (11.1). La aproximación de la integral $\int_a^b f(x)dx$ utilizando el método del trapecio compuesto con los subintervalos dados por los anteriores nodos nos proporciona

$$T_n = \frac{(b-a)}{2n}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right),$$

y el error cometido queda estimado por la fórmula

$$\int_a^b f(x)dx - T_n = -\frac{1}{12}\frac{(b-a)^3}{n^2}f''(\xi)$$

donde ξ es un número del intervalo $]a, b[$.

Método del punto medio compuesto

Considerando los mismos nodos que en el apartado anterior la regla del punto medio compuesto nos da una aproximación

$$M_n = \frac{(b-a)}{n}\sum_{i=0}^{n-1} f\left(a + \frac{(2i+1)}{2}\frac{(b-a)}{n}\right).$$

En este caso la expresión del error nos queda

$$\int_a^b f(x)dx - M_n = \frac{(b-a)^3}{24n^2}f''(\xi)$$

para $\xi \in]a, b[$.

Método de Simpson compuesto

En este caso necesitamos que el número n sea par. Usando el método de Simpson en cada intervalo $[x_k, x_{k+1}]$ obtenemos la aproximación

$$\begin{aligned} S_n = \frac{(b-a)}{3n}\big[&(f(x_0) + 4f(x_1) + f(x_2)) + (f(x_2) + 4f(x_3) + f(x_4)) + \cdots \\ &\cdots + (f(x_{n-2}) + 4f(x_{n-1}) + f(x_n))\big]. \end{aligned}$$

Haciendo un fácil cambio de variable obtenemos que la anterior expresión coincide con

$$S_n = \frac{(b-a)}{3n}\sum_{i=1}^{n/2}\left[f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})\right]$$

y el error que se comete viene dado por

$$\int_a^b f(x)dx - S_n = -\frac{1}{180}\frac{(b-a)^5}{n^4} f^{4)}(\xi)$$

para un $\xi \in]a, b[$.

El error en los tres métodos depende inversamente del número de nodos n. Pero mientras que en el método del trapecio y en del punto medio el error es inversamente proporcional a n^2 en el método de Simpson es inversamente proporcional a n^4. Está claro por tanto que el método de Simpson es más exacto que los otros dos cuando aumentamos el número de nodos.

Ejemplo 11.2. Consideremos la integral de la función $f(x) = e^x \operatorname{sen}(x)$ en el intervalo $[0, 1]$. A esta función se le puede calcular una primitiva fácilmente integrando por partes (¡hágase!) y evaluando en 0 y en 1 queda

$$\int_0^1 e^x \operatorname{sen}(x)dx = 0.90933067363148.$$

Trapecio	Aproximación	Error
n=20	0.9670887643	5.78×10^{-2}
n=40	0.9380661609	2.87×10^{-2}
n=100	0.9207904171	1.15×10^{-2}
Punto medio	Aproximación	Error
n=20	0.9090435575	2.87×10^{-4}
n=40	0.9092588997	7.18×10^{-5}
n=100	0.9093191900	1.15×10^{-5}
Simpson	Aproximación	Error
n=20	0.9093305474	1.26×10^{-7}
n=40	0.9093306657	7.89×10^{-9}
n=100	0.9093306734	2.02×10^{-10}

Tabla 11.1 Fórmulas de integración compuesta

Números complejos

A

Después de calcular la raíz cuadrada de 2, la primera idea que se nos ocurre a todos es probar con números negativos "a ver que pasa":

```
(%i1)   sqrt(-2);
(%o1)   √2 %i
```

Correcto. Ya habíamos comentado con anterioridad que `%i` representa a la unidad imaginaria. En *Maxima* podemos trabajar sin problemas con números complejos. Se pueden sumar, restar, multiplicar, dividir

```
(%i2)   ((2+3*%i)+(3-%i);
(%o2)   2 %i+5
```

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

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);
                3 %i+2
(%o3)   -------------
                1-%i
(%i4)   rectform((2+3*%i)/(1-%i));
```

```
(%o4)   5 %i/2 - 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 atan(3))
```

No hace falta calcular la forma polar para conocer el argumento principal de un número complejo, `carg` se encarga de de ello:

carg

```
(%i8)   carg(1+3*%i);
(%o8)   atan(3)
(%i9)   carg(exp(%i));
(%o9)   1
```

Recuerda que muchas funciones reales tienen una extensión al plano complejo. Por ejemplo, `exp` nos da la exponencial compleja,

