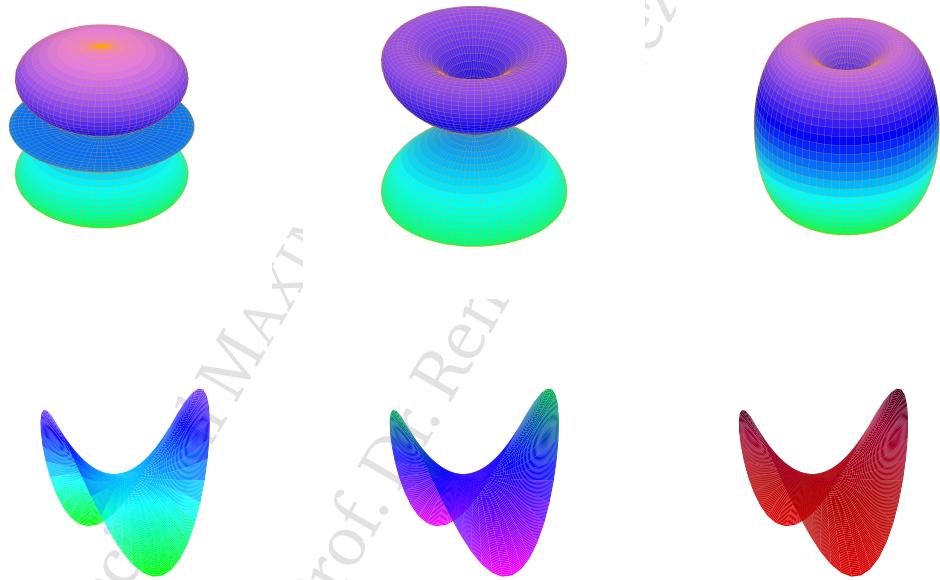


Introducción al Maxima CAS con algunas aplicaciones^{1b}

Renato Álvarez Nodarse

Dpto. Análisis Matemático, Universidad de Sevilla



<https://renato.ryn-fismat.es>

¹Versión del 14 de julio de 2024

A Niurka, Renato A. y Sofía

Introducción al MAXIMA CAS con algunas aplicaciones
Prof. Dr. Renato Álvarez Nodarse

Índice general

1. ¿Por qué usar el ordenador en las clases de matemáticas?	1
2. Primeros pasos con MAXIMA.	5
2.1. Empezando: primeras operaciones	5
2.2. Listas	14
2.3. Cálculo diferencial e integral con MAXIMA.	18
2.3.1. Trabajando con funciones elementales con MAXIMA.	18
2.3.2. Cálculo diferencial e integral de funciones de una variable.	21
2.3.3. Definiendo funciones con MAXIMA.	25
2.3.4. Aproximando funciones.	28
2.3.5. Definiendo funciones a trozos.	31
2.3.6. Cálculo diferencial de funciones de varias variables.	33
2.3.7. Cálculo vectorial.	34
3. Representaciones gráficas con MAXIMA.	39
3.1. Las órdenes plot2d y plot3d.	39
3.1.1. La orden plot2d y similares para gráficas en 2D.	39
3.1.2. Las órdenes plot3d y contour_plot para gráficas en 3D.	42
3.1.3. Exportando gráficas en pdf, eps, etc.	45
3.2. El paquete draw para representaciones gráficas.	46
3.2.1. Usando draw3d para gráficas 3D.	47
3.2.2. Usando draw2d para gráficas 2D.	51
3.2.3. Creando gráficas animadas.	57
3.3. MAXIMA, GNUPLOT y LATEX	61
4. Más comandos útiles de MAXIMA.	63
4.1. Resolviendo ecuaciones y manipulando expresiones algebraicas.	63
4.1.1. Resolviendo ecuaciones.	63
4.1.2. Resolviendo sistemas de ecuaciones.	65
4.1.3. Un poco más sobre la simplificación de expresiones.	67

4.1.4. Trabajando con desigualdades.	71
4.2. Trabajando con matrices.	72
4.3. Tratamiento de datos.	83
4.4. El problema de interpolación.	90
4.4.1. El polinomio de interpolación de Lagrange.	91
4.4.2. El polinomio de Lagrange con nodos de Chebyshev.	95
4.4.3. Splines.	96
4.5. Las funciones anónimas y su uso en MAXIMA.	100
5. Resolviendo EDOs con MAXIMA.	105
5.1. Soluciones analíticas.	105
5.2. Soluciones numéricas.	109
5.2.1. Exportando ficheros de datos.	112
5.3. Implementando un método numérico con MAXIMA.	113
6. Resolviendo sistemas de EDOs lineales.	123
6.1. Sistemas lineales de EDOs	123
6.2. Resolviendo sistemas con MAXIMA.	125
6.3. Un ejemplo de un sistema no lineal: el sistema Lotka-Volterra.	129
6.3.1. Representando campos de direcciones para EDOs.	131
6.4. Resolviendo EDOs lineales de orden n	134
6.4.1. Las funciones especiales de la física-matemática.	137
7. Resolución de EDOs mediante series de potencias.	145
7.1. Un ejemplo ilustrativo.	145
7.2. Automatizando el método de las series de potencias.	148
8. Otros ejemplos y ejercicios para el lector.	151
8.1. Dos ejemplos simples del álgebra lineal.	151
8.1.1. Ejemplo 1	151
8.1.2. Ejemplo 2	152
8.2. Fórmulas de cuadratura para funciones explícitas.	154
8.2.1. Fórmula de los rectángulos.	155
8.2.2. Fórmula de los trapecios.	156
8.2.3. Método de Simpson.	157
8.2.4. Ejercicios: Comparación de los métodos de cuadratura.	159
8.3. Fórmulas de cuadratura para una función definida por una lista.	160
8.4. Calculando los extremos de una función definida por una lista.	162
8.4.1. Definiendo el problema.	162

8.4.2. Ejemplo de aplicación.	165
8.4.3. Ejemplo de extremos e integración numérica.	167
8.5. Algunos modelos simples descritos por EDOs.	172
8.6. Modelos sencillos de dinámica de poblaciones.	175
8.6.1. El modelo malthusiano.	176
8.6.2. El modelo logístico de Verhulst.	180
8.6.3. Un modelo de poblaciones fluctuantes.	183
8.6.4. El modelo depredador-presa de Lotka-Volterra.	186
8.7. Encontrando los ceros de una función definida por una lista.	187
8.7.1. Ejemplo interesante: El tiro parabólico con resistencia del aire.	191
8.8. Un ejemplo de cambio variables para funciones de dos variables.	197
8.9. Calculando diferenciales de funciones escalares de varias variables.	200
8.10. Ajuste de mínimos cuadrados con pesos.	205
8.11. Un reto para el lector: la interpolación de Hermite.	209
Bibliografía adicional	211

Prefacio

Estas notas son una introducción al sistema de álgebra computacional MAXIMA. La motivación inicial para prepararlas era disponer de un breve manual para usar MAXIMA como complemento al curso de *Ampliación de Análisis Matemático* de la *Diplomatura de Estadística de la Universidad de Sevilla*. Dicha asignatura tenía como objetivo, esencialmente, el estudio y resolución numérica de ecuaciones diferenciales ordinarias. Más adelante se ampliaron para usarlas como complemento a un curso introductorio de Análisis Matemático hasta llegar a la presente versión donde se incluye mucho más material del usado en las asignaturas antes mencionadas. La filosofía general se convirtió en disponer de un texto para aprender MAXIMA a la vez que se repasaban algunos temas de matemáticas en general.

Así pues, el objetivo final de estas notas es mostrar como usar MAXIMA en la resolución de distintos problemas relacionados con las Matemáticas en general y con el Análisis matemático en particular —concretamente del Cálculo diferencial e integral y sus aplicaciones—. Haremos un énfasis especial en la resolución de ecuaciones diferenciales ordinarias (EDOs) tanto desde el punto de vista simbólico como numérico y discutiremos algunos problemas relacionados con las mismas. La elección de considerar ecuaciones diferenciales ordinarias no es arbitraria pues para su resolución se precisan muchos de los conceptos del Cálculo diferencial e integral y del Álgebra lineal y por tanto tendremos que mostrar como trabajar con ellos con MAXIMA. También presentaremos algunas aplicaciones a modelos *realistas* que nos obligarán a utilizar paquetes de tratamiento de datos, entre otros. En estas notas además se ha ampliado notablemente el apartado dedicado a la representación de funciones para mostrar al menos una pequeña parte del potencial de MAXIMA junto al de GNUPLOT.

Estas notas fueron ampliadas y corregidas en el año sabático que disfruté durante el curso 2016/2017 por lo que agradezco al Departamento de Análisis Matemático de la Universidad de Sevilla por su apoyo. En los tres cursos posteriores al susodicho curso se han corregido y ampliado y agregado algunos apartados como la §8.7.

Renato Álvarez Nodarse
Sevilla 10 de abril de 2020

Capítulo 1

¿Por qué usar el ordenador en las clases de matemáticas?

Los computadores, como instrumentos de experimentación matemática, han seguido un lento camino desde que John von Neumann los concibiera; Fermi, Pasta, Ulam y Tsingou-Menzel lo usaran por primera vez *experimentalmente* en los años 50, hasta el día de hoy, en que constituyen un elemento más de nuestro entorno cotidiano.¹ El uso de

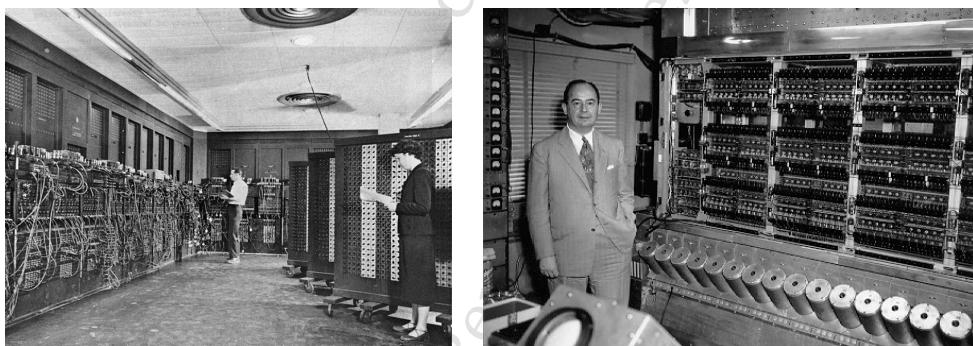
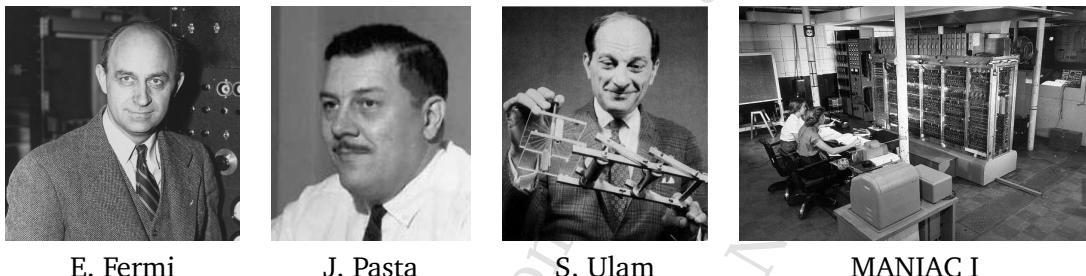


Figura 1.1: ENIAC (izquierda) y J. Von Neumann posando junto a MANIAC-I (derecha).

los ordenadores electrónicos se remonta a finales del 1945, principios del 1946 y estuvo muy relacionado con el esfuerzo de los aliados (en este caso americanos y británicos) por descifrar los famosos códigos alemanes. De hecho durante bastante tiempo solo algunas instituciones tenían alguna de estas máquinas que usualmente eran financiadas con el presupuesto militar de los EEUU. No es hasta mediados de 1952 que el físico Enrico Fermi (1901–1954) propuso resolver numéricamente con ayuda de la computadora MANIAC-I (Mathematical Analyzer Numerical Integrator And Calculator) del Laboratorio Nacional de Los Alamos (EEUU) — Laboratorio donde nacieron las bombas atómicas y de hidrógeno del ejército norteamericano— un sistema de ecuaciones no lineales asociada con cierto sistema de osciladores no lineales, para lo que se asoció con el especialista en

¹En realidad el primer ordenador *digital* fue el ENIAC (Electronic Numerical Integrator And Calculator) pero su programa estaba conectado al procesador y debía ser modificado manualmente, es decir si queríamos resolver un problema distinto había que cambiar las propias conexiones en el ordenador, i.e., cambios en el programa implicaban cambios en la estructura de la máquina (*hardware*). Fue el polifacético John von Neumann quien propuso que se separaran los programas (*software*) del *hardware*, por lo que se le considera el padre de los ordenadores modernos.

computación John Pasta (1918—1984) y el matemático Stanislaw Ulam (1909—1984) y la física y programadora del MANIAC, Mary Tsingou-Menzel. Aunque nunca se llegó a publicar el trabajo debido a la prematura muerte de Fermi (apareció como preprint titulado “Studies of nonlinear problems. I” y más tarde fue incluido en *Collected Papers of Enrico Fermi*, E. Segré (Ed.), University of Chicago Press (1965) p. 977-988, con una introducción de Ulam) fue un trabajo extremadamente influyente y marcó la fecha del nacimiento de la *matemática experimental* (para más detalle véase el magnífico artículo de M.A. Porter, N.J. Zabusky, B. Hu y D.K. Campbell, “Fermi, Pasta, Ulam y el nacimiento de la matemática experimental”, *Revista Investigación y Ciencia* 395 Agosto de 2009, p. 72–80). Aunque M. Tsingou fue quien desarrolló el algoritmo y programó MANIAC su nombre solo fue reconocido a partir de 2008 gracias al artículo de T. Dauxois en Physics Today 61, 1 (2008) 55-57 (<https://arxiv.org/abs/0801.1590>).

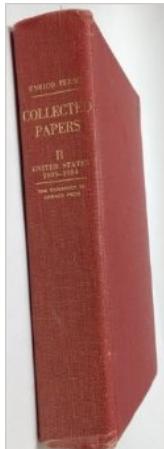


E. Fermi

J. Pasta

S. Ulam

MANIAC I



266.

STUDIES OF NON LINEAR PROBLEMS

E. FERMI, J. PASTA, and S. ULAM
Document LA-1940 (May 1955).

ABSTRACT.

A one-dimensional dynamical system of 64 particles with forces between neighbors containing nonlinear terms has been studied on the Los Alamos computer MANIAC I. The nonlinear terms considered are quadratic, cubic, and broken linear types. The results are analyzed into Fourier components and plotted as a function of time.

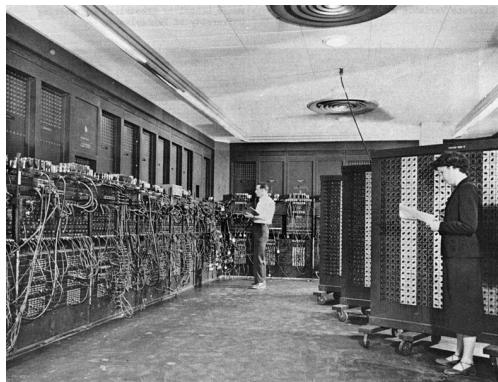
Figura 1.2: El famoso artículo de Fermi, Pasta y Ulam: *Studies of nonlinear problems*, tomado de las obras completas de Fermi pág. 979.

Hoy día un enorme número de personas utiliza diariamente los ordenadores, especialmente para navegar por internet, editar textos, jugar, etc. Un *smartphone* moderno tiene una potencia de cálculo muy superior a la del MANIAC-I y nos cabe en un bolsillo².

No obstante, sería deseable que, aparte de los muchos usos lúdicos que sin duda tienen los ordenadores, también aprendiésemos a utilizarlos como herramienta para resolver problemas y proyectos relacionados con las matemáticas y demás ciencias. De esta forma se puede, entre otras cosas, *comprobar* las predicciones analíticas (teóricas) de una teoría (tanto matemática como en el contexto del resto de las ciencias) mediante experimentos

²ENIAC por ejemplo, ocupaba 167m², pesaba 27000 Kg y costó medio millón de dólares (6 millones de dólares actuales) y sólo realizaba 200 ciclos básicos por segundo. Un ciclo básico consiste en la siguiente secuencia de acciones: buscar datos, decodificar, ejecutar y guardar. Muchos *smartphones* actuales tienen procesadores que realizan más de un millón de ciclos por segundo.

ENIAC – año 1946



Precio 6 millones de dólares
Ocupaba 167 m² y pesaba 27 000 Kg
ciclos básicos por segundo: 200

Toshiba z830 – año 2012



Precio 2000 dólares \$
Ocupa 0.07 m² y pesa 1.12 Kg
ciclos básicos por segundo: > 10⁶.

Figura 1.3: Comparación entre el primer ordenador moderno y un portátil actual.

numéricos. Exagerando un poco podemos perfectamente suscribir la afirmación del genial matemático ruso (soviético) V.I. Arnold (1937–2010) quien afirmaba³ que “*la Matemática es la parte de la Física donde los experimentos son baratos*” y que podemos extrapolar no sólo a la física sino al resto de las ciencias experimentales. Por todo ello es conveniente aprender a trabajar con un programa de cálculo matemático. Llegados a este punto se plantea el problema de qué software elegir. La elección de un software libre es, seguramente, la mejor opción ya que aparte de no tener ningún coste adicional, están escritos en código abierto disponible para casi todas las posibles plataformas: UNIX, LINUX, WINDOWS, MACINTOSH, etc. Dentro de la amplia gama de software matemático libre tenemos, por ejemplo, MAXIMA (programa basado en LIPS de corte simbólico/numérico) y OCTAVE (escrito en C++, para cálculo numérico esencialmente), programas matemáticos con licencia GNU/GPL (y por tanto gratuitos y de distribución libre) accesibles por internet en

- <http://maxima.sourceforge.net> y
- <http://www.gnu.org/software/octave>,

respectivamente⁴.

En estas notas vamos a describir brevemente como funciona el programa de cálculo simbólico (y numérico) MAXIMA. La elección de MAXIMA no es arbitraria. En primer lugar hay que destacar que es un programa que permite tanto el cálculo *simbólico* (es decir, como si lo hiciésemos con lápiz y papel) como numérico; en segundo lugar, tiene una comunidad de usuarios muy numerosa, con foros de discusión, etc. Por último, pero no por ello menos importante, existen aparte del manual de referencia [1] una gran cantidad de textos de acceso libre, muchos de ellos accesibles desde la propia web del programa <http://maxima.sourceforge.net>. Sin pretender dar una lista completa caben mencionar

³Ver el magnífico ensayo “On teaching mathematics” (Sobre la enseñanza de las Matemáticas) publicado en Russian Math. Surveys **53** (1998), No. 1, 229-236).

⁴Una lista bastante completa de las opciones se puede consultar en la wikipedia http://es.wikipedia.org/wiki/Software_matem%C3%A1tico

los siguientes:⁵

1. Jerónimo Alaminos Prats, Camilo Aparicio del Prado, José Extremera Lizana, Pilar Muñoz Rivas y Armando R. Villena Muñoz, *Prácticas de ordenador con wxMAXIMA*, (Granada, 2010).
2. José Manuel Mira Ros, *Manual de MAXIMA*. Disponible desde su web en la Universidad de Murcia: <http://webs.um.es/mira/maxima.php>
3. Rafa Rodríguez Galván, *MAXIMA con wxMAXIMA: software libre en el aula de matemáticas* (Cádiz, 2007)
4. Mario Rodríguez Riotorto, *Primeros pasos en MAXIMA* (Ferrol, 2015). Disponible desde la web del autor. <http://www.tecnostats.net>
5. José Antonio Vallejo Rodríguez, *Manual de uso de MAXIMA y wxMAXIMA en asignaturas de cálculo diferencial* (Méjico, 2008)

Por comodidad usaremos la interfaz gráfica wxMAXIMA (otra opción posible es XMAXIMA) que se puede descargar gratuitamente desde la página de wxMAXIMA o, si se usa LINUX, mediante el instalador de paquetes de la correspondiente distribución. En cualquier caso es conveniente asegurarse que GNUPLOT y MAXIMA están instalados.

Las notas están organizadas de la siguiente forma: el capítulo 2 constituye una brevísimas introducción a MAXIMA; el capítulo 3 desarrolla parte del potencial gráfico de MAXIMA; en el capítulo 4 se discuten temas más avanzados que en el introductorio 2 como son la resolución de ecuaciones algebraicas, la manipulación de expresiones algebraicas, el cálculo matricial y el tratamiento de datos; los capítulos 5, 6 y 7 se centran en el estudio de las ecuaciones diferenciales. Finalmente, en el capítulo 8 se resuelven y proponen diversos problemas usando MAXIMA. A las notas les acompañan las sesiones en el entorno wxMAXIMA. Cada fichero comienza con un distintivo de la sección. Por ejemplo, el s2 del fichero s2-maxima-intro.wxm indica que dicho fichero es la sesión usada en el capítulo 2; s8-2-3-cuadraturas.wxm será el fichero usado en el capítulo 8 (s8) secciones 2 y 3, etc. En la medida de lo posible cada fichero de wxm contiene la misma numeración de las subsecciones del libro. Dichos ficheros se encuentran disponibles en la página web del autor <https://renato.rynfismat.es/clases.html#maximabook>.

⁵Conviene visitar también la web <http://andrejv.github.io/wxmaxima/help.html> que contiene muchos manuales en inglés muy interesantes.

Capítulo 2

Primeros pasos con MAXIMA.

2.1. Empezando: primeras operaciones.

MAXIMA es un programa que funciona como una calculadora científica. Las operaciones aritméticas elementales son las habituales: + suma, - resta, * multiplicación, / división, ^ potenciación:¹

```
(%i1) 2+2; 3-3; 2*3; 5/10; 3^3;  
(%o1) 4  
(%o2) 0  
(%o3) 6  
(%o4) 1/2  
(%o5) 27
```

La notación que usa MAXIMA es (%in) y (%om) para indicar la *entrada* (inchar o input) n -ésima y la *salida* (outchar u output) m -ésima. Usando, por ejemplo, las órdenes inchar: "input"; y outchar: "output"; podemos cambiar las (%in) y (%om) por (%inputn) y (%outputm), respectivamente.

Que las salidas y entradas estén numeradas no es por casualidad. Podemos recuperar la n -ésima salida simplemente usando %on donde n es el número de la salida (output) correspondiente. Así %o4 nos devolvería la salida número 4, i.e. 1/2.

Dado que MAXIMA es un programa de cálculo *simbólico*, trabaja con variables definidas por símbolos (usualmente letras o letras y números):

```
(%i6) e;  
(%o6) e
```

Las funciones (comandos) tienen el argumento (o los argumentos) que van entre paréntesis, como por ejemplo el comando float(x) que nos da como resultado el valor numérico de la variable x . En este ejemplo e representa una variable que no tiene ningún argumento asignado:

```
(%i7) float(e);
```

¹Normalmente para que MAXIMA calcule es necesario apretar al mismo tiempo las teclas “Shift” (mayúsculas) y “Enter”.

```
(%o7) e
(%i8) pi;
(%o8) pi
(%i9) float(pi);
(%o9) pi
```

por lo que su respuesta es la propia variable. Nótese que si escribimos “pi”, (o π , lo que se puede conseguir con la combinación “Esc” p “Esc”) el resultado es pi. Para los valores numéricos de las constantes e (base de los logaritmos neperianos) o π MAXIMA usa una notación distinta:

```
(%i10) %e;
(%o10) %e
(%i11) float(%e);
(%o11) 2.718281828459045
```

Si no se le dice lo contrario, MAXIMA trabaja en doble precisión, i.e., con 16 dígitos. Dado que es un programa simbólico, podemos fijar con cuantas cifras queremos trabajar². Para ello hay que definir la variable `fpprec`. Para asignar valores a las variables dadas se usan los “dos puntos”. Por ejemplo si escribimos `e : %e` habremos definido la variable e con el valor de la base de los logaritmos neperianos. Vamos a definir el grado de precisión en 50 cifras significativas:

```
(%i12) fpprec:50;
(%o12) 50
```

y lo comprobamos pidiendo a MAXIMA que nos de los valores de e y π con 50 cifras, para lo cual hay que usar el comando `bfloat`

```
(%i13) bfloat(%e); bfloat(%pi);
(%o13) 2.7182818284590452353602874713526624977572470937b0
(%o14) 3.1415926535897932384626433832795028841971693993751b0
```

Aquí, la notación $b0$ indica $\times 10^0 = 1$. Calculemos ahora e^7 y pidamos a MAXIMA que escriba los valores numéricos del mismo:

```
(%i15) float(%e^7);
(%o15) 1096.633158428459
(%i16) bfloat(%e^7);
(%o16) 1.0966331584284585992637202382881214324422191348336b3
(%i17) %e^7;
(%o17) %e^7
```

Así si escribimos `%e^7` la salida es simplemente e^7 pues MAXIMA, como hemos dicho, trabaja simbólicamente. Si usamos `float`, la salida es en precisión normal, y sólo si usamos `bfloat` nos devuelve el resultado en la precisión definida previamente de 50 cifras. Probar como ejercicio que ocurre con e^{70} .

El orden de realización de las operaciones es el habitual. Así en la expresión

²Aunque formalmente MAXIMA trabaja con precisión infinita (o sea simbólicamente), si queremos obtener

```
(%i18) (2+3^2)^3*(5+2^2);
(%o18) 11979
```

primero calcula las potencias dentro de cada paréntesis, luego las sumas, luego las potencias externas y finalmente la multiplicación.

Como hemos mencionado, para definir los valores de las variables se usan los dos puntos. En la secuencia que aparece a continuación se definen la x y la y para que valgan 123 y 321, respectivamente, dejándose la z libre.

```
(%i19) x:123; y:321; x*y; x/y; x-y;
(%o19) 123
(%o20) 321
(%o21) 39483
(%o22) 41/107
(%o23) -198
(%i24) 123*321;
(%o24) 39483
(%i25) x;
(%o25) 123
(%i26) z=2;
(%o26) z=2
(%i27) z;
(%o27) z
(%i28) x;
(%o28) 123
(%i29) y;
(%o29) 321
(%i30) x*y;
(%o30) 39483
(%i31) x+z;
(%o31) z+123
```

De lo anterior se sigue, que, por ejemplo, al realizar la división x/y , MAXIMA simplifica al máximo el resultado. Nótese también que el resultado de la multiplicación de $x * y$ es precisamente el valor de $123 * 321$. Como vemos en la salida (%o26) la expresión $z=2$ no asigna el valor 2 a la variable z , ya que en realidad para MAXIMA $z=2$ es una ecuación (como veremos más adelante).

Es importante destacar, además, que hay que escribir el símbolo de cada multiplicación pues, si por ejemplo escribimos x y en vez de $x*y$ obtenemos un mensaje de error

```
Incorrect syntax: Y is not an infix operator
SpacexSpacey;
^
```

También es posible definir funciones. Hay múltiples formas en función de lo queramos hacer. La más sencilla es mediante la secuencia $::=$

un valor numérico MAXIMA necesita implementar algún algoritmo computacional por lo que el cálculo efectivo dependerá de la potencia del ordenador y el número de dígitos que queramos. A efectos prácticos con los 16 dígitos con que se trabaja por defecto es más que suficiente a la hora de trabajar numéricamente.

```
(%i32) f(x):= x^2 -x + 1;
(%o32) f(x):=x^2-x+1
(%i33) f(%pi);
(%o33) %pi^2-%pi+1
(%i34) float(%);
(%o34) 7.728011747499565
(%i35) float(f(%pi));
(%o35) 7.728011747499565
```

Nótese que a no ser que pidamos a MAXIMA que trabaje numéricamente, sigue usando cálculo simbólico (ver el valor de $f(\pi)$ de la salida 33).

Otro detalle interesante a tener en cuenta es que MAXIMA contiene una ayuda completa que puede ser invocada desde la propia línea de comandos. Para ello preguntamos a MAXIMA con ?? delante del comando *desconocido*

```
(%i36) ??limit;
0: Functions and Variables for Limits
1: limit  (Functions and Variables for Limits)
2: List delimiters  (Functions and Variables for Lists)
3: tlimit  (Functions and Variables for Limits)
4: zn_primroot_limit  (Functions and Variables for Number Theory)
```

La respuesta es una lista de opciones de funciones de MAXIMA que contiene la cadena de letras que se han escrito y MAXIMA se queda esperando que elijamos una opción. Las posibilidades son: uno de los números (por ejemplo 1), varios números separados por espacios en blanco (por ejemplo 1 4), all (imprime en pantalla todas las opciones) o none que no imprime nada.

Luego de introducir la opción (en este ejemplo escogeremos 1) obtenemos una explicación detallada de la sintaxis y de lo que hace dicho comando y, en muchos casos, ejemplos de utilización:

```
Enter space-separated numbers, 'all' or 'none': 1;
-- Function: limit
limit (<expr>, <x>, <val>, <dir>)
limit (<expr>, <x>, <val>)
limit (<expr>)
Computes the limit of <expr> as the real variable <x> approaches
the value <val> from the direction <dir>. <dir> may have the value
'plus' for a limit from above, 'minus' for a limit from below, or
may be omitted (implying a two-sided limit is to be computed).

etc.

(%o36) true
```

Si no existe ningún comando con ese nombre la respuesta es false (en caso contrario, al final de las explicaciones, la salida es true).

```
(%i37) ??renato;
(%o37) false
```

Otra de las interesantes opciones de MAXIMA es su potencia gráfica. Para hacer las gráficas MAXIMA usa GNUPLOT, un potente paquete gráfico GNU. El comando más sencillo de usar es `plot2d` (o si usas como interfaz gráfica el `wxMAXIMA`, `wxplot2d` que incrusta el gráfico en la propia sesión de `wxMAXIMA`). Su sintaxis es muy simple

```
wxplot2d([funcion1,funcion2,...],[variable,inicio,final],opciones)
```

Ahora bien, para usar ese comando tenemos que ser muy cuidadosos. Si hacemos en nuestro caso

```
(%i39) wxplot2d([f(x)], [x,-5,5]);
Bad range: [123,-5,5].
Range must be of the form [variable,min,max]
-- an error. To debug this try: debugmode(true);
```

obtenemos un error, que simplemente, leyendo nos indica que la variable `x` ya está asignada. Eso se resuelve cambiando la variable por otra no usada o, usando una variable *muda*, es decir una variable que solo se use dentro del propio comando `wxplot2d`. Estas variables se definen colocando delante de la misma el signo ', por ejemplo, en vez de `x` usamos '`x`

```
(%i40) wxplot2d([f('x)], ['x,-5,5])$
```

que nos muestra la gráfica en el propio entorno de `wxMAXIMA` —ver la gráfica de la izquierda en la figura 2.1—. Otra opción es usar `plot2d` que saca el resultado en una

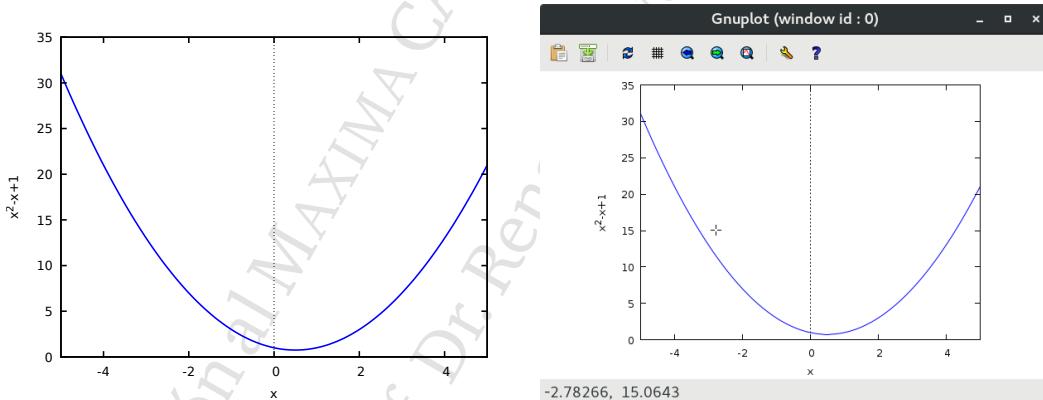


Figura 2.1: Gráfica de la función de la salida (%o32) $f(x) = x^2 - x + 1$ (izquierda) tal como se ve al usar la orden `wxplot2d([f('x)], ['x,-5,5])` y pantalla emergente que vemos si escribimos `plot2d([f('x)], ['x,-5,5])`. Para esta última, debajo a la izquierda, se muestran las coordenadas del cursor.

pantalla aparte (de `GNUPLOT`) y que permite algunas interacciones muy útiles. Por ejemplo marcando con el ratón en un punto de la ventana emergente podemos ver sus coordenadas en el ángulo izquierdo inferior (ver figura 2.1).

Por otro lado, si pinchamos con el botón derecho del ratón y nos movemos a otro punto de la ventana se abre un rectángulo azul y al pinchar el botón derecho obtenemos un zoom (ampliación) de dicho rectángulo. Un ejemplo lo vemos en la figura 2.2.

La combinación de estas dos operaciones son muy útiles a la hora de resolver problemas. Por ejemplo si necesitamos saber aproximadamente donde está el cero de una función, etc.

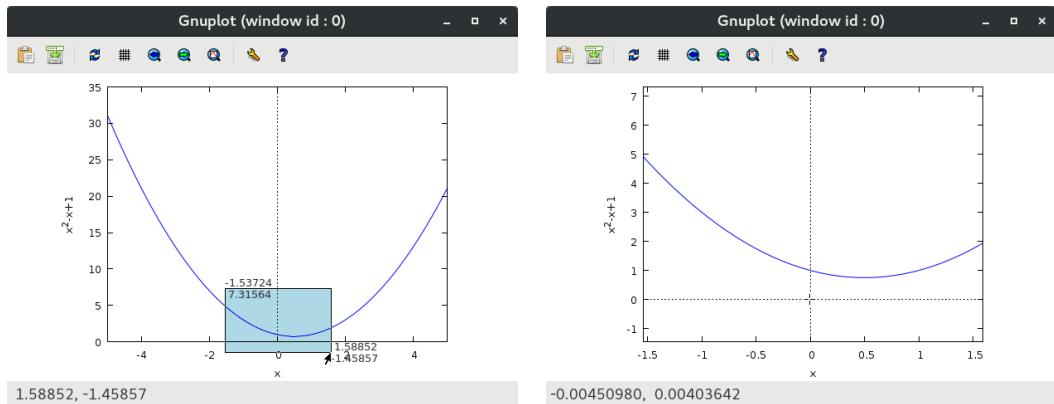


Figura 2.2: Escogiendo una región de la gráfica de la función para hacer una ampliación de la misma (izquierda) y resultado de dicha ampliación (derecha).

También podemos representar varias gráficas al mismo tiempo. Para ello creamos una lista³ de funciones de la siguiente forma $[f_1(x), \dots, f_n(x)]$. En el ejemplo representamos las funciones $f(x)$, $\sin(2x)$ y $\arctan(x)$ —ver gráfica 2.3 (derecha)—.

```
(%i41) wxplot2d([f('x),2*sin('x),atan('x)], ['x,-2,2])$
```

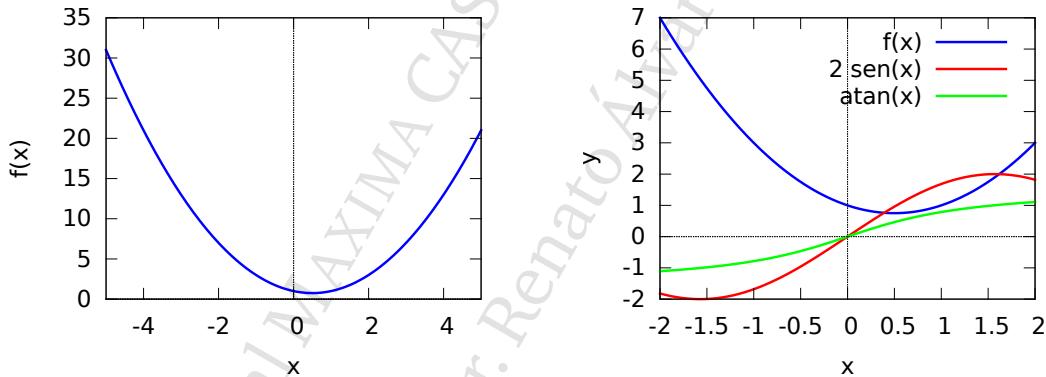


Figura 2.3: Los gráficos de la función de la salida (%o32) $f(x) = x^2 - x + 1$ (izquierda) y el de las tres funciones de la salida (%o41) (derecha).

Si comparamos los gráficos mostrados en la figura 2.3 con los que vemos en nuestra sesión notaremos algunas diferencias. Por ejemplo las líneas de éstos últimos son más finas, aparece la leyenda $f(x)$ en vez de $x^2 - x + 1$, el tamaño de las letras, etc. La razón está en las opciones gráficas que no vamos a discutir aquí y que pospondremos hasta el capítulo 3. En particular, en el apartado 3.1.3 veremos como exportar gráficas en distintos formatos que nos permitan la inclusión de los mismos en documentos de texto (en este caso LATEX). En cualquier caso para el caso de la gráfica de la derecha en 2.3 podemos usar la orden

```
wxplot2d([f('x),2*sin('x),atan('x)], ['x,-2,2], [ylabel,"y"],
```

³Las listas no son más que expresiones de la forma $[\text{objeto}_1, \dots, \text{objeto}_N]$, donde $\text{objeto}_1, \dots, \text{objeto}_N$ son objetos cuales quiera (números, variables, gráficas, listas, ...). Para más detalle véase la página

```
[style,[lines,3]], [legend,"f(x)","2 sen(x)","atan(x)"]);
```

MAXIMA también dispone de una gran cantidad de funciones *elementales* (exponencial, logaritmo, trigonométricas, etc.). Además las trata de forma simbólica como se puede ver en la secuencia que sigue. Nótese que la función `log` denota a los logaritmos neperianos (probar con la función `ln`).

```
(%i42) log(10);
(%o42) log(10)
(%i43) float(%);
(%o43) 2.302585092994046
(%i44) log(%e);
(%o44) 1
```

También podemos calcular factoriales

```
(%i45) fac:15!;
(%o45) 1307674368000
```

En el próximo apartado se incluye una lista de algunas de las funciones más usadas en MAXIMA.

Otra de las bondades de MAXIMA es que cuenta con un gran repertorio de comandos de manipulación algebraica, basta pinchar en la pestaña “Simplificar” y veremos las muchas posibilidades. A modo de ejemplo factorizaremos el factorial anterior (que al ser un número, MAXIMA lo que hace es encontrar sus factores primos).

```
(%i46) factor(fac);
(%o46) 2^11*3^6*5^3*7^2*11*13
```

Podemos factorizar expresiones algebraicas:

```
(%i47) factor(x^2+x+6);
(%o47) 2*3*2543
```

¡Ops! ¿qué es esto?

```
(%i48) x;
(%o48) 123
```

Claro, `x` estaba asignada al valor 123. Una opción es usar variables mudas

```
(%i49) factor('x^2 + 'x -6);
(%o49) (x-2)*(x+3)
```

o bien, limpiar el valor de una variable, para lo cual usamos el comando `kill`

```
(%i50) kill(x); x;
(%o51) done
```

Ahora ya podemos usar sin problemas los comandos de simplificación usando la variable x , como por ejemplo, `ratsimp` que

```
(%i52) ratsimp((x^2-1)/(x-1));
(%o52) x+1
```

Si en vez de factorizar queremos desarrollar en potencias usamos la orden `expand`. Su sintaxis es `expand(expr,n,m)`. Es importante destacar que esta orden desarrolla las potencias que van desde $-m$ a n en la expresión `expr`. Como ejercicio comprobar la salida de `expand` para la expresión $(x+1)^{10} + 2(x-3)^{32} + (x+2)^2 + 1/x + 2/(x-1)^2 + 1/(x+1)^3$ usando las órdenes

```
expand(expr); expand(expr,1); expand(expr,3,2);
```

Conviene saber que los comandos `functions` y `values` nos dicen en cada momento las funciones y las variables que están asignadas en la sesión. Si queremos saber cual es el la función asignada a cierta expresión, por ejemplo a f , escribimos `fundef(f)`.

Si queremos limpiar todas las variables⁴ de nuestra sesión podemos usar el comando `kill(all)`

```
(%i53) kill(all);
(%o0) done
```

Nótese que el número del output es 0. Esta orden limpia toda la memoria, incluido los paquetes cargados previamente, por lo que si queremos volverlos a usar hay que cargarlos nuevamente.

Las ecuaciones se definen en MAXIMA de forma *habitual*, mediante el signo igual, i.e., *miembro izquierdo = miembro derecho*, por ejemplo $x^2-4=0$ define la ecuación $x^2-4=0$. Lo interesante es que, definida una ecuación, podemos resolverla *analíticamente* con el comando `solve`:

```
(%i1) solve(x^2-4=0,x);
(%o1) [x=-2,x=2]
```

Además, MAXIMA lo hace simbólicamente, como ya hemos mencionado más de una vez. Incluso da las soluciones complejas si es necesario

```
(%i2) solve(x^2-4=a,x);
(%o2) [x=-sqrt(a+4),x=sqrt(a+4)]
(%i3) solve(x^2=-1);
(%o3) [x=-%i,x=%i]
```

Resolvamos ahora la ecuación⁵ $x^5 = 1$. Como se ve MAXIMA la resuelve exactamente:

[14.](#)

⁴Esto es muy recomendable cuando comenzemos a trabajar en un nuevo problema o queremos repetir los cálculos para asegurarnos que no hemos cometido errores. Otra opción es ir a la pestaña “Maxima” y elegir “Reiniciar”.

⁵Muchas veces a lo largo del texto no explicaremos alguno de los comandos que aparecen. El lector puede simplemente usar la ayuda de MAXIMA que ya comentamos en la página [8](#). En este caso nos aparece la orden `rectform`.

```
(%i4) solve(x^5=1,x);
(%o4) [x=%e^((2*i*pi)/5), x=%e^((4*i*pi)/5),
      x=%e^(-(4*i*pi)/5), x=%e^(-(2*i*pi)/5), x=1]
(%i5) rectform(%);
(%o5) [x=%i*sin((2*pi)/5)+cos((2*pi)/5),
      x=%i*sin((4*pi)/5)+cos((4*pi)/5),
      x=cos((4*pi)/5)-%i*sin((4*pi)/5),
      x=cos((2*pi)/5)-%i*sin((2*pi)/5),
      x=1]
```

Como ejercicio resolver la ecuación $x^n = 1$.

Pero el comando `solve` no siempre funciona

```
(%i6) solve(x^7+2*x+1);
(%o6) [0=x^7+2*x+1]
```

¿Qué ha pasado? Pues que no existe una solución analítica para esta ecuación y MAXIMA, simplemente, da como salida la misma entrada.

Finalmente, para grabar la sesión, podemos usar la opción “Guardar” en la pestaña “Archivo” de wxMaxima.

Antes de pasar al siguiente apartado sobre listas, debemos mencionar que para que MAXIMA reconozca el directorio local de trabajo (lo que es importante a la hora de importar y exportar ficheros, por ejemplo) es conveniente definir la variable `file_search_maxima`. En este ejemplo (con LINUX) el directorio de búsqueda es `/home/renato/`. La forma más sencilla de hacerlo es mediante la opción “Añadir a la ruta” (*Add to path*) en la pestaña “Maxima” del menú del programa:

```
(%i7) file_search_maxima : cons(sconcat(
"/home/renato/##.{lisp,mac,mc}"), file_search_maxima)$
```

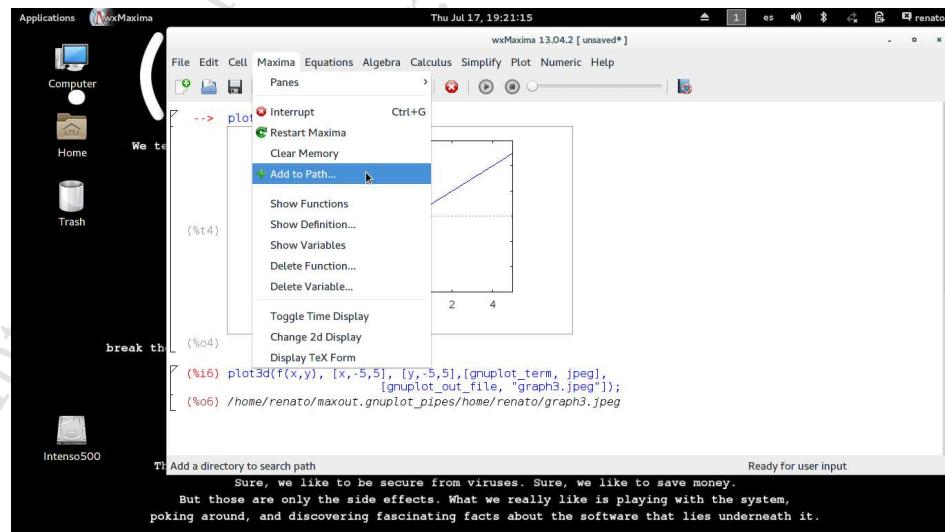


Figura 2.4: Fijando el directorio de trabajo con wxMAXIMA

2.2. Listas.

Hemos tenido necesidad antes de crear una lista, es decir expresiones de la forma `[objeto1, ..., objetoN]`, donde `objeto1, ..., objetoN` son objetos cuales quiera (números, variables, gráficas, listas, ...). El uso de listas y el tratamiento de las mismas es de gran importancia en MAXIMA así que veamos brevemente como tratar estos objetos.

Las listas pueden ser de cualquier tipo de *objetos*. Por ejemplo

```
(%i1) l:[a,2,sin(x),"hola soy renato",y+z];
(%o1) [a,2,sin(x),"hola soy renato",z+y]
```

donde "hola soy renato" es una cadena de caracteres. Las cadenas son objetos interesantes de mucha utilidad en MAXIMA. Aunque apenas las vamos a usar en estas notas conviene saber que hay un paquete específico para tratar con ellas: `stringproc` cuya descripción se puede encontrar en [1].

Si queremos sacar el k -ésimo elemento de la lista l anterior escribimos `l[k]`

```
(%i2) l[3];
(%o2) sin(x)
(%i3) l[4];
(%o3) "hola soy renato"
```

Los primeros diez elementos de una lista también se pueden obtener mediante los comandos `first`, `second`, ..., `tenth`.

Las listas se pueden crear si conocemos una expresión generadora. Para ello usamos el comando `makelist` cuya sintaxis es

```
makelist( expr , k , k_i , k_f , paso )
```

que lo que hace es generar una lista donde la entrada k -ésima es el resultado de evaluar la expresión `expr` para cada `k` desde `k_i` inicial hasta `k_f` final con paso `paso`.

Por ejemplo, vamos a generar la lista de los números de la forma $k^2 + 1$ comenzando por $k = 0$ hasta $k = 5$ escribimos

```
(%i4) makelist(k^2+1,k,0,5);
(%o4) [1,2,5,10,17,26]
```

El valor por defecto para el paso es 1. Si queremos otro paso distinto debemos especificarlo, por ejemplo `makelist(k^2,k,0,5,1/2)`.

También MAXIMA tiene otras formas de escribir las listas. Por ejemplo,

```
(%i5) makelist(k^2+1,k,i,i+3,1/2);
(%o5) [i^2+1,(i+1/2)^2+1,(i+1)^2+1,(i+3/2)^2+1,(i+2)^2+1]
```

Otra opción es

```
makelist( expr , k , list)
```

cuya salida es una lista de longitud `list` cuyos elementos son el resultado de evaluar la expresión `expr` en `k` que toma los valores de la lista `list`

```
(%i6) makelist (x^y, y, [a, b, c]);
(%o6) [x^a,x^b,x^c]
```

Existe otro comando más flexible para crear listas: `create_list` que no trataremos aquí y remitimos al lector al manual [1].

Si queremos agregar una lista `lista2` al final de la lista `lista1` usamos en comando `append(lista1,lista2)`. Otra opción es escribir `append(lista1,lista2,...,listan)` que genera una nueva lista donde las listas `lista1`, `lista1`, etc. van concatenadas.

```
(%i7) append(l, [z+7,x-2]);
(%o7) [a,2,sin(x),"hola soy renato",z+y,z+7,x-2]
```

Si queremos mezclar dos listas `l` y `m` de forma que el resultado sea un elemento de cada una de ellas de la forma `l[1]`, `m[1]`, `l[2]`, `m[2]`, ... usamos el comando `join (l, m)`

```
(%i8) ll:join (l, [1, 2, 3, 4,5]);
(%o8) [a,1,2,2,sin(x),3,"hola soy renato",4,z+y,5]
```

El comando `last(list)` devuelve el último elemento de la lista `list` mientras que la orden `length(list)` da el número de elementos de la lista `list`.

```
(%i9) last(l);
(%o9) z+y
(%i10) length(l);
(%o10) 5
```

Si de la lista `lista` queremos eliminar los primeros n elementos escribimos `rest(lista,n)` si n es negativo entonces eliminamos los n últimos elementos.

```
(%i11) rest(l, 3);
(%o11) ["hola soy renato",z+y]
(%i12) rest(l, -3);
(%o12) [a,2]
```

Para invertir el orden de una lista usamos el comando `reverse(list)`

```
(%i13) l;
(%o13) [a,2,sin(x),"hola soy renato",z+y]
(%o14) reverse(l);
(%o14) [z+y,"hola soy renato",sin(x),2,a]
```

Otros comando interesante es `sublist` cuya sintaxis es `sublist(list, p)` que devuelve una lista con los elementos de `list` para los cuales el valor de `p` es true. Por ejemplo

```
(%i15) sublist (ll, evenp);
(%o15) [2,2,4]
(%i16) sublist (ll, oddp);
(%o16) [1,3,5]
```

Más adelante veremos ejemplos donde nos interesa conocer la posición de cierto elemento en una lista. Para ello MAXIMA cuenta con el comando `sublist_indices(lista,P)` que devuelve el índice de los elementos de la lista `lista` para los cuales el valor del predicado `maybe(P(x))` es true. Antes de ver ejemplos conviene entender lo que hace el comando `maybe`.