```
(%i10)   exp(%i*%pi/4);
(%o10)   sqrt(2)%i/2 + sqrt(2)/2
```

`log` nos da el logaritmo principal

```
(%i11)   log(%i);
(%o11)   log(%i)
(%i12)   log(-3);
(%o12)   log(-3)
```

siempre que se lo pidamos

```
(%i13)  rectform(log(%i));
(%o13)  %i π/2
(%i14)  rectform(log(-3));
(%o14)  log(3)+%i π
```

Podemos calcular senos o cosenos,

```
(%i15)  cos(1+%i);
(%o15)  cos(%i+1)
(%i16)  rectform(%);
(%o16)  cos(1)cosh(1)-%i sin(1)sinh(1)
```

si preferimos la notación exponencial, `exponentialize` escribe todo en términos de exponenciales

```
(%i17)  exponentialize(%);
(%o17)  (%e+%e^-1)(%e^%i+%e^-%i)/4 - (%e-%e^-1)(%e^%i-%e^-%i)/4
```

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

```
(%i18)  demoivre(%);
(%o18)  (%e+%e^-1) cos(1)/2 - (%e-%e^-1) i sin(1)/2
```

Observación A.1. La orden `abs` calcula el módulo de la expresión pedida como ya hemos comentado

```
(%i19)  abs(1+%i);
(%o19)  √2
```

pero en algunas ocasiones el resultado no es el esperado

```
(%i20)  abs(log(-3));
(%o20)  -log(-3)
```

En este caso, `cabs` nos da el resultado correcto **cabs**

```
(%i21)  cabs(log(-3));
(%o21)  sqrt(log(3)^2+%pi^2)
```

Avisos y mensajes de error

B

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 $\text{sen}(x) = 0$ tiene que usar la función arcoseno (su inversa) y *Maxima* avisa de que es posible que falten soluciones y, de hecho, faltan.

```
(%i9)   integrate(1/x,x,0,1);
        Is x + 1 positive, negative, or zero? positive;
        Integral is divergent
        - an error. To debug this try debugmode(true);
```

La función $\frac{1}{x}$ no es integrable en el intervalo $[0, 1]$.

```
(%i10)   find_root(x^2,-3,3);
         function has same sign at endpoints
         [f(-3.0)=9.0,f(3.0)=9.0]
         - an error. To debug this try debugmode(true);
```

Aunque x^2 se anula entre 3 y -3 (en $x = 0$ obviamente) la orden `find_root` necesita dos puntos en los que la función cambie de signo.

```
(%i11)   A:matrix([1,2,3],[2,1,4],[2.1,4]);
         All matrix rows are not of the same length.
         - an error. To debug this try debugmode(true);
```

En una matriz, todas las filas deben tener el mismo número de columnas: hemos escrito un punto en lugar de una coma en la última fila.

```
(%i12)   A:matrix([1,2,3],[2,1,4],[2,1,4])$
(%i13)   B:matrix([1,2],[2,1],[3,1])$
(%i14)   B.A
         incompatible dimensions - cannot multiply
         - an error. To debug this try debugmode(true);
```

No se pueden multiplicar estas matrices. Repasa sus órdenes.

Bibliografía

C

a) La primera fuente de documentación sobre *Maxima* es el propio programa. La ayuda es muy completa y detallada.

b) En la página del programa *Maxima*, http://maxima.sourceforge.net/es/, existe una sección dedicada a documentación y enlaces a documentación. El manual de referencia de *Maxima* es una fuente inagotable de sorpresas. Para cualquier otra duda, las listas de correo contienen mucha información y, por supuesto, siempre se puede pedir ayuda.

c) "*Primeros pasos en Maxima*" de Mario Rodríguez Riotorto es, junto con la siguiente referencia, la base de estas notas. Se puede encontrar en
http://www.telefonica.net/web2/biomates/

d) "*Maxima con wxMaxima: software libre en el aula de matemáticas*" de Rafael Rodríguez Galván es el motivo de que hayamos usado *wxMaxima* y no cualquier otro entorno sobre *Maxima*. Este es un proyecto que se encuentra alojado en el repositorio de software libre de RedIris
https://forja.rediris.es/projects/guia-wxmaxima/

e) Edwin L. Woollett está publicando en su página, capítulo a capítulo unas notas (más bien un libro) sobre *Maxima* y su uso. El título lo dice todo: "*Maxima by example*". Su página es
http://www.csulb.edu/ woollett/

f) Esta bibliografía no estaría completa sin mencionar que la parte más importante de esto sigue siendo la asignatura de Cálculo. La bibliografía de ésta ya la conoces.

Glosario