Cuando escribimos `maybe(expr)` lo que hace `maybe` es intentar determinar si el predicado `expr` se puede deducir de los hechos almacenados en la base de datos gestionada por la orden `assume`. Si el predicado se reduce a `true` o `false`, `maybe` devuelve `true` o `false`, respectivamente. En otro caso devuelve `unknown`. Por ejemplo:

```
(%i17) maybe (x > 0);
(%o17) unknown
(%i18) assume(x>1);
(%o18) [x>1]
(%i19) maybe (x > 0);
(%o19) true
(%i20) maybe (x <= 0);
(%o20) false
```

Como se ve al escribir `assume (pred_1, ..., pred_n)` estamos obligando a MAXIMA que defina los predicados (o situaciones) definidas por `pred_1, ..., pred_n` dentro de la sesión. Hay que tener en cuenta que únicamente son válidos los predicados que cuentan con los operadores de relación⁶ `<`, `<=`, `equal`, `notequal`, `>=`, `>`. Para saber en cada momento que situaciones (o predicados) hemos fijado en la sesión tenemos el comando `facts()`.

```
(%i21) facts();
(%o21) [x>1]
```

Nótese que gracias al comando `assume` se pueden hacer suposiciones generales sobre las variables, algo tremadamente útil en muchos casos como veremos luego.

Volvamos con la orden `sublist_indices(lista,P)` mencionada antes mostrando unos ejemplos clarificadores.

```
(%i22) sublist_indices ('[a, b, b, c, 1, 2, b, 3, b], symbolp);
(%o22) [1,2,3,4,7,9]
(%i23) sublist_indices ('[a, b, b, c, 1, 2, b, 3, b], numberp);
(%o23) [5,6,8]
```

Aquí, el predicado `symbolp` nos da la salida `true` si tenemos un símbolo (no un número) y `false` en cualquier otro caso, mientras que `numberp` es `true` si tenemos un número.

También podemos usarlo en combinación con la función `identity` que devuelve su argumento cualquiera que sea éste.

```
(%i24) sublist_indices ([1 > 0, 1 < 0, 2 < 1, 2 > 1, 2 > 0], identity);
(%o24) [1,4,5]
(%i25) sublist_indices ([x > 0, x < 0, x < -2], identity);
(%o25) [1]
```

⁶No podemos usar el símbolo `=` pues está reservado para las ecuaciones así que para los operadores de

Nótese que en la última salida obtenemos el índice 1 porque en nuestra sesión hemos asumido que $x > 1$. Si eliminamos esa suposición con la orden `forget` (también `kill(all)` limpialas asignaciones de este tipo) la salida es nula pues en ese caso `maybe` siempre nos da la respuesta `unknown`⁷

```
(%i26) forget(x>1);
(%o26) [x>1]
(%i27) facts();
(%o27) []
(%i28) sublist_indices ([x > 0, x < 0, x < -2], identity);
(%o28) []
```

Finalmente, veamos unos comandos sencillos que pueden ser de utilidad. Por ejemplo dada cierta variable podemos preguntarle a MAXIMA si es una lista:

```
(%i1) kill(all)$
lista:[1,a,sin(x),[1,2],"hola",x+2];
(lista)[1,a,sin(x),[1,2],"hola",x+2]
(%i2) listp(lista);
(%o2) true
```

También podemos extraer el elemento k -ésimo de la lista con la orden `part(lista,k)`. Por ejemplo

```
(%i3) part(lista,3);
(%o3) sin(x)
```

cuya salida es efectivamente el tercer elemento de nuestra lista `lista`. También le podemos preguntar a MAXIMA si cierta expresión es parte de la lista-

```
(%i4) member(hola,lista);
(%o4) false
(%i5) member("hola",lista);
(%o5) true
```

Nótese que `hola` no es lo mismo que "hola", pues en el primer caso es una simple variable y en el segundo es una cadena. Podemos eliminar un elemento de la lista con la orden `delete(elemento a eliminar, lista)`. Por ejemplo borramos la cadena "hola",

```
(%i6) delete("hola",lista);
(%o6) [1,a,sin(x),[1,2],x+2]
(%i7) lista;
(%o7) [1,a,sin(x),[1,2],"hola",x+2]
```

No obstante como se ve, la orden `delete` no cambia la lista. Si queremos actualizarla debemos reasignarle la salida;

relación se usa `equal`.

⁷Probar la secuencia `maybe(x>0); maybe(x<0); maybe(x<-2);` con, y sin la asunción $x > 1$.

```
(%i8) lista:delete("hola",lista);
(lista)[1,a,sin(x),[1,2],x+2]
(%i9) lista;
(%o9) [1,a,sin(x),[1,2],x+2]
```

Conviene tener cuidado a la hora de crear una lista. Supongamos que escribimos la secuencia:

```
(%i11) A[1]:2;A[2]:1;
ARRSTORE: use_fast_arrays=false; allocate a new property hash table for |$a|
(%o10) 2
(%o11) 1
```

Ya MAXIMA nos indica que algo “raro” está ocurriendo, así que le preguntamos si es una lista y su respuesta es negativa:

```
(%i12) listp(A);
(%o12) false
(%i13) listarray(A);
(%o13) [2,1]
```

De hecho, lo que hemos definido de esta forma es un *macro* o *array*, que es otra de las estructuras de datos que usa MAXIMA y que tiene sus ventajas y desventajas con respecto a las listas y que no vamos a tratar aquí. Para más información sobre los *arrays* y las listas se recomienda al lector consultar el manual de MAXIMA [1].

2.3. Cálculo diferencial e integral con MAXIMA.

Vamos ahora a describir las órdenes más importantes que nos permiten trabajar con los conceptos del cálculo diferencial e integral.

2.3.1. Trabajando con funciones elementales con MAXIMA.

Comenzaremos escribiendo un listado con algunas de las funciones elementales que tiene MAXIMA:

- `abs(expresion)` valor absoluto o módulo de la expresión `expresion`,
- `entier(x)` parte entera de x ,
- `round(x)` redondeo de x ,
- `exp(x)` función exponencial de x ,
- `log(x)` logaritmo neperiano de x ,
- `max(x1, x2, ...)`, `min(x1, x2, ...)` máximo y mínimo de los números x_1, x_2, \dots ,
- `sign(expresion)` signo de la expresión `expresion`. Devuelve una de las respuestas

siguientes: pos (positivo), neg (negativo), zero (cero), pz (positivo o cero), nz (negativo o cero), pn (positivo o negativo), o pnz (positivo, negativo, o cero, i.e. no se sabe),

- `sqrt(x)` raíz cuadrada de x , i.e., \sqrt{x} ,
- `acos(x)` arco coseno de x ,
- `acosh(x)` arco coseno hiperbólico de x ,
- `acot(x)` arco cotangente de x ,
- `acoth(x)` arco cotangente hiperbólico de x ,
- `acs(x)` arco cosecante de x ,
- `acsch(x)` arco cosecante hiperbólico de x ,
- `asec(x)` arco secante de x ,
- `asech(x)` arco secante hiperbólico de x ,
- `asin(x)` arco seno de x ,
- `asinh(x)` arco seno hiperbólico de x ,
- `atan(x)` arco tangente de x ,
- `atan2(y,x)` proporciona el valor de $\arctan(y/x)$ en el intervalo $[-\pi, \pi]$,
- `atanh(x)` arco tangente hiperbólico de x ,
- `cos(x)` coseno de x ,
- `cosh(x)` coseno hiperbólico de x ,
- `cot(x)` cotangente de x ,
- `coth(x)` cotangente hiperbólica de x ,
- `csc(x)` cosecante de x ,
- `csch(x)` cosecante hiperbólica de x ,
- `sec(x)` secante de x ,
- `sech(x)` secante hiperbólica de x ,
- `sin(x)` seno de x ,
- `sinh(x)` seno hiperbólico de x ,
- `tan(x)` tangente de x ,
- `tanh(x)` tangente hiperbólica de x ,
- `gamma(x)` función gamma de Euler,
- `beta(x,y)` función beta de Euler,

- $\text{erf}(x)$ la función error, i.e. $\frac{2}{\pi} \int_0^x e^{-u^2} du$,
- $\text{isqrt}(x)$ parte entera de la raíz cuadrada de x , que debe ser entero,
- $\text{binomial}(x, y)$ número combinatorio $\binom{n}{m} = \frac{n!}{m!(n-m)!}$.

Antes de continuar conviene comentar una opción muy importante relacionada con la forma como MAXIMA trabaja con los logaritmos y las raíces. Esta opción tiene que ver con las variables `logexpand` y `radexpand` que son las que controlan si se simplifican logaritmos y radicales de productos, respectivamente. Por defecto su valor es `true` lo que implica que `expand` no desarrolla dichos productos

```
(%i1) log(a*b);
(%o1) log(a*b)
(%i2) sqrt(a*b);
(%o2) sqrt(a*b)
```

Si queremos que MAXIMA los desarrolle entonces tenemos que asignarle a las variables `logexpand` y `radexpand` el valor `all`

```
(%i3) radexpand:all$ logexpand:all$
(%i5) log(a*b);
(%o5) log(b)+log(a)
(%i6) sqrt(a*b);
(%o6) sqrt(a)*sqrt(b)
(%i7) log(x^r);
(%o7) r*log(x)
(%i8) log(1/b);
(%o8) -log(b)
```

Otra opción a tener en cuenta es el valor `super` para `logexpand` que, además de lo anterior, desarrolla $\log(a/b) = \log(a) - \log(b)$ cuando a/b es un número racional dado.

```
(%i9) log(4/5);
(%o9) log(4/5)
(%i10) logexpand:super$ 
(%i11) log(4/5);
(%o11) log(4)-log(5)
(%i12) log(1/5);
(%o12) -log(5)
(%i13) log(x^r);
(%o13) r*log(x)
(%i14) log(a*b);
(%o14) log(b)+log(a)
```

También es posible extraer de una fracción algebraica su numerador y denominador gracias a los comandos `num` y `denom`, respectivamente:

```
(%i15) p:(x^4-2*x^3-7*x^2+20*x-12)/(x^3+11*x^2+4*x-60);
```

```
(%o15) (x^4-2*x^3-7*x^2+20*x-12)/(x^3+11*x^2+4*x-60)
(%i16) num(p);
(%o15) x^4-2*x^3-7*x^2+20*x-12
(%i17) denom(p);
(%o17) x^3+11*x^2+4*x-60
(%i18) kill(all);
(%o0) done
```

2.3.2. Cálculo diferencial e integral de funciones de una variable.

Veamos ahora como MAXIMA trabaja con los principales conceptos del cálculo diferencial e integral. Ante todo, MAXIMA *sabe* calcular límites $\lim_{x \rightarrow x_0 \pm} f(x)$, para ello usa el comando `limit` cuya sintaxis es

```
limit(f(x),x,x0,dirección)

(%i1) limit(sin(a*x)/x,x,0);
(%o1) a
```

Incluso aquellos que dan infinito

```
(%i2) limit(log(x),x,0);
(%o2) infinity
(%i3) limit(log(x),x,0,plus);
(%o3) -inf
(%i4) limit(log(x),x,0,minus);
(%o4) infinity
```

Debemos aclarar que `infinity` es entendido por MAXIMA como el infinito complejo, mientras que `inf` lo entiende como el $+\infty$ real. Gracias a esto podemos calcular límites infinitos

```
(%i5) limit((1+2/x)^(x^(1/3)),x,inf);
(%o5) 1
```

Conviene saber que la salida de `limit` puede ser `und` (indefinido) y `ind` (indefinido pero acotado). En general, la orden `limit` lo que hace es aplicar la regla de L'Hospital para el cálculo de límites. El número de veces que aplica la Regla está controlado por la variable global `lhospitallim` cuyo valor por defecto es 4, pasado el cual MAXIMA lo intenta usando el polinomio de Taylor, para lo cual es necesario que la variable `tlimswitch` tenga asignado el valor `true`. También existe el comando `tlimit` cuya sintaxis es la misma pero que usa exclusivamente el desarrollo de Taylor de las funciones involucradas.

Como ejercicio calcula los límites $\lim_{x \rightarrow 0} \frac{\log(3x)}{x}$, $\lim_{n \rightarrow \infty} \left(1 + \frac{2}{n}\right)^{\frac{n}{3}}$ y $\lim_{x \rightarrow 2} \frac{x^2 - 4}{x - 2}$.

Muchas veces necesitamos evaluar una salida de MAXIMA en un determinado valor. Por ejemplo $x^2 + 3x + 2$ en $x = 1$, digamos. Entonces usamos en comando `ev` cuya sintaxis es

```
ev(expr,var1,var2,...)
```

que evalúa la expresión `expr` en los valores de `var1`, Por ejemplo

```
(%i6) kill(all)$  
(%i1) ev(x^2+3*x+2,x=1);  
(%o1) 6  
(%i2) ev(x^p+3*x+2,x=2);  
(%o2) 2^p+8  
(%i3) ev(x^p+3*x+2,x=2,p=2);  
(%o3) 12
```

Otra posibilidad es escribir las líneas `x^p+3*x+2,x=2` o `x^p+3*x+2,x=2,p=2`.

Existen más operadores relacionados con la evaluación, como el comando comilla-comilla '''. Para más detalle conviene consultar en el manual [1]. Aquí conviene recordar que el comando comilla simple ' evita la evaluación.

Otro comando sumamente útil que usaremos más adelante el comando `subst` que permite las sustituciones formales y cuya sintaxis es

```
subst(Sustituto,Original,Expresión)  
subst(Ecuación_1,Expresión)  
subst([Ecuación_1,Ecuación_2,...Ecuación_n],Expresión)
```

Por ejemplo,

```
(%i4) subst(v,x+y,(x+y)^3);  
(%o4) v^3  
(%i5) subst(x+y=v,(x+y)^3);  
(%o5) v^3
```

Comprobar, como ejercicio, la salida del comando

```
subst([x+y=v,x-y=u],(x+y)^3*(x-y)^2)
```

Pasemos ahora a describir el comando `diff` que permite calcular derivadas parciales de cualquier orden de una función. Su sintaxis es

```
diff(f(x,y),x,k) o diff(f(x,y),x,k,y,m)
```

donde `f(x)` es la función a la que le vamos a calcular la derivada `k`-ésima respecto a la variable `x` o la `k`-ésima respecto a la variable `x` y la `m`-ésima respecto a la variable `y`, respectivamente. Por ejemplo:

```
(%i6) diff(sin(x^2+2),x);  
(%o6) 2*x*cos(x^2+2)  
(%i7) diff(x^x, x,3);  
(%o7) x^(x-1)*(log(x)+(x-1)/x)+x^x*(log(x)+1)^3+2*x^(x-1)*(log(x)+1)  
(%i8) (sin(x))^x;  
(%o8) sin(x)^x  
(%i9) diff(sin(x)^x, x);  
(%o9) sin(x)^x*(log(sin(x))+(x*cos(x))/sin(x))  
(%i10) kill(all)$
```

Además, como vemos, también funciona para funciones definidas por el usuario

```
(%i1) g(x):= sin(x^2/(1+x^2));
(%o1) g(x):=sin(x^2/(1+x^2))
(%i2) diff(g(x),x,1);
(%o2) ((2*x)/(x^2+1)-(2*x^3)/(x^2+1)^2)*cos(x^2/(x^2+1))
```

Como ya mencionamos al principio MAXIMA recuerda todas las salidas. Como ya vimos, para invocar una salida anterior, digamos la 1, usamos `%1`. En particular, podemos recuperar la última usando simplemente la `%`. Aprovecharemos este ejemplo para hacer una breve descripción de algunos de los comandos de simplificación con los que cuenta MAXIMA.

Por ejemplo, la orden `ratsimp(%)` simplifica la salida anterior (que en nuestro ejemplo era la derivada de $g(x)$) —en este caso la misma operación la haría `ratsimp(%2)`—

```
(%i3) ratsimp(%);
(%o3) (2*x*cos(x^2/(x^2+1)))/(x^4+2*x^2+1)
```

Que podemos intentar factorizar

```
(%i4) factor(%);
(%o4) (2*x*cos(x^2/(x^2+1)))/(x^2+1)^2
```

Aquí debemos mencionar que el comando `ratsimp` está pensado para la simplificación racional de polinomios (*rational simplication*), incluso si estos están en el argumento de una función no polinómica como en el caso del ejemplo anterior. Más adelante en el apartado [4.1.3](#) volveremos a hablar de este comando.

Continuaremos nuestra discusión de los conceptos del cálculo con la integración, tanto definida como indefinida. Para ello MAXIMA cuenta con el comando `integrate(f(x),x)` que calcula una primitiva de la función $f(x)$

```
(%i5) kill(all)$
(%i1) integrate(sin(2*x), x);
(%o1) -cos(2*x)/2
```

En muchas ocasiones `integrate(f(x),x)` no *funciona* como es el caso de las siguientes órdenes

```
(%i2) integrate(exp(x^2)*atan(x),x);
(%o2) integrate(%e^x^2*atan(x),x)
(%i3) integrate(x^2/(%e^x^2+1),x);
(%o3) integrate(x^2/(%e^x^2+1),x)
```

donde vemos que la respuesta coincide con la entrada.

Para calcular la integral definida de la función $f(x)$ en la variable x , e.g. $\int_a^b f(x) dx$, hay que usar la orden `integrate(f(x), x, a, b)` que intenta calcular de forma analítica la integral de la función $f(x)$,

```
(%i4) integrate(sin(2*x), x,0,%pi/2);
(%o4) 1
```

algo que en contadas ocasiones es posible en la práctica

```
(%i5) integrate(x^2/(%e^x^2+1),x,0,1);
(%o5) integrate(x^2/(%e^x^2+1),x,0,1)
(%i6) integrate(exp(-x^2),x,0,1);
(%o6) (sqrt(%pi)*erf(1))/2
```

Nótese que en la segunda hay una salida un poco *rara*. De hecho si le pedimos que evalúe numéricamente dicha salida obtenemos un valor numérico

```
(%i7) float(%);
(%o7) 0.7468241328124269
```

Si le preguntamos a MAXIMA sobre la función `erf` su respuesta es:

```
(%i8) ??erf;
0: erf  (Error Function)
...
6: nextlayerfactor  (Package facexp)
Enter space-separated numbers, 'all' or 'none': 0;
-- Function: erf (<z>)
The Error Function erf(z) (A&S 7.1.1)
See also flag 'erfflag'.
(%o8) true
```

Es decir, la respuesta la da en términos de la función de error $\text{erf}(z)$ (A&S 7.1.1) donde la referencia entre paréntesis se refiere al apartado 7.1.1. del manual clásico [2]. En los apartados §19.1 y §19.2 del manual [1] se pueden encontrar más comandos útiles para la integración analítica de funciones.

Cuando MAXIMA no puede calcular la integral analíticamente podemos calcularla numéricamente. Dado que dedicaremos un apartado especial a la integración (véase los apartados 8.2 y 8.3) vamos simplemente a mencionar aquí que MAXIMA cuenta con el paquete quadpack que tiene implementado una serie de reglas de cuadratura (véase también el comando `romberg`). Este paquete tiene muchísimas opciones así que nos restringiremos al comando `quad_qag`. que permite calcular el valor de la integral de una función en un intervalo acotado `quad_qag`. Su sintaxis es

```
quad_qag (f(x), x, a, b, key, [epsrel, epsabs, limit])
```

donde $f(x)$ es la función a integrar, x es la variable, a y b los extremos del intervalo. el valor `key` es un número del 1 al 6 (es el orden de la regla de integración de Gauss-Kronrod que se va a usar). La segunda lista consta de los siguientes elementos `epsrel` que es el error relativo deseado de la aproximación (el valor por defecto es 10^{-8} , `epsabs` es el error absoluto deseado de la aproximación (el valor por defecto es 0, y `limit` es el número máximo de subintervalos a utilizar (el valor por defecto es 200). La función `quad_qag` devuelve una lista de cuatro elementos: [la aproximación a la integral, el error absoluto estimado de la aproximación, el número de evaluaciones del integrando, un código de error]. Es deseable que la salida del último argumento (el código de error) sea 0 (no hay errores).

Por ejemplo, para calcular numéricamente la integral $\int_0^1 x^2 dx$ escribimos

```
(%i9) quad_qag (x^(2), x, 0, 1, 3, 'epsrel=1d-10);
(%o9) [0.333333333333333, 3.70074341541719*10^-15, 31, 0]
```

El primer elemento de la salida es casi el valor real $1/3$ de la integral con una precisión de 10^{-15} (como nos indica el segundo elemento) y el programa ha necesitado 31 iteraciones y a culminado sin errores.

Lo mismo podemos hacer con las dos integrales que vimos antes. Por ejemplo, para la que nos dio como salida la función de error tenemos

```
(%i10) quad_qag (exp(-x^2), x, 0, 1, 3, 'epsrel=1d-10);
(%o10) [0.746824132812427, 8.291413475940725*10^-15, 61, 0]
```

mientras que para la otra

```
(%i11) quad_qag (x^2/(%e^x^2+1), x, 0, 1, 3, 'epsrel=1d-10);
(%o11) [0.1188326884808158, 1.319307868295103*10^-15, 61, 0]
```

Más adelante, como ya mencionamos, volveremos a discutir sobre este tema, no obstante el lector interesado puede consultar en los apartados §19.3 y 19.4 del manual de MAXIMA [1] una descripción del paquete QUADPACK que contiene distintas funciones para el cálculo numérico de integrales de una variable.

2.3.3. Definiendo funciones con MAXIMA.

Vamos ahora a aprender como definir funciones a partir de las salidas de MAXIMA. Por ejemplo, definamos la función `int(x)` como la salida de la orden `integrate`

```
(%o12) kill(all)$
(%i1) int(x):=integrate(x+2/(x-3),x);
(%o1) int(x):=integrate(x+2/(x-3),x)
(%i2) int(x);
(%o2) 2*log(x-3)+x^2/2
```

que luego podemos usar para representarla gráficamente:

```
(%i3) wxplot2d([int('x)], [x,2,7])$ 
      plot2d: expression evaluates to non-numeric value somewhere
      in plotting range.
(%t3) << Graphics >>
```

Nótese que MAXIMA nos previene que hay valores para los cuales la función `int(x)` no está definida. Además podemos comprobar que la derivada de `int(x)` es precisamente nuestra función de partida:

```
(%i4) diff(int(x),x,1);
(%o4) x+2/(x-3)
```

Hay que ser cuidadoso a la hora de trabajar con las funciones definidas a partir de las salidas de MAXIMA. El siguiente ejemplo muestra que *no siempre* MAXIMA *funciona bien*:

```
(%i5) vnum1:integrate(x+2/(x -3), x, 0,1);
(%o5) -2*log(3)+2*log(2)+1/2
(%i6) int(1)-int(0);
Attempt to integrate wrt a number: 1
#0: int(x=1)
-- an error. To debug this try: debugmode(true);
(%i7) int(1);
Attempt to integrate wrt a number: 1
#0: int(x=1)
-- an error. To debug this try: debugmode(true);
```

¿Qué es lo que está ocurriendo? Como se ve, cuando definimos la función `int(x)` mediante `:=` en la entrada `%i8`, su salida es exactamente la misma que la entrada. La razón es que cuando usamos la definición `:=` lo que hace MAXIMA es asignarle a `int(x)` la orden correspondiente pero sin ejecutarla y sólo lo hace cuando se intenta evaluar la función. Este tipo de definiciones se les suele llamar *definiciones o asignaciones diferidas*. Así, al escribir `int(1)` MAXIMA intenta sustituir la `x` por 1 en la expresión `integrate(x+2/(x-3),x,0,1)`, lo que causa el error ya que `x` es una variable de integración que no podemos cambiar por 1, ya que no tiene sentido la operación.

Veamos otra forma diferente de proceder: la *definición o asignación inmediata* que lo que hace es realizar todas las operaciones indicadas y asignar el resultado final a nuestra función. Para este tipo de definiciones MAXIMA usa el comando `define` cuya sintaxis es⁸

```
define (f( x_1, ... , x_n), expr)
```

siendo `f` la función a definir cuyas variables son `x_1, ..., x_n` y `expr` es la expresión que define nuestra función. Así, por ejemplo, podemos definir la siguiente función:⁹:

```
(%i8) define(intprim(x),integrate(x+2/(x-3),x));
(%o8) intprim(x):=2*log(x-3)+x^2/2
```

lo que nos permite, efectivamente, usar la salida de MAXIMA para definir la nueva función `intprim`, que ahora no nos produce ningún error al evaluarla en 1.

```
(%i9) intprim(1);
(%o9) 2*log(-2)+1/2
(%i10) intprim(1)-intprim(0);
(%o10) 2*log(-2)-2*log(-3)+1/2
(%i11) %-vnum1;
(%o11) 2*log(3)-2*log(2)+2*log(-2)-2*log(-3)
(%i12) rectform(%);
(%o12) 0
```

Hemos de destacar que en la expresión anterior aparece `log(-2)` y `log(-3)` lo que en principio no tiene sentido ya que el logaritmo no está definido para los negativos a no ser que ... sea el logaritmo complejo,

⁸Esta forma de definir una función junto con la anteriormente usada mencionada `:=` se denominan en MAXIMA *funciones ordinarias*.

⁹También valdría `define(intprim(x),int(x))`

```
(%i13) log(-2); rectform(%);
(%o13) log(-2)
(%o14) log(2)+%i*pi
```

que como vemos es precisamente el caso. Además define también permite usar las funciones que hemos definido antes

```
(%i15) define(fun(x),int(x)+sin(x));
(%o15) fun(x):=sin(x)+x^2/2+2*log(x-3)
(%i16) fun(1);
(%o16) sin(1)+2*log(-2)+1/2
```

Está claro que las dos formas anteriores de definir funciones son muy distintos por lo que tenemos que ser cuidadosos a la hora de elegir el modo de hacerlo. Suele dar menos problemas la asignación inmediata pero puede haber situaciones donde sea conveniente utilizar la forma diferida.

Como mencionamos antes, una opción muy interesante que tiene MAXIMA es que se pueden hacer suposiciones sobre las variables (ver página 16). Por ejemplo si pedimos a MAXIMA que calcule la integral

```
(%i17) kill(all)$
(%i1) integrate (x^a*e^(-x), x, 0, inf);
"Is "a+1" positive, negative or zero?"p;
(%o1) gamma(a+1)
```

este nos pregunta si la a es positiva, negativa, etc. Al responder nos da la solución (¿preguntar a MAXIMA qué significa gamma?). Si sabemos de antemano que, por ejemplo, la a es positiva, etc. podemos usar el comando assume, concretamente en este caso vamos a asumir que $a > -1$:

```
(%i2) assume (a > -1)$
      integrate (x^a*exp(-x), x, 0, inf);
(%o3) gamma(a+1)
```

que le dirá a MAXIMA que la variable a siempre es mayor que -1 a lo largo de la sesión lo que nos permitirá calcular la integral directamente. Si no sabemos que significa la salida $\text{gamma}(a+1)$ podemos preguntar a MAXIMA que nos dará como respuesta que es la función gamma definida en el apartado 6.1.1 de [2]

```
(%i4) ??gamma;
```

Finalizaremos este apartado con algunos otros ejemplos sencillos

```
(%i5) assume (b > 1)$
integrate (x^(a)*(1-x)^(b), x, 0,1);
(%o6) beta(a+1, b+1)
```

¿Qué significa esta salida?

```
(%i7) integrate (x**(1/2)/(x+1)**(5/2), x, 0, inf);
(%o7) 2/3
(%i8) integrate (x**(2/3)/(x+1)**(3), x, 0, inf);
(%o8) (2*%pi)/3^(5/2)
```

Pero si intentamos la integral

```
(%i9) integrate (x**a/(x+1)**(5/2), x, 0, inf);
"Is "a" an "integer"?n;
"Is "2*a-1" positive, negative or zero?p;
(%o9) integrate(x^a/(x+1)^(5/2),x,0,inf)
```

nos hace una pregunta curiosa pues MAXIMA *sabe* que $a > 1$

```
(%i10) facts();
(%o10) [a>1,b>1]
```

sin embargo lo vuelve a preguntar.

2.3.4. Aproximando funciones.

Veamos otros dos comandos interesantes relacionados con el cálculo diferencial. El primero es el comando `taylor` que permite calcular la serie de Taylor de orden n alrededor del punto x_0 de la función $f(x)$. Formalmente MAXIMA lo que hace es calcular el polinomio

$$P_n(x, x_0) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(x_0)(x - x_0)^2 + \cdots + \frac{1}{n!}f^{(n)}(x_0)(x - x_0)^n + \cdots$$

es decir, el comando `taylor(f(x), x, x0, n)` devuelve la serie de Taylor alrededor del punto x_0 truncada hasta el orden n . Por ejemplo

```
(%i1) taylor(%e^x,x,0,5);
(%o1)/T/ 1+x+x^2/2+x^3/6+x^4/24+x^5/120+...
```

Si queremos el polinomio de Taylor (o sea, eliminar formalmente los ...) hay que usar el comando `taytorat(expr)` que devuelve la forma racional de la expresión `expr`.

Hay que destacar que MAXIMA puede trabajar formalmente con series de potencias infinitas. Para ello cuenta con el comando `powerseries(f(x), x, x0)` que devuelve la serie de potencias de f alrededor del punto x_0 .

```
(%i2) powerseries(%e^x,x,0);
(%o2) sum(x^i1/i1!,i1,0,inf)
(%i3) sere: niceindices(%);
(%o3) sum(x^i/i!,i,0,inf)
```

Lo interesante es que MAXIMA es capaz de integrar y diferenciar las series de potencias

```
(%i4) ser:niceindices(powerseries(1/(1+x),x,0));
(%o4) sum((-1)^i*x^i,i,0,inf)
(%i5) integrate(ser,x);
(%o5) sum((-1)^i*x^(i+1))/(i+1),i,0,inf)
(%i6) diff(%,,x);
(%o6) sum((-1)^i*x^i,i,0,inf)
```

Otro comando relacionado con las series de potencias es `pade`. Su sintaxis es:

```
pade(taylor_series, grado_numer, grado_denom)
```

Su salida es una lista con todas las funciones racionales $p_m(x)/q_n(x)$ (p_m y q_n polinomios de grado a lo más m y n , respectivamente) que tienen el mismo desarrollo de Taylor $a_0 + a_1x + \dots + a_Nx^N + \dots$ dado y en las que la suma de los grados del numerador m (`grado_numer`) y denominador n (`grado_denom`) es menor o igual que el orden N de truncamiento de la serie de potencias. Los parámetros `grado_numer` y `grado_denom` son los grados máximos que pueden tener los polinomios numerador p_m y denominador q_n , respectivamente. Además, en el caso cuando la suma de los grados del numerador y denominador de la fracción coincide con el orden de truncamiento de la serie ($n + m = N$) la salida coincide con el aproximante de Padé de la función, normalmente denotados por $[m, n]$.

Comenzamos con el ejemplo de una serie de potencias

```
(%i7) kill(all)$
(%i1) ser: taylor(1+2*x+3*x^2+4*x^3+5*x^4, x, 0, 4);
(%o1) 1+2*x+3*x^2+4*x^3+5*x^4+...
(%i2) pade(ser,1,1);
(%o2) []
```

La lista nula [] de la salida implica que no existe ninguna función racional del tipo $p_1(x)/q_1(x)$ con el mismo desarrollo que el dado, así que aumentaremos el grado de los polinomios p_m y q_m :

```
(%i3) pade(ser,2,2);
(%o3) [1/(x^2-2*x+1)]
(%i4) pade(ser,3,1);
(%o4) [-(x^3+2*x^2+3*x+4)/(5*x-4)]
(%i5) pade(ser,1,3);
(%o5) [1/(x^2-2*x+1)]
```

Si $n + m > N$ podemos tener más de una solución

```
(%i6) pa:pade(ser,4,4);
(%o6) [1/(x^2-2*x+1), -(x^3+2*x^2+3*x+4)/(5*x-4), 5*x^4+4*x^3+3*x^2+2*x+1]
(%i7) wxplot2d([ser,pa[1],pa[2]], [x,-1.2,1.2], [y,0,5],
               [legend,"taylor","pade1","pade2"])$
(%t7) (Graphics)
```

Nótese que en este último caso, entre las tres salidas de MAXIMA, tenemos un polinomio que coincide con la serie hasta el orden de truncamiento de la misma. La gráfica la podemos ver en la figura 2.5 (izquierda).

Veamos ahora la aproximación de funciones. Comenzamos con la función $\cos(x)$

```
(%i8) ser: taylor(cos(x), x, 0, 8);
      pa:pade(ser,4,4);
(%o8) 1-x^2/2+x^4/24-x^6/720+x^8/40320+...
(%o9) [(313*x^4-6900*x^2+15120)/(13*x^4+660*x^2+15120)]
(%i10) wxplot2d([cos(x),pa[1]], [x,-%pi,%pi], [y,-1.1,1.1],
                [legend,"cos(x)","pade"])$
(%t10) (Graphics)
```

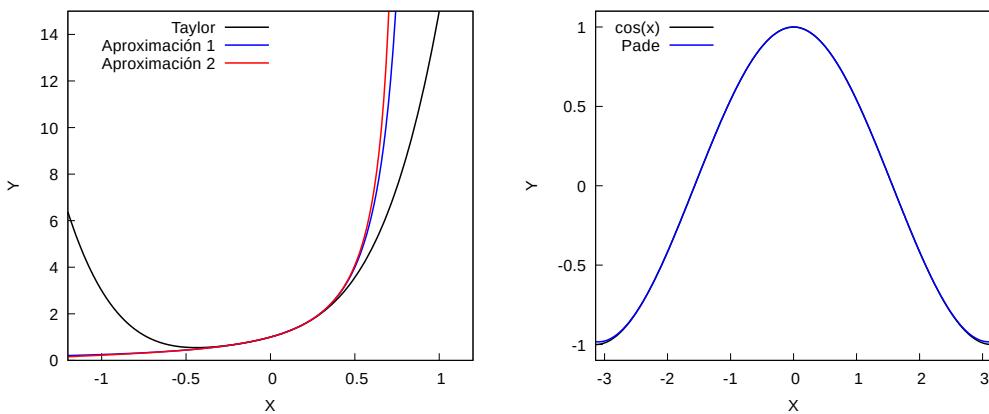


Figura 2.5: Aproximantes racionales de la serie $1+2x+3x^2+4x^3+5x^4+\dots$ (izquierda) correspondientes a la salida (%o6) y aproximante de Padé [4, 4] del $\cos(x)$ (derecha).

En este caso como hemos elegido $n+m=N$ tenemos el aproximante de Padé [4, 4] del $\cos(x)$. La gráfica la está representada en la figura 2.5 (derecha) y como se ve ambas curvas son prácticamente idénticas en el intervalo $[-\pi, \pi]$.

Como ejercicio encontrar los aproximantes de la función $\sin(x)$ con 8 términos en su serie de Taylor. Para este caso, encontrar el aproximante de Padé [4, 4]. Mostrar que ocurre si calculamos el aproximante con $m=n=5$. Representar gráficamente los resultados.

Pasemos a estudiar el caso de la función exponencial $f(x) = e^x$.

```
(%i11) kill(all)$
(%i1) f(x):=exp(x);
      ser:taylor(f(x),x,0,4);
      pa:pade(ser,2,2);
      plot2d([f(x),pa[1]], [x,-2,2], [y,0,8], [legend,"exp(x)","pade"])$
      plot2d([f(x),pa[1]], [x,-2,4], [y,0,20], [legend,"exp(x)","pade"])$
(%o1) f(x):=exp(x)
(%o2) 1+x+x^2/2+x^3/6+x^4/24+...
(%o3) [(x^2+6*x+12)/(x^2-6*x+12)]
(%t4) (Graphics)
      plot2d: some values were clipped.
(%t5) (Graphics)
```

Como vemos de las gráficas (ver figura 2.6) tenemos una aproximación muy buena con el aproximante de Padé [2, 2] en el intervalo $[-2, 2]$, no así en $[-2, 4]$ donde se ve a simple vista que el aproximante se aleja bastante del valor de la función. Calculemos a continuación el error en norma L^2 en ambos intervalos mediante la fórmula $\int_a^b (e^x - r(x))^2 dx$, donde $r(x)$ es el correspondiente aproximante de Padé de la función exponencial. Para ello usamos la orden quad_qag que ya vimos antes que calculará numéricamente la integral:

```
(%i7) er02:quad_qag( (exp(x)-pa[1])^2,x,-2,2,3,'epsrel=1d-10);
      er04:quad_qag( (exp(x)-pa[1])^2,x,-2,4,3,'epsrel=1d-10);
(%o6) [0.02008264729862179,2.229621742635612*10^-16,61,0]
(%o7) [546.0358341386714,6.062215553311562*10^-12,61,0]
```

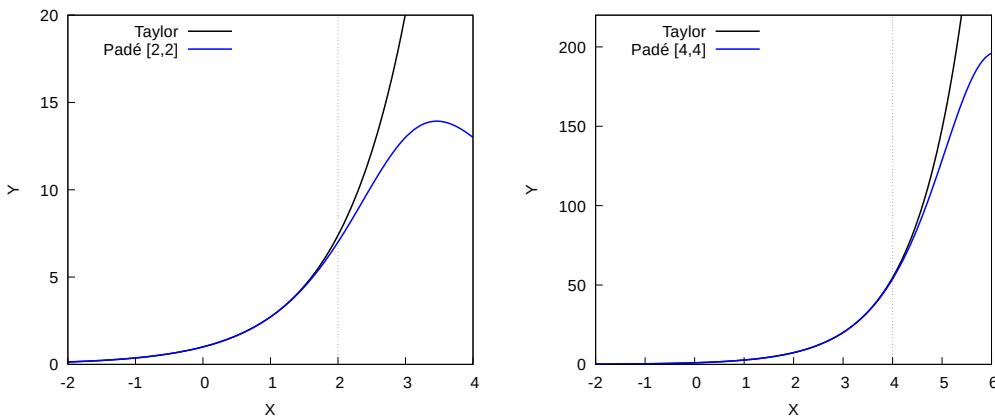


Figura 2.6: Aproximantes de Padé [2, 2] (izquierda) y [4, 4] (derecha) de la serie función $\exp(x)$.

Claramente el error de la segunda salida es inmenso.

Si aumentamos el orden del aproximante, e.g., calculando el [4, 4] la mejora es notable:

```
(%i8) f(x):=exp(x);
      ser:taylor(f(x),x,0,8);
      pa:pade(ser,4,4);
      wxplot2d([f(x),pa[1]], [x,-2,5], [y,0,50], [legend,"f(x)","pade"]);$ er02:quad_qag ((exp(x)-pa[1])^2,x,-2,2,3,'epsrel=1d-10);
(%o8) f(x):=exp(x)
(%o9) 1+x+x^2/2+x^3/6+x^4/24+x^5/120+x^6/720+x^7/5040+x^8/40320+...
(%o10) [(x^4+20*x^3+180*x^2+840*x+1680)/(x^4-20*x^3+180*x^2-840*x+1680)]
(%t11) (Graphics)
(%o12) [2.405645817987696*10^-9, 2.67080337622316*10^-23, 61, 0]
```

Aumentando el intervalo se puede apreciar mejor la diferencia con el caso [2, 2] considerado antes:

```
(%i13) wxplot2d([f(x),pa[1]], [x,-2,5], [y,0,50], [legend,"f(x)","pade"]);$ er04:quad_qag ((exp(x)-pa[1])^2,x,-2,4,3,'epsrel=1d-10);
(%t13) (Graphics)
(%o14) [0.1078519986905549, 1.197397721980963*10^-15, 61, 0]
```

Las gráficas se pueden ver en la figura 2.6.

Como último ejercicio de este apartado proponemos al lector que haga el estudio para la función $f(x) = 1/\sqrt{4 - x^2}$.

2.3.5. Definiendo funciones a trozos.

En MAXIMA también se pueden definir funciones a trozos. Esto se hace con los operadores lógicos. Por ejemplo, definamos la función

$$f(x) = \begin{cases} x^2 & \text{si } x \leq 0, \\ \sin(x) & \text{si } x > 0. \end{cases}$$

```
(%i15) kill(all)$
(%i1) f(x):=if x<=0 then x^2 else sin(x) $
(%i2) f(-1);f(0);f(1);
(%o2) 1
(%o3) 0
(%o4) sin(1)
```

Si tiene más trozos habrá que combinar más bloques `if-then-else`. Por ejemplo

$$g(x) = \begin{cases} x^2 & \text{si } x \leq 0, \\ \sin(x) & \text{si } 0 < x < \pi, \\ x^2 \sin(x) & \text{si } x \geq \pi. \end{cases}$$

se escribe

```
(%i5) g(x):=if x<=0 then x^2 else if x <= %pi then sin(x) else x^2*sin(x)$
(%i6) makelist(g(x),x,-3,3,1);
(%o6) [9,4,1,0,sin(1),sin(2),sin(3)]
```

A parte de estar definidas a trozos puede ocurrir que nuestra función tome ciertos valores en puntos concretos. Por ejemplo,

$$\text{sinc}(x) = \begin{cases} \frac{\sin x}{x} & \text{si } x \neq 0, \\ 1 & \text{si } x = 0. \end{cases}$$

Este caso requiere del operador lógico `equal(a,b)` que iguala a y b .

```
(%i7) sinc(x):= if equal(x,0) then 1 else sin(x)/x$
(%i8) sinc(-2);sinc(0);sinc(2);
(%o8) sin(2)/2
(%o9) 1
(%o10) sin(2)/2
```

Finalmente podemos combinarlas. Así la función

$$h(x) = \begin{cases} \cos(x) & \text{si } x < 0, \\ \pi & \text{si } x = 0, \\ \sin(x) & \text{si } 0 < x < \pi, \\ e^x & \text{si } x \geq \pi. \end{cases}$$

se define por

```
(%i11) h(x):=if equal(x,0) then %pi else if x<0 then cos(x) else
      if x <= %pi then sin(x) else exp(x)$
```

Para comprobarlo creamos la lista $(h(-3), h(-2), h(-1), h(0), h(1), h(2), h(3))$

```
(%i12) makelist(g(x),x,-3,5,1);
(%o12) [cos(3),cos(2),cos(1),%pi,sin(1),sin(2),sin(3),%e^4,%e^5]
```

2.3.6. Cálculo diferencial de funciones de varias variables.

El comando `diff` que ya discutimos en la página 22 también calcula las derivadas parciales de una función vectorial de varias variables. En este caso (en el ejemplo hemos tomado una función de dos variables) si queremos encontrar la k -ésima derivada parcial respecto a la variable x y la p -ésima respecto a la variable y de una función vectorial de dos variables escribimos

```
diff([ f1(x,y) , ... , fm(x,y) ] , x,k, y,p)
```

donde f_1, \dots, f_m son las coordenadas de la función vectorial f . Por otro lado, la orden `diff([f1(x,y) , ... , fm(x,y)])` calcula el diferencial de cada una de las componentes.

Como ejemplo consideremos la función $f(x,y) = (\sin(x)\sin(y), x^3y^3)$. Primero calculamos la derivada $\frac{\partial^2}{\partial x^2 \partial y^2}$ de cada componente y luego el de la función completa:

```
(%i13) kill(all)$
(%i1) diff([sin(x)*sin(y)],x,2,y,2);
(%o1) [sin(x)*sin(y)]
(%i2) diff([x^3*y^3],x,2,y,2);
(%o2) [36*x*y]
(%i3) diff([sin(x)*sin(y),x^3*y^3],x,2,y,2);
(%o3) [sin(x)*sin(y),36*x*y]
```

Comprobar, como ejercicio, la salida de la orden `diff([sin(x)*sin(y),x^3*y^3])`.

El diferencial de una función $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ viene dado por la matriz jacobiana

$$Df(a) = \begin{pmatrix} \frac{\partial f_1(a)}{\partial x_1} & \frac{\partial f_1(a)}{\partial x_2} & \cdots & \frac{\partial f_1(a)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(a)}{\partial x_1} & \frac{\partial f_m(a)}{\partial x_2} & \cdots & \frac{\partial f_m(a)}{\partial x_n} \end{pmatrix}.$$

Si queremos encontrar la matriz jacobiana $Df(a)$ com MAXIMA usamos el comando

```
jacobian([f1(x1,...,xn),...,fm(x1,...,xn)], [x1,...,xn])
```

donde f_1, \dots, f_m son las coordenadas de la función vectorial f y x_1, \dots, x_m , las correspondientes variables.

```
(%i4) jacobian([sin(x)*sin(y),x^3*y^3,x^2+y^2],[x,y]);
(%o4) [[cos(x)*sin(y), sin(x)*cos(y)],
 [3*x^2*y^3, 3*x^3*y^2],
 [2*x, 2*y]]
```

En el caso de funciones escalares de n variables, i.e., $f : \mathbb{R}^n \mapsto \mathbb{R}$, podemos calcular la matriz hessiana de f en el punto $a \in \mathbb{R}^n$

$$H_f(a) = \begin{pmatrix} \frac{\partial^2 f(a)}{\partial^2 x_1} & \cdots & \frac{\partial^2 f(a)}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(a)}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f(a)}{\partial^2 x_n} \end{pmatrix},$$

mediante la orden `hessian(f(x1, ..., xn), [x1, ..., xn])`. En el ejemplo siguiente mostramos como calcular la matriz hessiana de una función de dos variables:

```
(%i5) hessian(sin(x)*sin(y),[x,y]);
(%o5) [[-sin(x)*sin(y),cos(x)*cos(y)],[cos(x)*cos(y),-sin(x)*sin(y)]]
(%i6) hessian(x^3*y^3,[x,y]);
(%o6) [[6*x*y^3,9*x^2*y^2],[9*x^2*y^2,6*x^3*y]]
```

El comando `taylor` que vimos anteriormente también nos permite encontrar polinomios de Taylor para funciones de varias variables. En este caso la sintaxis es distinta

```
taylor(function , lista de variables , vector, orden)
```

donde `funcion` es la función de la que queremos desarrollar en las variables `lista de variables` (escritas como una lista) alrededor del punto definido por la lista `vector` y cada una de ellas hasta el orden definido por en número `orden`.

Por ejemplo¹⁰

```
(%i7) define(f(x,y),x*exp(y)+y*sin(2*x));
(%o7) f(x,y):=x*%e^y+sin(2*x)*y
(%i8) taylor (f(x,y) , [x, y] , [0,0] , 2)$ expand(%);
(%o9) 3*x*y+x
(%i10) taylor (f(x,y) , [x, y] , [0,0] , 5)$ expand(%);
(%o11) (x*y^4)/24+(x*y^3)/6+(x*y^2)/2-(4*x^3*y)/3+3*x*y+x
```

Hay otras variantes de este comando cuya salida no es exactamente el polinomio de Taylor del orden dado. Para más detalles véase el manual [1, pág. 478].

2.3.7. Cálculo vectorial.

A parte de lo anterior MAXIMA cuenta con un paquete específico para el cálculo vectorial: el paquete `vect`. Describamos brevemente como funciona dicho paquete. Lo primero que hay que destacar es que `vect` está pensado para trabajar en 2 o 3 dimensiones únicamente lo cual, en la mayoría de los casos prácticos, es suficiente. Para realizar cálculos con `vect` hay que combinar los nombres de las operaciones con las órdenes `express` y `ev(·, diff)`. El resto de operadores definidos en `vect` son: `grad` (gradiente), `div` (divergencia), `curl` (rotacional), `laplacian` (laplaciano) y `~` (producto vectorial).

Dados dos vectores $v = [a, b, c]$ y $w = [x, y, z]$ podemos calcular el producto escalar con la orden $v.w$, es decir usando como operador de multiplicación el “.”. Nótese que si usamos el operador multiplicación “*” obtenemos un resultado muy distinto (comprobarlo como ejercicio). Si queremos calcular el producto vectorial $v \times w$ hay que usar el operador “~”.

```
/* Cálculo vectorial */
```

¹⁰Aquí hemos usado `expand` para eliminar los puntos suspensivos. Esto en principio se debería poder hacer con la orden `taytorat` que mencionamos en el caso de una variable, pero por alguna razón hay versiones de MAXIMA que introducen elementos extraños. En este caso se puede usar la combinación equivalente `rat(ratdisrep(expr))`, o bien `expand` como hemos hecho en este ejemplo.

```
(%i12) kill(all)$
(%i1) load(vect)$
(%i2) [a, b, c] . [x, y, z];
(%o2) c*z+b*y+a*x
(%i3) [a, b, c] ~ [x, y, z];
(%o3) [a,b,c] ~ [x,y,z]
(%i4) express(%);
(%o4) [b*z-c*y, c*x-a*z, a*y-b*x]
```

Lo primero que notamos al escribir $[a, b, c] \sim [x, y, z]$ la salida es la misma. La razón es la forma en que está escrito dicho paquete vect por lo que hay que usar, como ya hemos dicho, la orden `express`.

Calculemos el gradiente de una función escalar $f(x, y, z)$, $\text{grad } f = \vec{\nabla}f = \left(\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz} \right)$

```
(%i5) arm:x^2+y^2-2*z^2;
(%o5) -2*z^2+y^2+x^2
(%i6) grad(arm);
(%o6) grad(-2*z^2+y^2+x^2)
(%i7) express(%);
(%o7) ['diff((-2*z^2+y^2+x^2),x,1), 'diff((-2*z^2+y^2+x^2),y,1),
      'diff((-2*z^2+y^2+x^2),z,1)]
(%i8) vv:ev(%,diff);
(%o8) [2*x,2*y,-4*z]
```

Aunque MAXIMA no ejecute inmediatamente la orden `grad` si que podemos operar con ella. Por ejemplo encontremos el producto escalar $\vec{\nabla}f \cdot \vec{\nabla}f$ para nuestra función

```
(%i9) express(grad(arm).grad(arm));
      ('diff((-2*z^2+y^2+x^2),z,1))^2+('diff((-2*z^2+y^2+x^2),y,1))^2+
      ('diff((-2*z^2+y^2+x^2),x,1))^2
      ev(%,diff);
(%o10) 16*z^2+4*y^2+4*x^2
```

Compruébese que ocurre si escribimos `grad(f).grad(f); express(%)`.

Calculemos ahora la divergencia de una función vectorial $\vec{v} = (v_x, v_y, v_z)$, $\text{div } \vec{v} = \vec{\nabla} \cdot \vec{v} = \frac{dv_x}{dx} + \frac{dv_y}{dy} + \frac{dv_z}{dz}$

```
(%i11) div(vv);
(%o11) div([2*x,2*y,-4*z])
(%i12) express (%); ev(%,diff);
(%o12) 'diff((-4*z),z,1)+'diff((2*y),y,1)+'diff((2*x),x,1)
(%o13) 0
```

Para calcular el rotacional de una función vectorial $\vec{v} = (v_x, v_y, v_z)$

$$\text{rot } \vec{v} = \vec{\nabla} \times \vec{v} = \left[\frac{dv_z}{dy} - \frac{dv_y}{dz}, \frac{dv_x}{dz} - \frac{dv_z}{dx}, \frac{dv_y}{dx} - \frac{dv_x}{dy} \right],$$

usamos la orden `curl`:

```
(%i14) curl(vv);
(%o14) curl([2*x, 2*y, -4*z])
(%i15) express (%); ev(%,diff);
(%o15) [’diff((-4*z),y,1)-’diff((2*y),z,1),
      ’diff((2*x),z,1)-’diff((-4*z),x,1),
      ’diff((2*y),x,1)-’diff((2*x),y,1)]
(%o16) [0,0,0]
```

El laplaciano de una función está definido por $\Delta f = (\vec{\nabla} \cdot \vec{\nabla})f = \operatorname{div}(\operatorname{grad} f) = \frac{d^2 f}{dz^2} + \frac{d^2 f}{dy^2} + \frac{d^2 f}{dx^2}$. Así

```
(%i17) laplacian(arm); express (%); ev(%,diff);
(%o17) laplacian(-2*z^2+y^2+x^2)
(%o18) ’diff((-2*z^2+y^2+x^2),z,2)+’diff((-2*z^2+y^2+x^2),y,2)+
      ’diff((-2*z^2+y^2+x^2),x,2)
(%o19) 0
```

El laplaciano se generaliza al caso de funciones vectoriales simplemente actuando por coordenadas así, si tenemos $\vec{v} = (v_x, v_y, v_z)$ $\Delta\vec{v} = (\Delta v_x, \Delta v_y, \Delta v_z)$

```
(%i20) laplacian(vv); express (%); ev(%,diff);
(%o20) laplacian([2*x, 2*y, -4*z])
(%o21) ’diff([2*x, 2*y, -4*z],z,2)+’diff([2*x, 2*y, -4*z],y,2)+
      ’diff([2*x, 2*y, -4*z],x,2)
(%o22) [0,0,0]
```

Calculemos ahora el laplaciano de la función escalar $f(x, y) = \log(x^2 + y^2)$

```
(%i23) arm2:log(x^2+y^2); laplacian(arm2); express (%);
      ev(%,diff); ratsimp(%);
(%o23) log(y^2+x^2)
(%o24) laplacian(log(y^2+x^2))
(%o25) ’diff(log(y^2+x^2),z,2)+’diff(log(y^2+x^2),y,2)+
      ’diff(log(y^2+x^2),x,2)
(%o26) 4/(y^2+x^2)-(4*y^2)/(y^2+x^2)^2-(4*x^2)/(y^2+x^2)^2
(%o27) 0
```

y repitamos en cálculo usando la combinación $\operatorname{div}(\operatorname{grad} f)$

```
(%i28) ev(express(grad(arm2)),diff);
(%o28) [(2*x)/(y^2+x^2),(2*y)/(y^2+x^2),0]
(%i29) div(%);
(%o29) div([(2*x)/(y^2+x^2),(2*y)/(y^2+x^2),0])
(%i30) express(%); ev(%,diff); ratsimp(%);
(%o30) ’diff((2*y)/(y^2+x^2),y,1)+’diff((2*x)/(y^2+x^2),x,1)
(%o31) 4/(y^2+x^2)-(4*y^2)/(y^2+x^2)^2-(4*x^2)/(y^2+x^2)^2
(%o32) 0
```

En general, podemos comprobar que $\Delta f = \operatorname{div}(\operatorname{grad} f)$, para ello definimos la dependencia de f usando la orden `depends`

```
(%i33) depends(f,[x,y,z]);
```

y realizamos las operaciones:

```
(%o33) [f(x,y,z)]  
(%i34) ev(express(grad(f)),diff);  
(%o34) [diff(f,x,1),diff(f,y,1),diff(f,z,1)]  
(%i35) div(%);  
(%o35) div([diff(f,x,1),diff(f,y,1),diff(f,z,1)])  
(%i36) divgra:express(%);  
(%o36) diff(f,z,2)+diff(f,y,2)+diff(f,x,2)  
(%i37) laplacian(f);  
(%o37) laplacian(f)  
(%i38) lap:express(%);  
(%o38) diff(f,z,2)+diff(f,y,2)+diff(f,x,2)  
(%i39) divgra-lap;  
(%o39) 0
```

Como ejercicio probar que $\Delta \vec{v} = \operatorname{div}(\operatorname{grad} \vec{v})$ cuando $\vec{v} = (v_x, v_y, v_z)$ es una función vectorial, i.e., $\vec{v}(x, y, z) = (v_x(x, y, z), v_y(x, y, z), v_z(x, y, z))$.

Introducción al MAXIMA CAS con algunas aplicaciones
Prof. Dr. Renato Álvarez Nodarse

Capítulo 3

Representaciones gráficas con MAXIMA.

Vamos a mostrar a continuación algunas de las posibilidades gráficas de MAXIMA. Un simple vistazo al manual de MAXIMA [1] (la versión en inglés suele estar más actualizada) nos permite descubrir un sinfín de opciones para los gráficos como, por ejemplo, eliminar ejes, controlar los títulos, los nombres de los ejes, dibujar las gráficas en coordenadas polares, etc. En el apartado 2.1 (página 9) hemos visto ejemplos sencillos de gráficas de funciones de una variable. Por completitud vamos a ver en este capítulo algunos ejemplos más sofisticados, no obstante se recomienda al lector que consulte el manual [1] para más información.

3.1. Las órdenes plot2d y plot3d.

Comenzaremos con algunos gráficos en dos dimensiones para luego pasar a tres dimensiones.

3.1.1. La orden plot2d y similares para gráficas en 2D.

Dado que en el apartado 2.1 vimos el caso más sencillo de la representación de funciones explícitas vamos a comenzar este apartado viendo como se pueden representar funciones implícitas. Para ello tenemos el comando `implicit_plot` del paquete `implicit_plot`. Veamos un ejemplo:

```
(%i1) load(implicit_plot)$  
(%i2) wximplicit_plot([x^2=y^3-3*y+1, x^2+y^2=1], [x,-4,4], [y,-4,4])$
```

La gráfica la podemos ver en la figura 3.1 (izquierda). Vamos ahora a cambiar algunas de las opciones por defecto que usa MAXIMA al usar `implicit_plot`, las cuales también son válidas para otros comandos similares como `plot2d`.

Por ejemplo, vamos a redefinir los nombres de las funciones usando la opción `legend`, vamos a incluir una red de líneas con la opción `grid2d` y, finalmente, usaremos la opción `same_xy` para obligar a MAXIMA a usar la misma escala para los dos ejes coordinados:

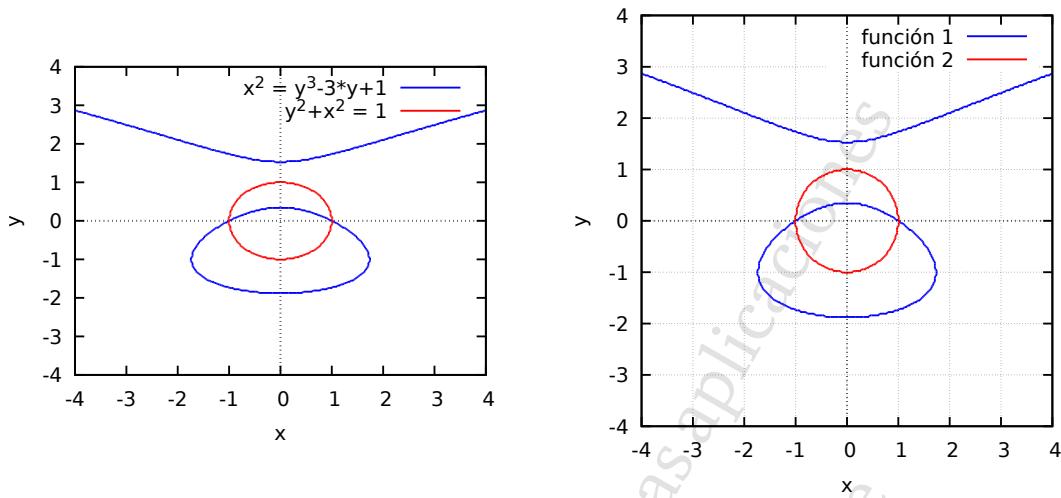


Figura 3.1: La gráfica de la función implícita definida por las ecuaciones $x^2 = y^3 - 3y + 1 = 0$ y $x^2 + y^2 = 1$ obtenidas en las salidas (%i5) (izquierda) y (%i6) —con otras opciones— (derecha).

```
(%i3) wximplicit_plot([x^2=y^3-3*y+1, x^2+y^2=1], [x,-4,4], [y,-4,4],
[style,[lines,3]],same_xy,[legend,"función 1","función 2"],grid2d)$
```

El resultado lo tenemos en la gráfica de la derecha de la figura 3.1.

También se pueden dibujar gráficos paramétricos, incluso junto a gráficos explícitos (ver la gráfica de la izquierda en la figura 3.2):

```
(%i4) wxplot2d([[parametric, cos(t), sin(t), [t,0,2*pi]],
[x, -sqrt(2), sqrt(2)], [y, -sqrt(2), sqrt(2)]]$)
(%i5) wxplot2d([[parametric, cos(t), sin(t), [t,0,2*pi]], -abs(x)],
[x, -sqrt(2), sqrt(2)], [y, -sqrt(2), sqrt(2)], same_xy]$)
```

Finalmente, dibujamos un gráfico *discreto*, o sea, a partir de una lista de datos (ver la gráfica de la derecha en la figura 3.2):

```
(%i6) N:60$
xx:makelist(2*pi*j/N,j,1,N)$
yy:makelist(sin(2*pi*j/N),j,1,N)$
zz:makelist(sin(2*pi*j/N)+(2*random(2)-1)/20,j,1,N)$
(%i10) plot2d([[discrete,xx,yy],[discrete,xx,zz]], [style,points,lines],
[point_type,diamond], [color,red,blue], [y,-1.1,1.1],grid2d,
[legend,"sin(x)","sin(x)+ruido"])$
```

Representemos la misma figura anterior pero incluyendo además la función $\sin(x)$ y defiendo un estilo distinto para cada una de las gráficas, por ejemplo la primera con círculos, la segunda con diamantes y la tercera con una linea. Así tenemos:

```
(%i11) wxplot2d([[discrete,xx,yy],[discrete,xx,zz],sin(x)], [x,0,2*pi],
[y,-1.1,1.1], [style,[points,3],[points,2],[lines,2]],
[point_type,bullet,diamond], [color,red,blue,black],
grid2d, [legend,"puntos del sin(x)","sin(x)+ruido","sin(x)"]);
```

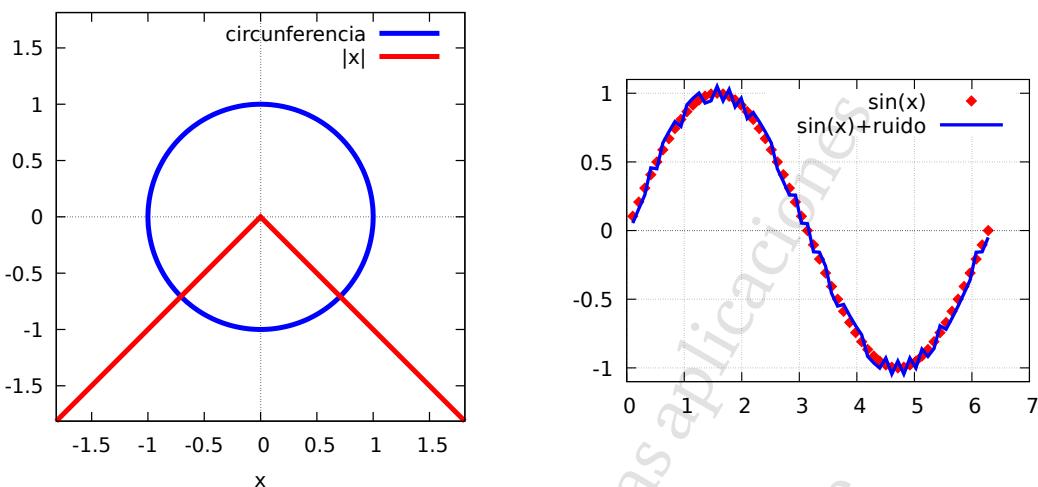


Figura 3.2: La gráfica paramétrica de una circunferencia junto a la función $|x|$ (izquierda) y una gráfica a partir de un conjunto discreto de puntos (derecha).

El resultado se puede ver en la figura 3.3.

Como último ejemplo vamos a representar la lemniscata de Bernoulli. Para ello usaremos las tres formas más usuales:

1. la expresión implícita definida por la ecuación $(x^2 + y^2)^2 = 2a^2(x^2 - y^2)$,
2. la expresión en coordenadas polares $r^2 = 2a^2 \cos(2\theta)$ y
3. la forma paramétrica $x = \frac{\sqrt{2}a \cos t}{\sin^2 t + 1}$ $y = \frac{\sqrt{2}a \cos t \sin t}{\sin^2 t + 1}$.

Así tenemos, respectivamente,

```
(%i12) kill(all)$
(%i1) load(implicit_plot)$
a:4$ 
wximplicit_plot((x^2+y^2)^2=2*a^2*(x^2-y^2),[x,-7,7],[y,-6,6],
same_xy,[legend,"implícita"],[ylabel,"y"] )$ 
(%t3)  (Graphics)
(%i4) wxplot2d(sqrt(2)*a*sqrt(cos(2*t)),[t,-7,7],[y,-7,7],[ylabel,"y"],
[xlabel,"x"],[gnuplot_preamble,"set polar"],same_xy,
[legend,"polares"] )$ 
(%t4)  (Graphics)
(%i5) wxplot2d([parametric,a*sqrt(2)*cos(t)/(1+sin(t)^2),
a*sqrt(2)*cos(t)*sin(t)/(1+sin(t)^2),[t,-%pi,%pi]],
same_xy,[ylabel,"y"],[xlabel,"x"],[y,-6,6])$ 
(%t5)  (Graphics)
```

Las salidas están representadas en las tres gráficas de la figura 3.4.

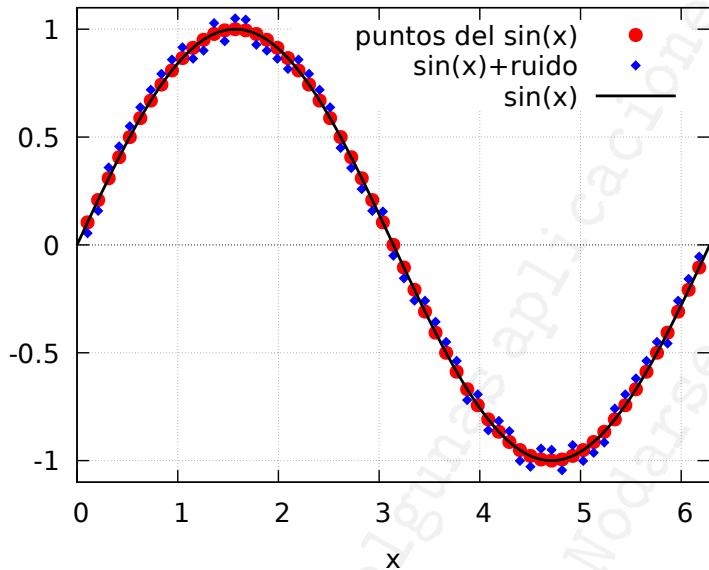


Figura 3.3: Gráfica del $\sin(x)$ a partir de puntos y con pequeñas modificaciones aleatorias usando distintos estilos.

3.1.2. Las ordenes `plot3d` y `contour_plot` para gráficas en 3D.

Cuando trabajamos con funciones escalares de dos variables es muy útil poder representarlas. MAXIMA también dibuja gráficas 3D con el comando `plot3d` cuya sintaxis es muy similar a la de `plot2d`

```
plot3d([funcion1,funcion2,...],[variable1,inicio,final],[variable2,inicio,final])

(%i6) kill(all)$
(%i1) f(x,y):= sin(x) + cos(y);
(%i2) wxplot3d(f(x,y), [x,-5,5], [y,-5,5])$
```

La gráfica 3D de la función $f(x, y) = \sin(x) + \cos(y)$ está representada en el gráfico de la figura 3.5. También podemos representar el gráfico de curvas de nivel, o dibujo de contorno, de $f(x, y)$ con el comando `contour_plot`

```
(%i3) wxcontour_plot(f(x,y), [x, -5, 5], [y, -6, 6], [legend,""], 
                     [gnuplot_preamble, "set cntrparam levels 20"])$
```

cuya salida la representamos en la gráfica de la izquierda de la figura 3.6. Como se ve en dicha figura, el problema que tiene la orden `contour_plot` es que la forma de representar las curvas de nivel de $f(x, y)$ no es muy elegante. Hay una forma más ilustrativa de representar un dibujo de contorno que consiste en hacer un gráfico 3D pero con opciones que equivalgan a mirarlo desde arriba. En nuestro ejemplo escribiríamos

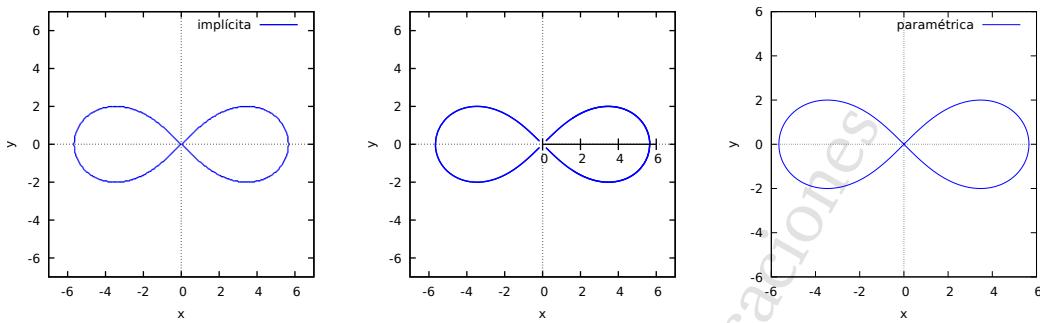


Figura 3.4: Las tres gráficas de la lemniscata de Bernoulli. De izquierda a derecha: forma implícita, en polares y paramétrica.

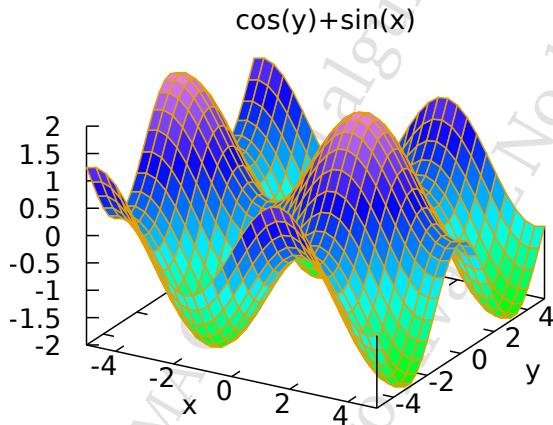


Figura 3.5: La gráfica 3D de la función $f(x, y) = \sin(x) + \cos(y)$.

```
(%i4) plot3d (f(x,y), [x,-4,4], [y,-4,4], [zlabel,""],  
[mesh_lines_color,false], [elevation,0], [azimuth,0],  
color_bar, [grid,80,80], [zticks,false], [color_bar_tics,1])$
```

donde [elevation,numx] controla el giro del eje z alrededor del x con un ángulo numx y [azimuth,numz] gira el plano xy obtenido alrededor del nuevo eje z un ángulo numz. La salida está representada en el gráfico de la derecha de la figura 3.6.

Antes de continuar vamos a mostrar como cambiar los colores de una misma figura. Como ejemplo vamos a representar la función definida paramétricamente de la siguiente forma: $x = \sqrt{u} \cos(v)$, $y = \sqrt{u} \sin(v)$ y $z = x^2 - y^2$:

```
(%i5) x1:sqrt(u)*cos(v); y1:sqrt(u)*sin(v); z:x1*x1-y1*y1;  
  
(%i6) wxplot3d([x1,y1,z],[u,0,3], [v,-%pi,%pi],[grid,50,50],  
[ xlabel,""], [ ylabel,""], [legend,"superficie"],  
[mesh_lines_color,false], [box,false], [elevation,110], [azimuth,23]);
```

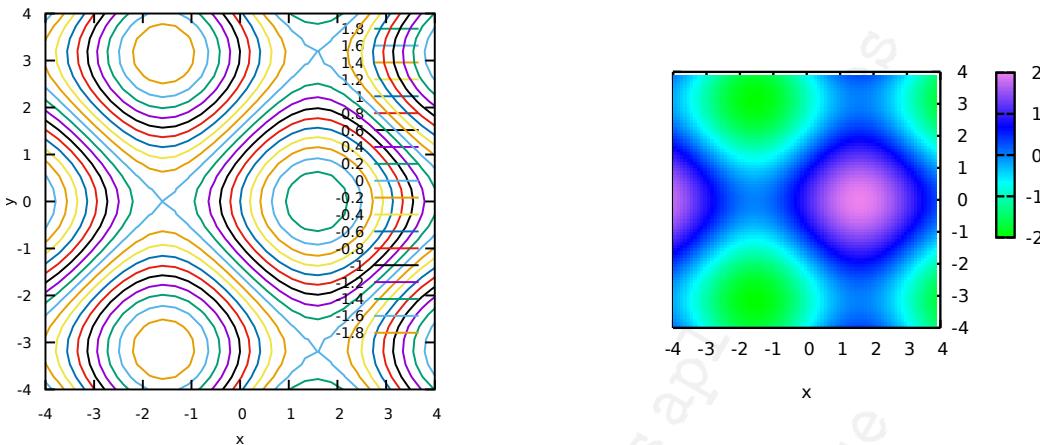


Figura 3.6: Gráficas de contorno de la función $f(x,y) = \sin(x) + \cos(y)$ usando el comando `contour` (izquierda) y `plot3d` (derecha).

La gráfica la vemos en la figura 3.7 (izquierda). Nótese que hemos eliminado el nombre de los ejes y las líneas de superficie. Vamos ahora a definir los colores de la superficie. Hay



Figura 3.7: La superficie definida por $x = \sqrt{u} \cos(v)$, $y = \sqrt{u} \sin(v)$ y $z = x^2 - y^2$ con las opciones por defecto para los colores (izquierda) y definiendo los colores a usar (centro y derecha).

dos formas de hacerlo, una usando los colores predefinidos `red`, `green`, `blue`, `magenta`, `cyan`, `yellow`, `orange`, `violet`, `brown`, `gray`, `black`, `white`, o bien definiéndolos en números sexagesimales, dos para cada una de las componentes RGB (rojo, verde, azul) precedidos por el símbolo `#`. Para ellos hay que definir la opción `palette` de la forma `[palette, [gradient, color1, ..., colorn]]` que define las tonalidades de color de la superficie siendo el `color1` el color de los valores más pequeños de z y `colorn` el de los más grandes (si se quieren ver los valores de las z hay que incluir la opción `color_bar`). Las dos posibles opciones las tenemos a continuación

```
(%i7) wxplot3d([x1,y1,z],[u,0,3],[v,-%pi,%pi],[grid,50,50],
              [xlabel,""],[ylabel,""],[legend,false],
              [mesh_lines_color,false],
              [palette,[gradient,magenta,blue,green]],
```

```
(%i8) wxplot3d([x1,y1,z],[u,0,3], [v,-%pi,%pi],[grid,50,50],
      [ xlabel,""], [ ylabel,""], [ legend, false ],
      [ mesh_lines_color, false ],
      [ palette,[gradient,"#dd0000","#b30000","#990033","#000000"] ],
      [ box, false ], [ elevation, 110 ], [ azimuth, 23 ]);
```

cuyas gráficas podéis ver en la figura 3.7 centro y derecha, respectivamente.

3.1.3. Exportando gráficas en pdf, eps, etc.

Otra opción muy útil de Máxima es la de exportar los gráficos como ficheros eps (postscript encapsulado), pdf o png, por citar algunos de los formatos más comunes. En el primer caso podemos usar la orden

```
(%i5) plot3d(f(x,y), [x,-5,5], [y,-5,5], [gnuplot_term, ps],
      [gnuplot_out_file, "/home/renato/maxima/graph3.eps"]);$
```

y en el segundo

```
(%i6) contour_plot (f(x,y), [x, -4, 4], [y, -4, 4],
      [gnuplot_preamble, "set cntrparam levels 20"],
      [gnuplot_term,"png size 1000,1000"],[gnuplot_out_file,"gra3c.png"]);$
```

que nos generan en el directorio /home/renato/maxima/, el primero, o en el directorio por defecto (en linux suele ser el directorio raíz del usuario) el segundo. Si queremos otros formatos como jpg, pdf, gif, animated_gif, etc., hay que usar las opciones de GNUPLOT o bien conversores de imágenes.

Por ejemplo, si queremos dibujar una rosa polar (que son curvas cuya ecuación en coordenadas polares es $r(\theta) = \cos(k\theta)$) podemos usar la orden

```
(%i7) plot2d( 3*(cos(2*t)),[t,-%pi,%pi],[gnuplot_preamble,"set polar"],
      [pdf_file, "rosapolar1.pdf"]);
```

que nos genera un fichero pdf con el gráfico en coordenadas polares de la misma en el directorio de trabajo de MAXIMA.

Lo anterior lo podemos combinar con las opciones de estos comandos para controlar los nombres de los ejes, títulos, etc. Es importante saber como controlar el tamaño y tipo de letra. Para ello hay que usar las opciones de GNUPLOT. A modo de ejemplo mostramos el caso de un gráfico en pdf.

```
(%i8) plot2d( 3*(cos(2*t)),[t,-%pi,%pi],[gnuplot_preamble,"set polar"],
      same_xy, [ylabel,"rosa polar"], [style,[lines,3]],
      [gnuplot_pdf_term_command,
      "set term pdfcairo color solid lw 3 size 17.2 cm, 12.9 cm font \"\",24\""],
      [xlabel, "r=3 cos(2 theta)"],
      [pdf_file, "/home/renato/rosapolar2.pdf"]);
```

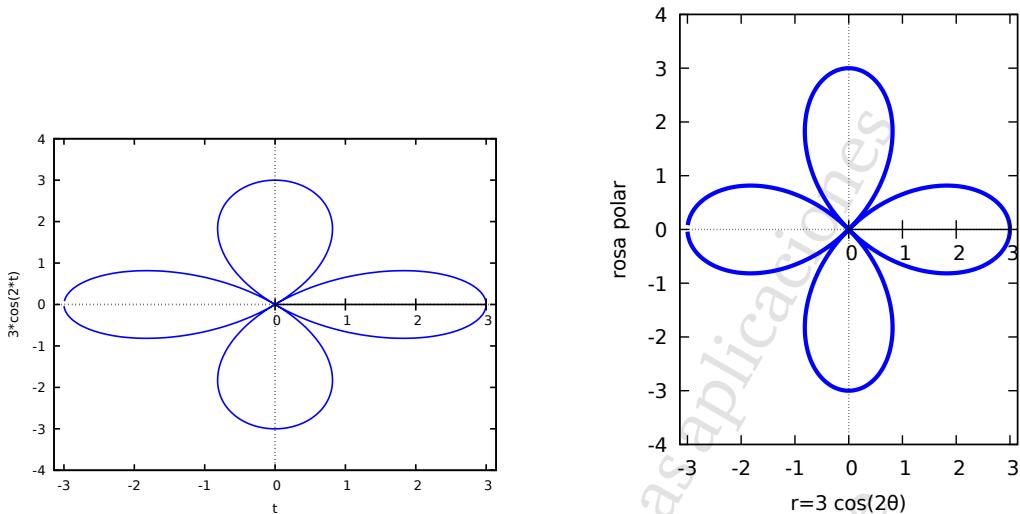


Figura 3.8: Gráficas de la rosa polar $r(\theta) = 3 \cos(2\theta)$ con las opciones por defecto (izquierda) y con algunas opciones de GNUPLOT (derecha).

Si queremos letras más grandes o más pequeñas cambiamos el último número 24 (`font`), `size` controla las dimensiones y `lw` (`line width`) el grosor de las líneas.

Los gráficos obtenidos por estos comandos se pueden ver en la figura 3.8 (ambos están representados fijando la anchura de los mismos de forma que se puedan ver las diferencias entre los mismos, como el tamaño de letra, el grosor de la curva, etc.).

La exportación de gráficos también funciona con los comandos `implicit_plot` y `plot3d`.

Para más información de las muchas opciones de los comandos `plot2d` y `plot3d` es recomendable consultar el manual [1].

3.2. El paquete `draw` para representaciones gráficas.

Antes de pasar a ver otros comandos de MAXIMA conviene tener en cuenta que además de las órdenes para representar funciones `plot2d` y `plot3d` discutidas anteriormente, MAXIMA cuenta también con un paquete específico para representaciones gráficas: el paquete `draw`¹. Dicho paquete es una potente interfaz para comunicar MAXIMA con Gnuplot y ha sido desarrollado por Mario Rodríguez Riotorto y es tremadamente versátil. Vamos a incluir algunos ejemplos sencillos de su uso sin dar una explicación exhaustiva de las opciones utilizadas ya que en casi todos los casos es fácil de intuir o adivinar la acción de cada una de ellas. Para más información sobre dicho paquete recomendamos al lector que consulte el manual [1] así como la web de su desarrollador

<http://www.tecnostats.net> o <http://riotorto.sourceforge.net>

Antes de comenzar a describir como funciona `draw` conviene aclarar que cuando se usa `Gnuplot`, y no sólo, MAXIMA precisa escribir una serie de archivos temporales. La variable

¹Conviene hacer notar que en actualizaciones de MAXIMA puede ocurrir que algunas de las opciones de `draw` dejen de funcionar. Ello podría estar relacionado con el hecho de que `draw` usa las librerías `pgplot` externas a MAXIMA.

que controla a donde van a parar dichos ficheros temporales es `maxima_tempdir`. Por defecto MAXIMA suele usar el directorio raíz del usuario pero se puede definir el camino que se quiera. Por ejemplo, `maxima_tempdir:"/home/renato/tmp/maxima"` define como directorio de ficheros temporales al directorio `/home/renato/tmp/maxima`.

3.2.1. Usando `draw3d` para gráficas 3D.

Comenzaremos con el comando `draw3d` para dibujar gráficos en tres dimensiones. Existen dos formas de invocar este comando: uno es dentro de la propia sesión `wxdraw3d` y la otra mediante una ventana exterior de `GNUPLOT` con `draw3d`.

Por ejemplo, dibujemos la función $f(x, y) = \sin(x^2 + y^2)$. Para ello usamos el código

```
(%i9) kill(all)$
(%i1) load(draw)$
(%i2) define(f(x,y),sin(x^2+y^2));
(%o2) f(x,y):=sin(y^2+x^2)
(%i3) wxdraw3d (font="Arial", font_size = 20, view=[83,25],
enhanced3d = false, xu_grid = 100, color = cyan,
explicit(f(x,y), x,-2,2,y,-2,2),
point_type=filled_circle, point_size = 3,color = black,
points([[0,0,0]]), color = red, points([[1,1,f(1,1)]]) )$
```

De la secuencia anterior quizá la única opción que puede provocar desconcierto es `view` cuyos argumentos (dos ángulos) determinan la posición del observador. Si en vez de la forma integrada `wxdraw3d` usamos la externa `draw3d`, podemos de forma análoga al caso de `plot2d` interactuar con la gráfica usando el ratón y cambiar la posición del observador, la cual la podemos ver en la esquina inferior izquierda (ver gráfica 3.9).

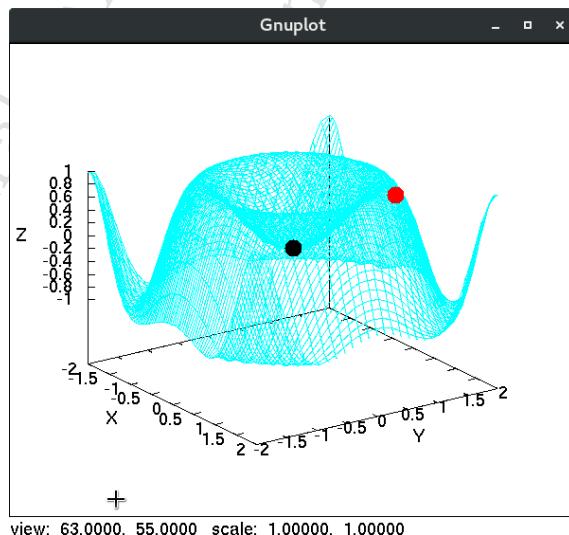


Figura 3.9: Ventana externa a `wxMAXIMA` de `GNUPLOT` para el gráfico de la salida (%i3). En la esquina inferior izquierda vemos la posición angular del observador.

Si queremos exportar la figura anterior como un fichero pdf tenemos que incluir las siguientes opciones dentro de la orden `draw`:

```
file_name="nombre del fichero", terminal=pdf, dimensions=[1500,1500]
```

que determinan el nombre del fichero (se puede incluir la dirección completa si se quiere, si no MAXIMA lo graba en el directorio por defecto), el tipo de fichero y las dimensiones en número de puntos. Así, por ejemplo, para producir la figura 3.10 escribimos

```
(%i4) draw3d(font="Arial", font_size=30, xtics=1, ytics=1, ztics=1/2,
enhanced3d = false, xu_grid = 100, color = cyan, view=[55,57],
explicit(f(x,y), x,-2,2,y,-2,2), point_type=filled_circle,
point_size = 2,color = black, points([[0,0,0]]),
color = red, points([[1,1,f(1,1)]]),dimensions = [1500,1500],
file_name = "draw3bf1", terminal = 'pdf);
```

En la orden anterior hay una opción que no estaba incluida en la entrada (%i3) y que es muy útil: `xtics=1, ytics=1, ztics=1/2`.

Lo que hace la opción `xtics` es controlar lo que aparece en el eje x (las otras dos controlan el resto de los ejes de coordenadas). La opción `xtics=1` indica que la distancia entre dos marcas consecutivas en el eje x es de 1 (en el caso del eje z es $1/2$). Se pueden definir por ejemplo los valores que queremos que aparezcan en los ejes enumerándolos entre llaves. Así por ejemplo, si incluimos las opciones `xtics={-2,-0.75,0.5,1.5}`, `ztics={-1,0,1}` en la entrada (%i3) nos dibujará la misma gráfica pero en el eje x aparecerán especificados los valores $-2, -0.75, 0.5, 1.5$ y el eje z los valores $-2, 0, 1$ a diferencia de los valores por defecto (ver la figura 3.9) o bien una lista de la forma `{"cadena", valor numérico}` que pone en la posición de `valor numérico` la expresión `cadena`. Por ejemplo, si agregamos a la entrada (%i3) la opción `ytics={[y=2",2],[y=0",0],[y=-2",-2]}` obtendremos una gráfica donde en las posiciones $-2, 0, 2$ del eje y aparecerán las expresiones $y = -2$, $y = 0$ e $y = 2$, respectivamente.

Otra forma de exportar el fichero consiste en escribir tras la orden `draw` la orden `draw_file(terminal = tipo, file_name="nombre")` donde `tipo` puede ser `jpg`, `png`, o `pdf`, por ejemplo.

```
(%i4) draw3d(font="Arial", font_size=30, xtics=1, ytics=1, ztics=1/2,
enhanced3d = false, xu_grid = 100, color = cyan, view=[55,57],
explicit(f(x,y), x,-2,2,y,-2,2), point_type=filled_circle,
point_size = 2,color = black, points([[0,0,0]]),
color = red, points([[1,1,f(1,1)]]),dimensions = [1500,1500])$  
draw_file(terminal = 'pdf, file_name = "draw3bf1")$
```

Otras opciones para la `terminal` son `jpg` o `png` que graba una figura en formato `jpg` o `png`, respectivamente. Si en vez de una rejilla queremos que MAXIMA dibuje una superficie coloreada usamos la opción `enhanced3d = true`. En la figura 3.10 se muestran dos salidas de `draw3d`, con las dos posibilidades para la opción `enhanced3d`. Si queremos eliminar la barra de colores que sale en la gráfica de la derecha de la figura 3.10 debemos usar la opción `colorbox=false`.

En el siguiente ejemplo vamos a dibujar la superficie $z = f(x, y)$ definida por la función $f(x, y)$ y el plano tangente a la misma en un punto dado (a, b) .

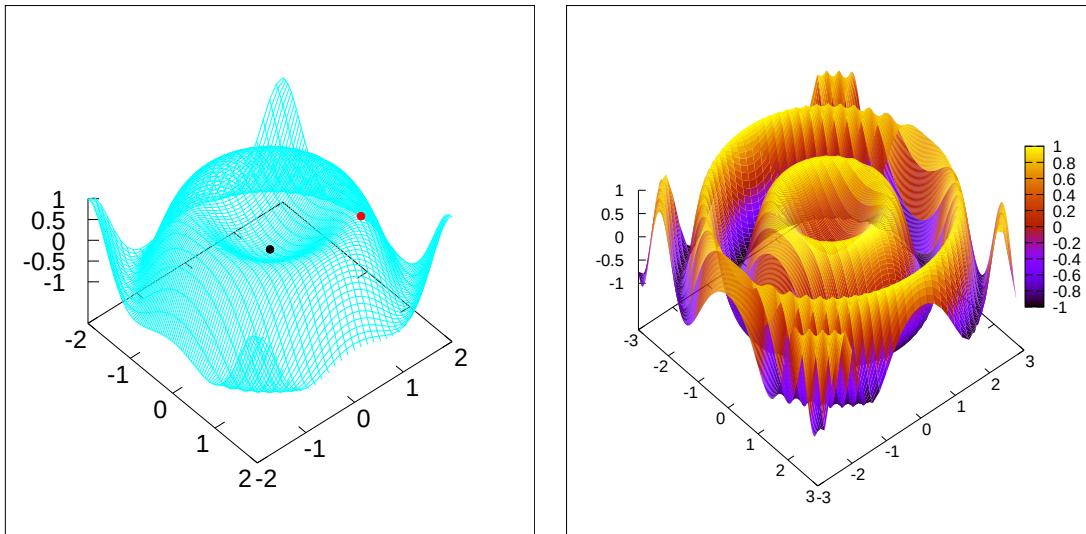


Figura 3.10: Los gráficos de la función $\sin(x^2 + y^2)$ con la opción `enhanced3d = false` (izquierda) y `enhanced3d = true` (derecha).

```
(%i5) define(f(x,y),exp(-x^2-y^2));
      define(dxf(x,y),diff(f(x,y),x));
      define(dyf(x,y),diff(f(x,y),y));
(%o5) f(x,y):=%e^-y^2-x^2
(%o6) dxf(x,y):=-2*x*%e^-y^2-x^2
(%o7) dyf(x,y):=-2*y*%e^-y^2-x^2
```

Para ello usaremos la ecuación del plano tangente en el punto (a, b) cuya expresión analítica es:

$$z - f(a, b) = \frac{\partial f}{\partial x}(a, b)(x - a) + \frac{\partial f}{\partial y}(a, b)(y - b),$$

```
(%i8) a:1/2$ b:1/2$
      define(z(x,y),f(a,b)+dxf(a,b)*(x-a)+dyf(a,b)*(y-b));
(%o10) z(x,y):=-(y-1/2)/sqrt(%e)-(x-1/2)/sqrt(%e)+1/sqrt(%e)
```

y luego lo dibujamos (ver figura 3.11 izquierda):

```
(%i11) draw3d( view=[57,115], color=blue,xu_grid = 100,
      explicit(f(x,y),x,-2,2,y,-2,2), color=cyan,
      parametric_surface(x,y,z(x,y),x,0,1,y,0,1),
      point_type=filled_circle, point_size = 2,color = black,
      points([[a,b,f(a,b)]]) )$
```

Dibujemos ahora el plano tangente a la esfera con centro en $(0, 0, 0)$ y radio 1 en el punto $(\sqrt{2}/2, 1/2, 1/2)$ con su correspondiente vector normal (ver figura 3.11 derecha).

```
(%i12) draw3d(view = [111, 40], xu_grid = 200,axis_3d = false,
      xtics=false,ytics=false,ztics=false,proportional_axes = xyz,
      parametric_surface(sin(x)*cos(t),sin(x)*sin(t),cos(x),
      t,0,2*pi,x,0,%pi/2),
```

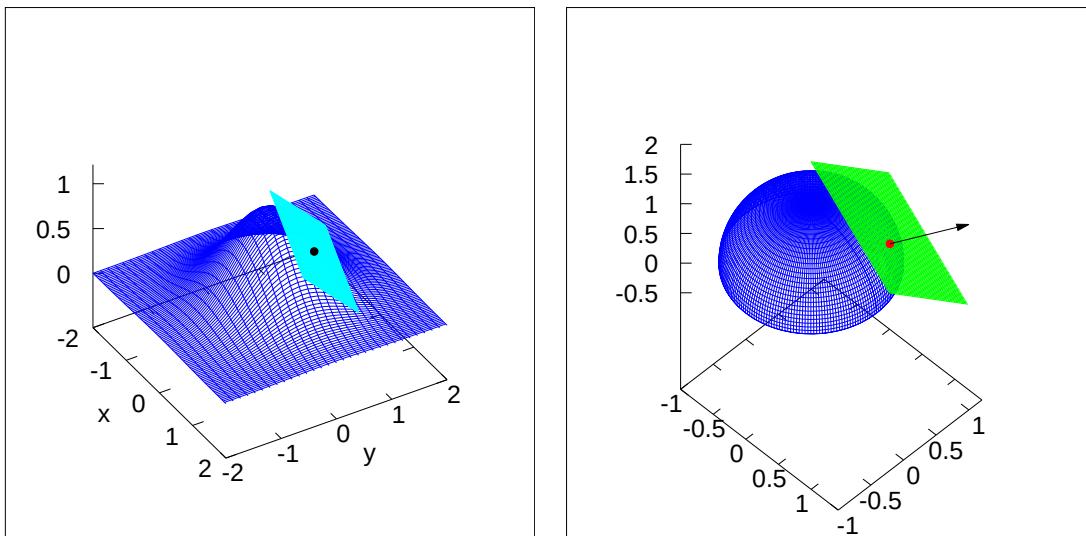


Figura 3.11: Los gráficos de las funciones $z = \exp(-x^2 - y^2)$ (izquierda) y $z = \sqrt{1 - x^2 - y^2}$ (derecha) con sus correspondientes planos tangentes en los puntos $(1/2, 1/2, \exp(-1/2))$ y $(\sqrt{2}/2, 1/2, 1/2)$, respectivamente.

```
color = green, enhanced3d = false, xu_grid = 100,
explicit(-(x-sqrt(2)/2)-(y-1/2)+1/2,x,0,1.2,y,0,1.2),
point_type=filled_circle, point_size = 1,color = red,
points([[sqrt(2)/2,1/2,1/2]]),
head_length = 0.1,head_angle = 15,color = black,
vector([sqrt(2)/2,1/2,1/2],[sqrt(2)/2,1/2,1/2]) )$
```

Consideremos ahora un ejemplo de una función definida implícitamente:

```
(%i13) draw3d(xu_grid=200, yv_grid=200, enhanced3d=[-x+y,x,y,z],
points_joined=true,colorbox=false,xlabel="x",ylabel="y",
zlabel="z",font="Arial", font_size=20, palette = color,
implicit(z^2+2*y^2+x^2=16, x,-3,3,y,-4,4,z,-4,2) )$
```

Si queremos representar el elipsoide completo entonces debemos escribir los intervalos de las variables de forma que se cubra toda la región. En este caso por ejemplo basta elegir $x \in [-4, 4]$, $y \in [-4, 4]$ y $z \in [-4, 4]$. Es decir:

```
(%i14) draw3d(xu_grid=200, yv_grid=200, enhanced3d=[-x+y,x,y,z],
nticks = 80, font="Arial", font_size=20, key="f(x,y)",
xlabel="x",ylabel="y",zlabel="z",points_joined=true,
colorbox = false, font="Arial", font_size=20, palette=color,
implicit(z^2+2*y^2+x^2=16, x,-4,4,y,-4,4,z,-4,4) ) $
```

Las gráficas obtenidas se pueden ver en la figura 3.12

Por ejemplo, si queremos dibujar la curva definida por la función $z = f(x, y) = (x + y)^2 + 1$ cuando las variables x e y están sujetas a la condición $x^2 + y^2 = 4$ podemos reescribir la función de forma paramétrica: $x = 2 \cos u$, $y = 2 \sin u$, $u \in [0, 2\pi)$ y escribir la siguiente secuencia:

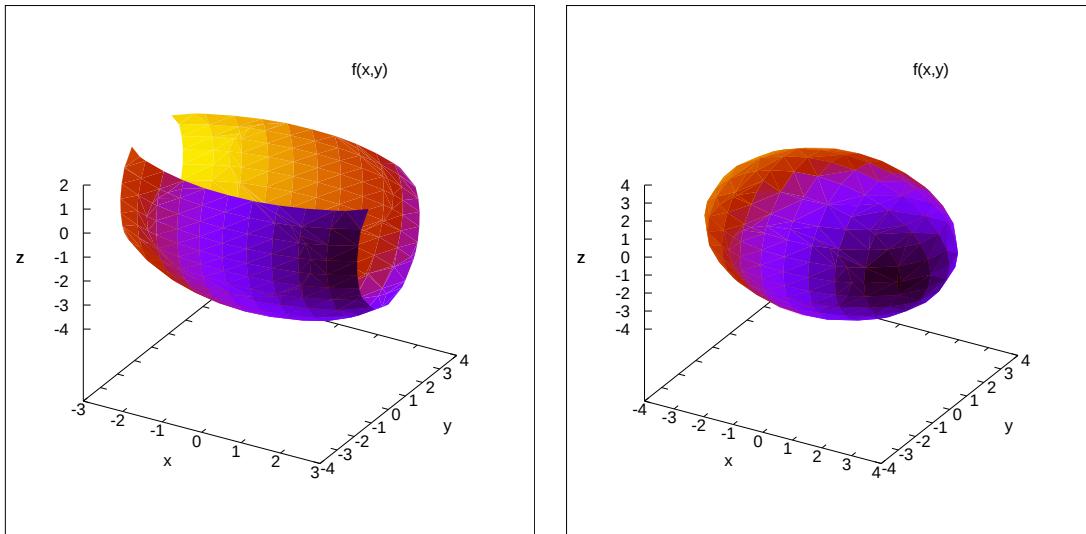


Figura 3.12: Los gráficos del elipsoide $x^2 + 2y^2 + z^2 = 16$ con $x \in [-3, 3]$, $y \in [-4, 4]$ y $z \in [-4, 2]$ (izquierda) y con $x \in [-4, 4]$, $y \in [-4, 4]$ y $z \in [-4, 4]$ (derecha).

```
(%i14) draw3d(view=[51,60], font="Arial", font_size = 20, key = "f(x,y)",
 xlabel = "x", ylabel = "y", zlabel = "z",
 color = magenta, nticks = 50, line_width = 3,
 parametric(2*cos(u),2*sin(u),(2*cos(u)+2*sin(u))^2+4,u,0,2*pi))$
```

Finalmente, dibujemos la figura de revolución alrededor del eje $0x$ de una función. Para ello usamos el comando `parametric_surface`

```
(%i15) draw3d(xu_grid = 50, ytics = 1/2, ztics=1/2,
 zrange=[-1,1], yv_grid = 25, font="Arial", font_size = 16,
 view = [65,15],enhanced3d = none ,colorbox = false,
 parametric_surface(x,x*cos(t),x*sin(t), x,-1,1, t,0,2*pi) )$
```

Ambos gráficos están representados en la figura 3.13.

3.2.2. Usando `draw2d` para gráficas 2D.

Veamos ahora algunos ejemplos del comando `draw2d` para gráficos en dos dimensiones.

```
(%i16) kill(all)$
(%i1) load(draw)$
(%i2) f(x):=(2*x)/(x^2+1)$
(%i3) wxdraw2d(explicit(f(x),x,-8,8));
(%i4) wxdraw2d(grid = true, axis_top = false, axis_right = false,
 yrange = [-1,1], key = "f(x)", line_width = 3,
 explicit(f(x),x,-8,8),dimensions = [800,600]);
```

El primero es la típica orden para dibujar el gráfico de una función $f(x)$ usando las opciones por defecto de `gnuplot`. En la segunda, hemos redefinido algunas de las opciones

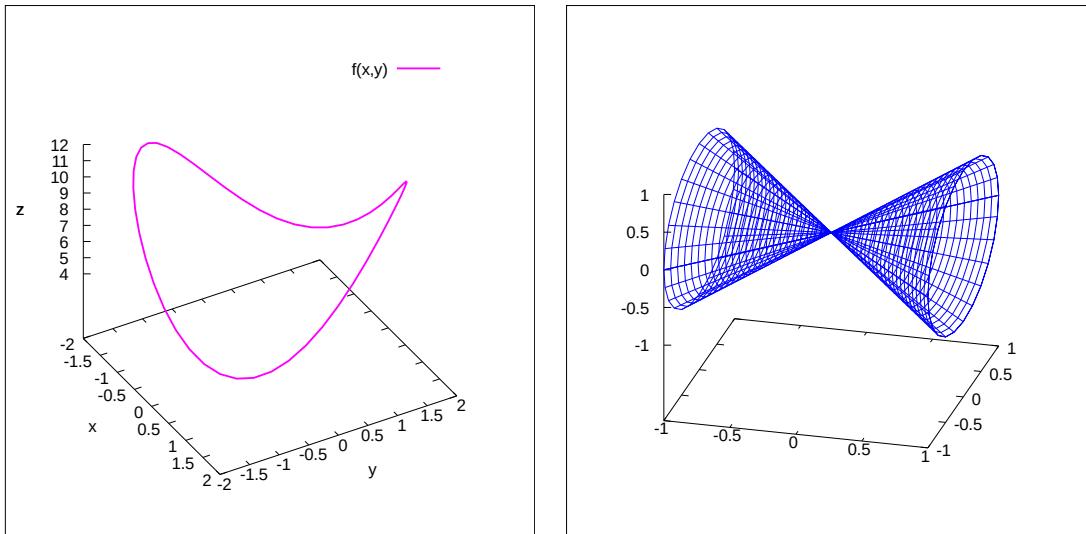


Figura 3.13: El gráfico de la función curva definida por $z = f(x, y) = (x + y)^2 + 1$ cuando las variables x e y están sujetas a la condición $x^2 + y^2 = 4$ (izquierda) y la figura de revolución definida por la función $f(x) = x$ (derecha).

de `gnuplot`. Como hemos dicho antes es conveniente leer la documentación de `draw` para más detalles sobre las mismas.

Con `draw2d` se pueden dibujar varias funciones en un mismo gráfico —ver figura 3.14 (izquierda)—

```
(%i5) draw2d( line_type=solid, line_width=4,
    yrange = [-1.2,1.2], xrange=[-.5,6.5],
    xaxis =true, xaxis_type=solid, yaxis =true, yaxis_type=solid,
    line_type = solid, color=green, explicit(sin(x/2),x,0,2*pi),
    line_type=dashes,color=dark_violet,explicit(sin(x),x,0,2*pi),
    line_type=short_dashes,color=navy,explicit(sin(3*x/2),x,0,2*pi),
    line_type=short_long_dashes,color=blue,explicit(sin(2*x),x,0,2*pi),
    line_type=dot_dash,color=red,explicit(sin(5/2*x),x,0,2*pi) )$
```

Un opción interesante que vemos son las distintas opciones de líneas, lo cual es conveniente si el gráfico se va a visualizar en blanco y negro. Dicha opción es `line_type` y las posibilidades son para todas los terminales `solid` y `dots`. También es posible usar las opciones `dashes`, `short_dashes`, `short_long_dashes`, `short_short_long_dashes` y `dot_dash`, pero no están disponibles en los terminales `png`, `jpg` y `gif`.

También se pueden mezclar distintos tipos de gráficos: explícitos, implícitos y paramétricos:

```
(%i6) draw2d(grid=true, line_type=solid, line_width=5, nticks=500,
    xaxis =true, xaxis_type=solid,
    yaxis =true, yaxis_type=solid,
    key = "implicita", color=blue,
    implicit(y^2=x^3-2*x+1, x, -4,5, y, -4,5),
    key = "explicita", color=red,
    explicit(x^(3/2),x,0,3), key = "", explicit(-x^(3/2),x,0,3),
```

```

key = "parametrica", color=green,
parametric(2*(cos(2*t)-cos(1*t)^3),2*(sin(2*t)-sin(1*t)^3),t,-%pi,%pi),
title = "gráficas explícita, implícita y paramétrica")$
```

Aquí `nticks` controla el número de puntos que se dibujan en dos dimensiones con las órdenes `explicit`, `parametric`, `polar` y en tres con `parametric`. La gráfica resultante se puede ver en la figura 3.14 (derecha).

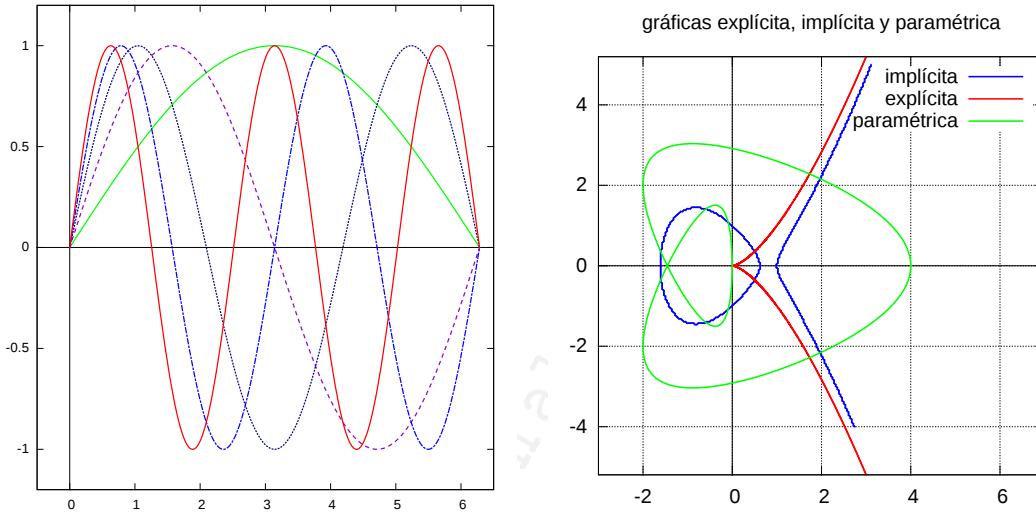


Figura 3.14: A la izquierda el gráfico de las funciones $\sin(kx/2)$, $k = 1, 2, 3, 4, 5$. A la derecha la función definida implícitamente $y^2 = x^3 - 2x + 1$, las definidas explícitamente $f_1(x) = x^{3/2}$, $f_2(x) = -x^{3/2}$ y la definida paramétricamente $g(t) = (x(t) = 2(\cos(2t) - \cos(t)^3), y(t) = 2(\sin(2t) - \sin(t)^3))$.

Podemos dibujar el área bajo la curva de una función dada. Por ejemplo, el área bajo la curva de la función $f(x)$ que definimos antes en el intervalo $[1, 5]$ —ver figura 3.15 (derecha)—

```
(%i17) draw2d(grid = true,dimensions = [1500,1000],
    axis_top=false, axis_right=false, yrangle=[-1,1],
    filled_func=true,fill_color="light-blue", explicit(f(x),x,1,5),
    filled_func=false,key ="f(x)",line_width=3,explicit(f(x),x,-8,8))$
```

o dibujar dos funciones y sombrear la región definida entre ellas²

Tomemos por ejemplo las funciones $f(x) = 2x^2 - 5x + 3$ y $g(x) = -8x^2 - x + 30$ y dibujemos el área entre ambas:

```
(%i18) f1(x):=2*x^2-5*x+3;
f2(x):=-8*x^2-x+30;
[x1,x2]: map('rhs,solve(f1(x)=f2(x),x));
(%o10) [-sqrt(274)-2/10,(sqrt(274)+2)/10]
(%i11) wxdraw2d(fill_color = grey,font="Arial", font_size = 16,
    filled_func = f2, line_width = 5, dimensions = [800,600],
```

²El comando `map` lo describiremos en detalle más adelante (página 103).

```
explicit(f1(x),x,x1,x2), filled_func = false,
xaxis = true, xtics_axis = true, yaxis = true,
line_width=3, key ="f(x)", color=red, explicit(f1(x),x,-3,3),
key = "g(x)", color = blue, explicit(f2(x),x,-3,3) )$
```

La gráfica resultante se puede ver en la figura 3.15 (izquierda).

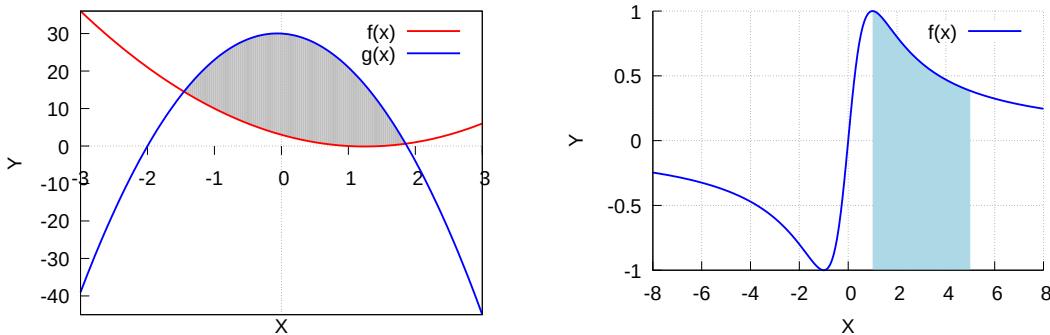


Figura 3.15: Región entre las curvas $f(x) = 2x^2 - 5x + 3$ y $g(x) = -8x^2 - x + 30$ (izquierda) y el área bajo la función $f(x) = (2x)/(x^2 + 1)$ en $[2, 5]$ (derecha).

Si queremos representar una región podemos usar la orden

```
region (expr,var1,minval1,maxval1,var2,minval2,maxval2)
```

que dibuja una región del plano definida por las desigualdades descritas en expr. Como ejemplo veamos la región entre las mismas dos funciones de antes $f(x) = 2x^2 - 5x + 3$ y $g(x) = -8x^2 - x + 30$ ³

```
(%i12) grasom1: gr2d(line_width = 3, xaxis=true, yaxis=true,
                      fill_color=gray,x voxel=30, y voxel=30,
                      region(y>f1(x) and y<f2(x),x,-3,3,y,-40,40))$
(%i13) draw(grasom1)$
(%i14) grasom2: gr2d(line_width = 3, xaxis=true, yaxis=true,
                      color=blue, explicit(f1(x), x,-3, 3),
                      color=magenta, explicit(f2(x), x,-3, 3),
                      fill_color=gray,x voxel=30, y voxel=30,
                      region(y>f1(x) and y<f2(x) , x,-3, 3,y,-40,40))$
(%i15) draw(grasom2)$
```

donde en la primera solo dibujamos la región, y en la segunda incluimos las dos funciones (ver figura 3.16). Además hemos usado una opción interesante de MAXIMA que permite guardar en una variable la propia gráfica. Eso se hace con el comando gr2d (también se puede hacer con gráficas 3D usando gr3d). Las gráficas así guardadas luego se pueden reutilizar como mostraremos más adelante.

Una opción interesante del paquete draw es que permite hacer una matriz (*array*) de gráficas.

Por ejemplo, vamos a dibujar una rosa polar y una espiral hiperbólica

³Aquí hemos usado el operador lógico and que discutiremos con más detalle en la página 163.

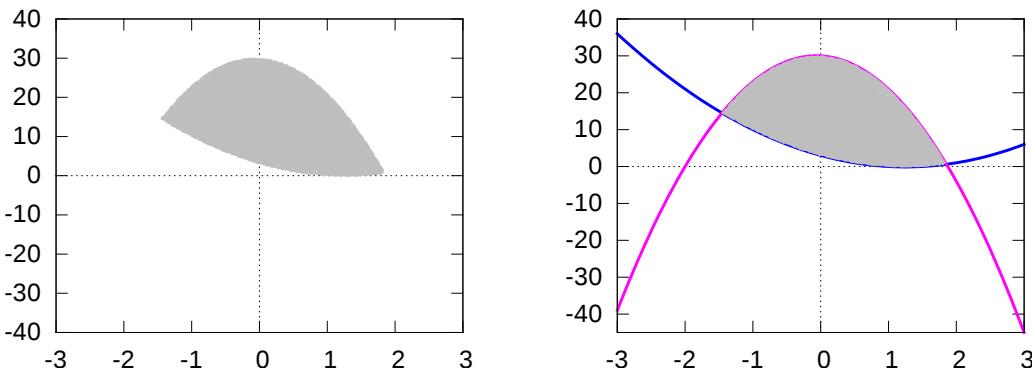


Figura 3.16: Región entre dos curvas $f(x) = 2x^2 - 5x + 3$ y $g(x) = -8x^2 - x + 30$ (izquierda) y dicha región junto a las funciones (derecha).

```
(%i16) grapol1:gr2d( user_preamble = "set grid polar",
                      nticks = 500,xrange = [-5,5],yrange = [-5,5],
                      color = red, line_width = 3, title = "rosa polar",
                      polar(3*cos (3/4*t),t,-10*pi,10*pi) )$
(%i18) draw(grapol1)$
(%i19) grapol2: gr2d( user_preamble = "set grid polar",
                      nticks = 200, xrange = [-5,5], yrange = [-5,5],
                      color = blue, line_width = 3,
                      title = "espiral hiperbólica",
                      polar(10/theta,theta,1,10*pi) )$
draw(grapol2)$
```

Y ahora representemos las dos gráficas en una misma ventana:

```
draw( columns = 2, dimensions = [1200,600],grapol1,grapol2);
```

Incluso podemos representar tres, cuatro, etc. (la última se representa en la figura 3.17):

```
(%i20) draw(columns=3, dimensions=[1800,600],grapol1,grapol2,grasom1);
(%t20) << Graphics >>
(%i21) wxdraw(columns=2,dimensions=[1200,1200],
               grapol1,grapol2,grasom1,grasom2);
(%t21) << Graphics >>
```

Vamos ahora a dibujar los primeros polinomios de Taylor de la función exponencial $f(x) = e^x$ alrededor de $x_0 = 0$. Para construirlos usaremos la orden `taylor` que ya hemos visto antes. Lo queharemos es crear la lista `tay` con los polinomios de Taylor de orden 1 a 5, y la exponencial, respectivamente. A continuación creamos la lista `legenda` que usaremos para nombrar los polinomios. Para ello usaremos el comando `concat(arg1,arg2,...)` que nos permite *concatenar* los argumentos `arg1, arg2, ...`. Aquí, por ejemplo, lo usamos para crear una lista con los distintos valores del orden del polinomio de Taylor. Nótese que los argumentos pueden ser cadenas, variables, etc. El

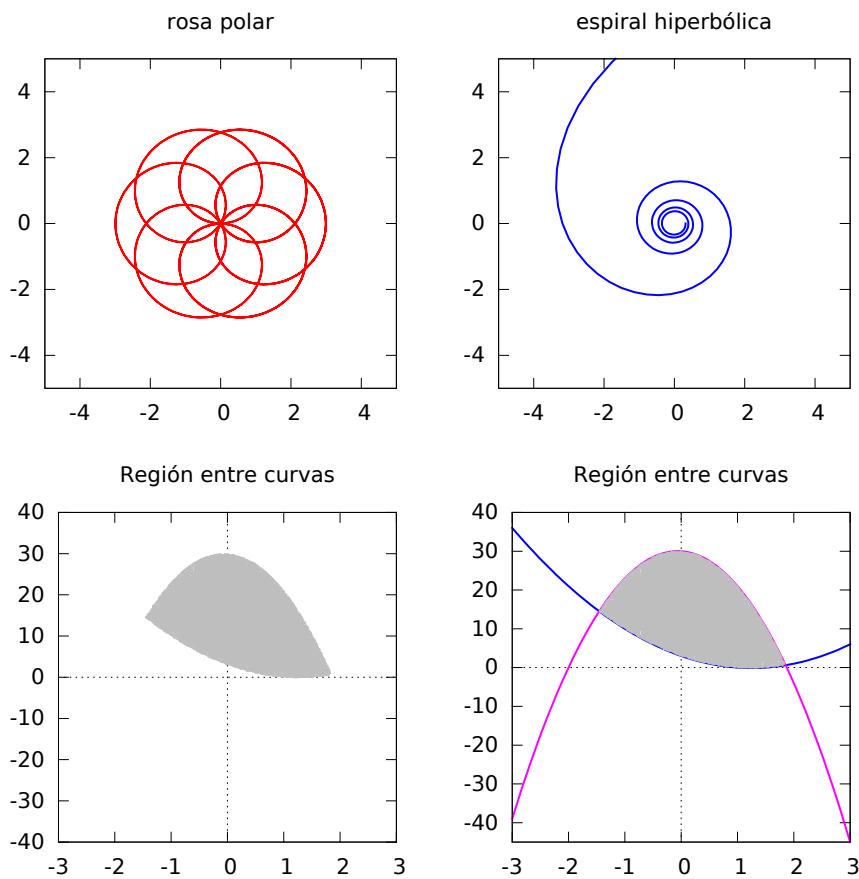


Figura 3.17: Varias figuras juntas en una matriz (array) de gráficas.

valor devuelto por `concat` es un símbolo si el primer argumento es a su vez un símbolo, o una cadena en caso contrario. Conviene además saber que la función `concat` evalúa sus argumentos a no ser que usemos el símbolo `'` para evitarlo. Comenzaremos usando el comando `plot2d`:

```
(%i1) p(x,k):=taylor(exp(x),x,0,k)$
(%i2) makelist(p(x,k),k,1,5)$
      tay:append([exp(x)],%)$
      makelist(concat("pol. Taylor orden ",k),k,0,4)$
      legenda: append([legend,"exp(x)"],%)$
(%i6) wxplot2d(tay,[x,-2,1],[y,-1,4],legenda,[xlabel,""])$
```

```
(%t6) << Graphics >>
```

La gráfica resultante la podemos ver en la figura 3.18 (izquierda). Si queremos tener más control sobre las opciones gráficas podemos usar `draw2d`. Por ejemplo con la secuencia siguiente, cuya gráfica está representada en la figura 3.18 (derecha).

```
(%i7) load(draw)$
      colors : ['green,'orange,'red,'magenta,'blue,'brown]$
      dd:makelist([key=concat("Pol. Taylor de orden ",k),color=colors[k],
      explicit(p(x,k),x,-2,1)],k,1,5)$
```

```
(%i10) draw2d( line_width=3, dd, yrange=[-1,3], color=black, key="exp(x)",
    explicit(exp(x),x,-2,1), grid = true, key_pos = top_left,
    dimensions = [800,800], font = "Arial", font_size = 12)$
(%t10) << Graphics >>
```

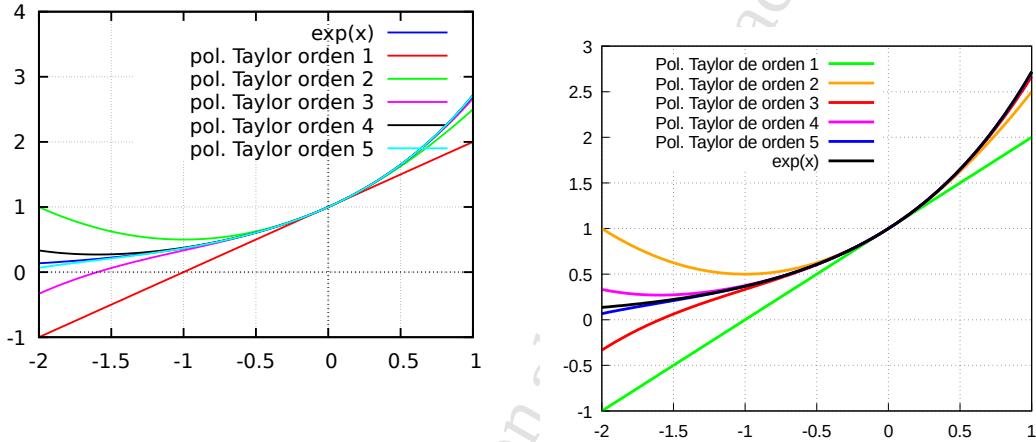


Figura 3.18: Gráficas de los polinomios de Taylor $p_1(x, 0), \dots, p_5(x, 0)$ de la función e^x alrededor del cero usando plot2d (izquierda) y draw2d (derecha).

Finalmente, debemos mencionar que podemos eliminar los distintos ejes usando las órdenes `axis_left=false`, `axis_right=false`, `axis_top=false`, `axis_bottom=false`, `xtics=false`, `ytics=false`.

3.2.3. Creando gráficas animadas.

Veamos a continuación como crear una animación con MAXIMA. Como ejemplo dibujemos la secuencia de aproximación de la exponencial e^x mediante los sucesivos polinomios de Taylor hasta orden 9.

La primera forma es muy simple, lo que se hace es representar una serie de gráficas, una a continuación de la otra, con `draw` combinadas con un ciclo para retrasar la visualización de cada uno:

```
(%i11) retraso:200000$ /* Depende de la potencia del ordenador */
for k:0 thru 9 step 1 do
  block([tiempo:1], while tiempo < retraso do tiempo: tiempo+1,
  draw2d( line_width = 3,dimensions = [1000,800],grid = true,
  key_pos = top_left, font = "Arial", font_size = 24,
  color=black, key="exp(x)", explicit(exp(x),x,-2,1),
  color=blue, key = concat("Pol. taylor de orden ",k),
  explicit(p(x,k),x,-2,1), yrange = [-0.5,3]) )$
```

Aquí hemos usado un comando muy cómodo para definir funciones dentro de MAXIMA: el comando `block` cuya sintaxis es

```
block ([v_1, ..., v_m], expr_1, ..., expr_n)
```

y que lo que hace es evaluar consecutivamente las expresiones `expr_1, ..., expr_n` y dando como resultado el valor de la última de las mismas, donde `v_1, ..., v_m` son variables locales dentro del `block`, es decir, no interfieren sus valores con los posibles valores globales dentro de la sesión de MAXIMA. También hemos usado el operador especial `while` que discutiremos en la página [153](#).

A parte de la forma anterior, MAXIMA cuenta con dos comandos muy sencillos: `with_slider` y `with_slider_draw` basados en `plot2d` y `draw`, respectivamente y que tienen sintaxis similares:

```
with_slider( variable , lista de valores de la variable , función )
```

donde `variable` es el nombre de la variable que cambia en los fotogramas (o gráficas individuales) que en nuestro ejemplo será el orden del polinomio de Taylor, `lista de valores de la variable` es, como su nombre indica, los valores que toma la variable, y por último, la orden para dibujar el gráfico. Dado que la manera de dibujar de `plot2d` y `draw` es distinta mostraremos como hacerlos en cada uno. De más está decir que ambos comandos aceptan las opciones de `plot2d` y `draw`, respectivamente.

```
(%i13) with_slider( k , /* variable de cambio */
makelist(i,i,0,9,1), /* lista de valores de la variable */
[exp(x),p(x,k)], [x,-2,1], [y,-0.5,3], /* funciones y opciones */
[legend,"exp(x)",concat("pol. Taylor orden ",k)])$
```



```
(%i14) with_slider_draw( k , /* variable de cambio */
makelist(i,i,0,9,1), /* lista de valores de la variable */
line_width = 3, /* funciones y opciones */
dimensions = [1000,800], grid = true, yrangle = [-0.5,3],
key_pos = top_left, font = "Arial", font_size = 24,
key=concat("Pol. Taylor de orden ",k), explicit(p(x,k),x,-2,1),
color=black, key="exp(x)", explicit(exp(x),x,-2,1))$
```

Un fotograma de ambas animaciones están representados en la figura [3.19](#).

Si queremos, podemos exportar los resultados a un fichero gif animado. Hay dos formas de hacerlo que funcionan de forma equivalente (al menos en LINUX). Una es con `draw` donde la variable interna `delay` determina el retraso en centésimas de segundo (por defecto se toma 5) y donde se ha de especificar el tipo de terminal como `animated_gif`

```
(%i15) draw( terminal = animated_gif, delay = 40, file_name= "taygifanim",
makelist(gr2d( line_width = 3,dimensions = [1000,800],
grid = true, key_pos = top_left, font = "Arial", font_size=24,
color=black, key = "exp(x)", explicit(exp(x),x,-2,1),
color=blue, key = concat("Pol. taylor de orden ",k),
explicit(p(x,k),x,-2,1), yrangle = [-.5,3]),k,9) )$
```

En en caso que no funcione este tipo de terminal (no todos los sistemas la tienen instalada) podemos combinar la creación de varios ficheros png y su conversión en un gif animado. La siguiente secuencia nos crea la animación deseada y la salva en el directorio animado:

```
(%i16) for k:1 thru 9 do
    draw2d(terminal = png,
    file_name = concat("animado/tay",add_zeroes(k)),
    line_width = 3,dimensions = [1000,800],grid = true,
    key_pos = top_left, font = "Arial", font_size = 24,
    color=black, key="exp(x)", explicit(exp(x),x,-2,1),
    color=blue, key = concat("Pol. taylor de orden ",k),
    explicit(p(x,k),x,-2,1), yrangle = [-0.5,3] ) $
(%i16) system("convert -delay 100 animado/*.png animado/tay.gif")$
```

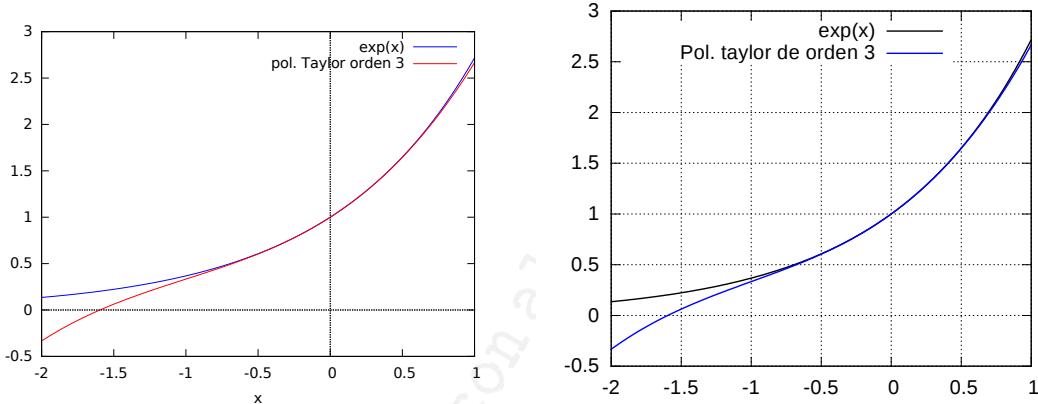


Figura 3.19: Fotograma para $k=3$ de las animaciones de las salidas (%i13) con `with_slider` (izquierda) y (%i14) con `with_slider_draw` (derecha), respectivamente.

Veamos una animación 3D. Para ello usaremos la orden `with_slider_draw3d` y representaremos una partícula que se mueve por la trayectoria paramétrica definida por $(2 \cos u, 2 \sin u, u)$. Es conveniente fijar todos los parámetros relacionados con la dimensión para que la animación no cambie la escala ni el ángulo de visión en cada una de sus gráficas (fotogramas). Así, fijamos el ángulo de visión con `view`, y seleccionamos el rango de las variables con `xrange`, `yrange`, `zrange`. En la primera animación vemos la partícula moverse en el espacio donde hemos dejado los ejes coordenados visibles

```
(%i17) kill(all)$
(%i1) load(draw)$
(%i2) with_slider_draw3d(k , /* variable de cambio */
    makelist(i,i,0,30,1), /* lista de valores de la variable */
    color = blue, point_type=filled_circle, point_size =2,
    points([[2*cos(2*pi*k/10),2*sin(2*pi*k/10), 2*pi*k/10]]),
    color = magenta, nticks = 50, xtics = 1, ytics = 1, ztics = 10,
    line_width=3, zrange=[0,20], xrange=[-2.1,2.1],yrange=[-2.1,2.1],
    parametric(2*cos(u),2*sin(u),u,u,0,2*pi*k/10),
    view=[60,60],proportional_axes = xy)$
(%t2) << Graphics >>
```

y en la segunda, hemos eliminado los ejes:

```
(%i3) with_slider_draw3d(k , /* variable de cambio */
    makelist(i,i,0,30,1), /* lista de valores de la variable */
    axis_3d = false, xtics=false,ytics=false,ztics=false,
    color = blue, point_type=filled_circle, point_size =2,
    points([[2*cos(2*pi*k/10),2*sin(2*pi*k/10), 2*pi*k/10 ]]),
    color = magenta, nticks = 50,
    line_width=3,zrange=[0,20],xrange=[-2.1,2.1],yrange=[-2.1,2.1],
    parametric(2*cos(u),2*sin(u),u,u,0,2*pi*k/10),
    view=[60,60],proportional_axes = xy)$
(%t3) << Graphics >>
```

Una vista del fotograma correspondiente a $k=20$ lo tenemos en la figura 3.20

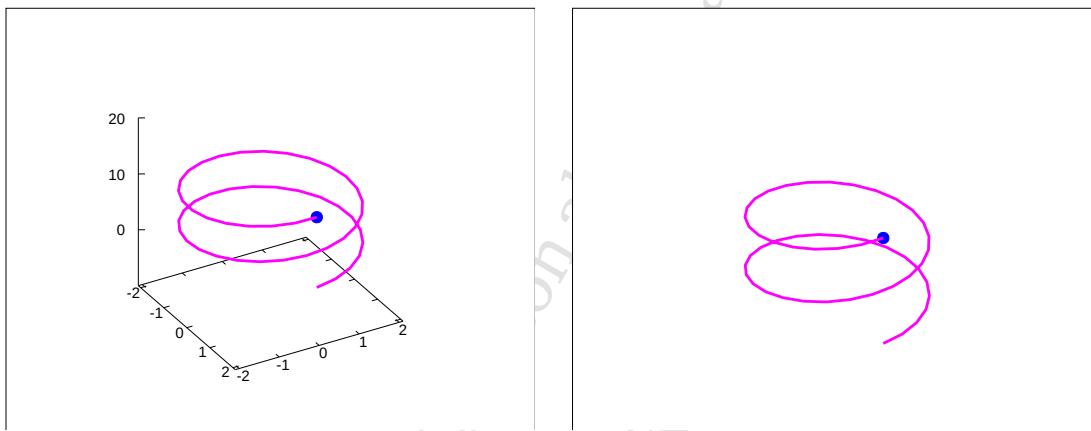


Figura 3.20: Fotograma para $k=20$ del movimiento de la partícula según la trayectoria definida por $(2 \cos u, 2 \sin u, u)$ (con y sin ejes).

En todos los casos donde hemos usado los comandos `with_slider`, `with_slider_draw` y `with_slider_draw3d` conviene tener en cuenta que la variable que identifica cada fotograma (en nuestros ejemplos k) no debe tener asignado previamente ningún valor.

Si queremos crear un gif animado podemos usar la siguiente secuencia, muy similar a la que ya vimos para el caso 2D:

```
(%i3) draw( terminal = animated_gif, delay=40, file_name = "pelipartejes",
    makelist(gr3d( xtics = 1, ytics = 1, ztics = 10,
    color = blue, point_type=filled_circle, point_size =2,
    points([[2*cos(2*pi*k/10),2*sin(2*pi*k/10), 2*pi*k/10 ]]),
    color = magenta, nticks = 50,
    line_width=3,zrange=[0,20],xrange=[-2.1,2.1],yrange=[-2.1,2.1],
    parametric(2*cos(u),2*sin(u),u,u,0,2*pi*k/10),
    view=[60,60],proportional_axes = xy ) ,k,0,30))$
```

Como ejercicio dejamos al lector que genere el gif animado eliminando los ejes coordenados.

Para terminar esta sección dedicada a la representación gráfica con MAXIMA recomendamos al lector visitar la web de Mario Rodríguez Riotorto, desarrollador del paquete `draw` entre otros del proyecto MAXIMA: <http://www.tecnostats.net> donde podrá encontrar otros ejemplos interesantes.

3.3. MAXIMA, GNUPLOT y LATEX

En este último apartado vamos a discutir como crear gráficas en pdf de gran calidad usando MAXIMA y las capacidades de LATEX. Comenzaremos definiendo las siguientes dos funciones ϕ y ψ

$$\phi(x) = \frac{\sqrt{2} e^{-2x^2}}{\pi^{1/4}}, \quad \psi(x) = \frac{e^{-x^2/8}}{\sqrt{2} \pi^{1/4}}.$$

```
(%i1) phi(p):=(sqrt(2)*%e^(-2*p^2))/%pi^(1/4)$
(%i2) psi(x):=%e^(-x^2/8)/(sqrt(2)*%pi^(1/4))$
```

Una forma de dibujarlas, ver la gráfica de la izquierda de la figura 3.21, es usando la siguiente secuencia de comandos⁴:

```
(%i3) wxdraw2d( yrangle = [0,1.25], ylabel=concat("funciones phi y psi"),
    grid=true, line_type=solid, line_width=5, nticks=100, key_pos=top_right,
    key=concat("phi(x)"), color=red, explicit(abs(phi(x))^2,x,-10,10) ,
    key=concat("psi(x)"), color=blue, explicit(abs(psi(p))^2,p,-10,10),
    font="Arial", font_size=25, dimensions = [1500,1200] )$
```

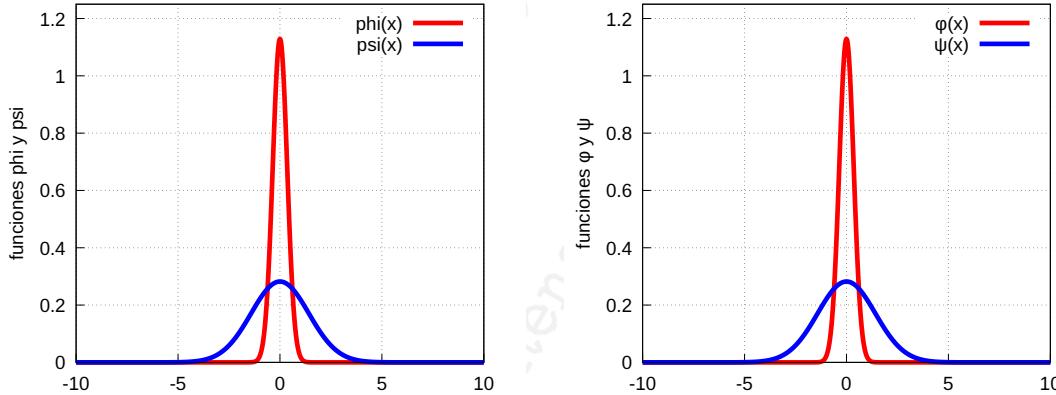


Figura 3.21: Gráfica de las funciones ψ y ϕ usando los comandos estándar de MAXIMA (izquierdo) y wxMAXIMA (derecha).

Aunque la gráfica de la derecha puede parecer bastante buena, esta no es nada comparable si usamos el entorno LATEX. Para ello hay que tener en cuenta que MAXIMA usa GNUPLOT para dibujar las gráficas. Para ello, aparte de saber usar MAXIMA hay que estar familiarizado con LATEX⁵

El código para nuestro ejemplo es el siguiente:

```
(%i5) grafica: gr2d( yrangle=[0,1.25], ylabel="funciones $\\phi$ y $\\psi$",
    grid=true, line_type=solid, line_width=5, nticks=100, key_pos=top_right,
```

⁴Usando wxMAXIMA se pueden escribir las letras griegas ϕ y ψ de forma que la gráfica es la que se muestra en la figura 3.21 derecha.

⁵Un ejemplo representativo se puede encontrar en <https://riotorto.users.sourceforge.net/Maxima/gnuplot/label/index.html#label18>. Nótese que hay que usar dobles \\ para los comandos LATEX, i.e., \\phi en vez de \phi.

```

key = "$\\phi(x)$", color=red, explicit(abs(phi(x))^2,x,-10,10),
key = "$\\psi(x)$", color=blue, explicit(abs(psi(p))^2,p,-10,10),
dimensions = [1000,800] )
wxdraw(grafica);

```

cuya salida vemos en la gráfica de la izquierda de la figura 3.22. Para obtener el pdf de la figura en el entorno LaTeX hay que ejecutar (en mi caso en LINUX, en otro sistema operativo puede ser diferente) la secuencia

```
(%i5) draw(terminal=epslatex_standalone, dimensions=[1000,800], grafica)$
system("pdflatex maxima_out.tex")$
```

que genera el fichero `maxima_out.pdf` que mostramos en la en la gráfica de la derecha de la figura 3.22. El lector puede comparar las distintas gráficas y decidir por si mismo cuál es la de mejor calidad.

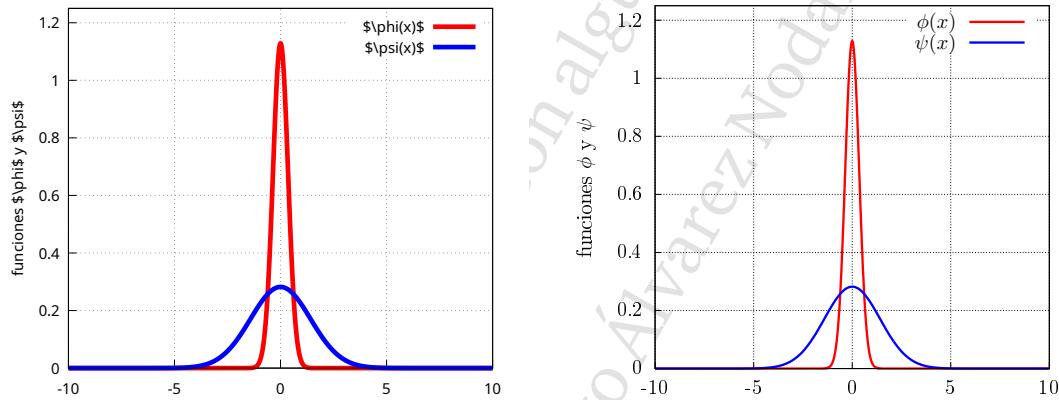


Figura 3.22: Gráfica de las funciones ψ y ψ usando los comandos de LATEX.

Capítulo 4

Más comandos útiles de MAXIMA.

En este capítulo vamos a discutir otros comandos útiles a la hora de resolver problemas matemáticos usando MAXIMA.

4.1. Resolviendo ecuaciones y manipulando expresiones algebraicas.

4.1.1. Resolviendo ecuaciones.

Comenzaremos con las ecuaciones. Ya hemos visto que el comando `solve` es capaz de resolver muchas ecuaciones. Veamos dos muy sencillas. La primera es $e^{2x} = 3$ que MAXIMA resuelve sin problemas¹

```
(%i1) solve(exp(2*x)-3=0);
(%o1) [x=log(-sqrt(3)),x=log(3)/2]
(%i2) ec: sin(x)=1;
```

y la segunda, $\sin(x) = 1$, con la que tiene un problema

```
(%o2) sin(x)=1
(%i3) solve(ec);
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
(%o3) [x=%pi/2]
```

y nos advierte que puede perder soluciones, como de hecho ocurre pues $\sin(\pi/2 + 2k\pi) = 1$ para todo k entero. Sin embargo, al intentar resolver la ecuación $e^x - x - 2 = 0$, la salida es la propia ecuación reescrita

```
(%i4) solve(exp(x)-x-2=0);
(%o4) [x=%e^x-2]
```

¹El lector avisado se habrá dado cuenta de que la primera solución que da MAXIMA es algo “rara”. ¿Está MAXIMA calculando mal? Como pista de lo que ha hecho MAXIMA recomendamos al lector que use la opción `logexpand:super` que mencionamos en la página 20. Esto es un ejemplo más de que programas simbólicos como MAXIMA deben usarse con cierto cuidado y analizando las correspondientes salidas pues, en este caso a

es decir, MAXIMA no puede resolver analíticamente la ecuación². Cuando ocurre esto no tenemos otra opción que obtener una solución numérica. Para ello usamos el comando `find_root`, cuya sintaxis es

```
find_root(ecuacion, variable, inicio de intervalo, final de intervalo)
```

donde además de la ecuación y la variable respecto a la cual hay que resolvérla, tenemos que indicar el inicio y el final del intervalo donde MAXIMA debe buscar la solución. Así, escribimos

```
(%i5) find_root(exp(x)-x-2=0,x,0,1);
find_root: function has same sign at endpoints:
mequal(f(0.0), -1.0), mequal(f(1.0), -0.28171817154095)
-- an error. To debug this try: debugmode(true);
```

Aquí MAXIMA nos indica que no puede resolver la ecuación pues el signo de la función en los extremos del intervalo coinciden (en este ejemplo ambos son negativos) por lo que su algoritmo falla (se basa en el teorema de Bolzano para funciones continuas). Una solución es dibujar la función

```
(%i6) wxplot2d(exp(x)-x-2,[x,-3,3]);
```

y a partir del gráfico definir un intervalo adecuado (en nuestro ejemplo el $[0, 2]$ nos valdría). Lo anterior nos permite calcular la solución entre en el interior del intervalo $(0, 2)$

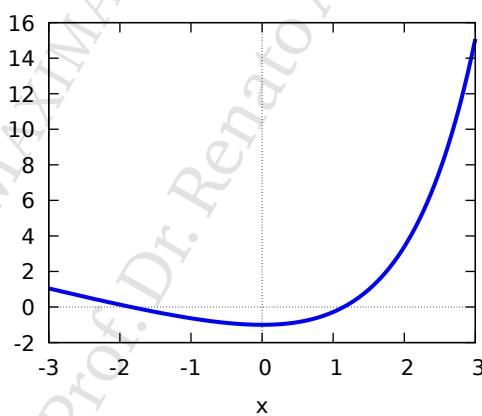


Figura 4.1: Gráfica de la función $f(x) = e^x - x - 2$ en $(-3, 3)$.

```
(%i7) find_root(exp(x)-x-2=0,x,0,2);
(%o7) 1.146193220620583
```

Precisamente para encontrar un intervalo viable que nos permita calcular las raíces de una ecuación usando la orden `find_root` son de gran utilidad las posibilidades interactivas del comando `plot2d` que comentamos en la página 9 del capítulo 2.

diferencia del ejemplo de la ecuación $\sin(x) = 1$ MAXIMA no nos da ninguna alerta (warning).

²Algo similar pasa si usamos la orden `algsys([exp(x)-x-2=0],[x])` que discutiremos en breve.

Por otro lado hay que tener cuidado al usar `find_root` para calcular las raíces ya que este no tiene por qué encontrar todas las soluciones de la ecuación. Por ejemplo si intentamos resolver la ecuación $\sin(x) = 1/2$ en el intervalo $[-7, 7]$ MAXIMA nos calcula una única solución cuando en realidad hay siete.

Para resolver ecuaciones numéricamente existen otros dos comandos que conviene conocer y que son `newton(expr, x, x0, eps)` y `mnewton(listafun, listavar, listvalini)`, que ejecutan el método de Newton para el cálculo de raíces para ecuaciones (`expr=0`) y sistemas de ecuaciones (`listafun=[expr1, ..., exprN]`), respectivamente. En el primer caso `x`, `x0`, `eps` son la variable, el valor inicial de la misma y la precisión, respectivamente, mientras que en la segunda `listavar`, `listvalin` es la lista de las variables y la lista de sus valores iniciales. En ambos casos hay que cargar previamente los paquetes “`newton1`” y “`mnewton`”, respectivamente. Dejamos como ejercicio que el lector los use en los ejemplos siguientes:

```
load(newton1)$
newton (exp(x)-x-2, x, 0.1, 10^(-3));
newton (exp(x)-x-2, x, 0.1, 10^(-16));

load(mnewton)$
mnewton([exp(x)-x-2],[x],[0.1]);
mnewton([exp(x)-y-2,exp(y)+x-1],[x,y],[0.1,1]);
```

Los comandos `find_root`, `newton` y `mnewton` calculan las soluciones para ecuaciones en general, no obstante MAXIMA tiene un comando específico para calcular las raíces numéricas reales y complejas de un polinomio: la orden `allroots` (véase también la orden `realroots`).

4.1.2. Resolviendo sistemas de ecuaciones.

El comando `solve` también resuelve sistemas de ecuaciones. Consideremos un sistema de dos ecuaciones con tres incógnitas. Le podemos pedir a MAXIMA que lo resuelva pero tenemos que especificarle con respecto a qué variables

```
(%i8) ec1:2*x+3*y-z=1;
(%o8) -z+3*y+2*x=1
(%i9) ec2:3*x-y+z=0;
(%o9) z-y+3*x=0
(%i10) solve([ec1,ec2],[x,y]);
(%o10) [[x=-(2*z-1)/11,y=(5*z+3)/11]]
```

Como se ve, MAXIMA entiende que `z` es un parámetro libre y deja la solución en función del mismo. Si le pedimos que resuelva el sistema respecto a las tres variables entonces introduce las constantes que necesite (en este caso `%r1`) en el caso de que el sistema sea indeterminado (como el del ejemplo que nos ocupa). Así, la solución queda definida en función del parámetro independiente `%r1`

```
(%i11) solve([ec1,ec2],[x,y,z]);
(%o11) [[x=-(2*%r1-1)/11,y=(5*%r1+3)/11,z=%r1]]
```

En caso de un sistema determinado, nos da la solución completa:

```
(%i12) ec3:x+y+z=1;
(%o12) z+y+x=1
(%i13) solve([ec1,ec2,ec3],[x,y,z]);
(%o13) [[x=0,y=1/2,z=1/2]]
```

Siempre que puede `solve` nos da una solución analítica:

```
(%i14) solve([x^2+y^2=1,x-y=2],[x,y]);
(%o14) [[x=-(sqrt(2)*%i-2)/2,y=-(sqrt(2)*%i+2)/2],
       [x=(sqrt(2)*%i+2)/2,y=(sqrt(2)*%i-2)/2]]
```

Cuando no puede, pasa directamente a calcularla numéricamente

```
(%i15) solve([x^3+y^2=1,x-y=2],[x,y]);
(%o15) (%o25) [[x=0.5033313284659912*%i+0.903150358370507,
               y=0.5033313284659912*%i-1.096849641629493],
               [x=0.903150358370507-0.5033313284659912*%i,
               y=-0.5033313284659912*%i-1.096849641629493],
               [x=-2.80630068621335,y=-4.806301050175029]]
```

o bien nos da error en caso de no poder resolverlo de ninguna de las dos formas

```
(%i16) solve([x^3+y^2=1,sin(x)-y=2],[x,y]);
algsys: tried and failed to reduce system to a polynomial
in one variable; give up.
-- an error. To debug this try: debugmode(true);
```

Además, si nos fijamos en el mensaje que nos da MAXIMA vemos que cuando `solve` no puede calcular analíticamente la solución entonces pasa al cálculo numérico usando `algsys`. De hecho `algsys` es junto a `linsolve` otra de las órdenes que tiene MAXIMA para resolver sistemas aparte de la orden `solve`. La sintaxis de dichas órdenes son

```
algsys([ecuaciones],[variables]) y linsolve([ecuaciones],[variables])
```

respectivamente. La primera de ellas se usa para resolver sistemas de ecuaciones algebraicas y tiene la ventaja de que si el sistema no es resoluble de forma analítica lo intenta de forma numérica

```
(%i17) kill(all)
(%i1) solve(x^7+2*x+1);
(%o1) [0=x^7+2*x+1]
(%i2) algsys([x^7+2*x+1=0],[x]);
(%o2) [[x=0.07635557774869883-1.140946142049927*%i], ...
       , [x=-0.4962920707358813]]
```

Si solo nos interesan las soluciones reales conviene saber que `algsys` reconoce la variable global `realonly` que cuando toma el valor `true` no tendrá en cuenta las soluciones complejas. Por ejemplo, la misma salida anterior con la opción `realonly:true` sería

```
(%o2) [[x=-0.4962920707358813]]
```

El otro comando `linsolve` resuelve sistemas de ecuaciones lineales

```
(%i3) eq: [x+2*y+3*z=a, 3*x+2*y+z=-a, x-y+z=a] $  

(%i4) solve(eq, [x, y, z]);  

(%o4) [[x=-a/4, y=-a/2, z=(3*a)/4]]  

(%i5) linsolve(eq, [x, y, z]);  

(%o5) [x=-a/4, y=-a/2, z=(3*a)/4]  

(%i6) linsolve(eq, [x, y, z, a]);  

(%o6) [x=-%r4/4, y=-%r4/2, z=(3*%r4)/4, a=%r4]
```

Nótese que la salida depende de cuales son las incógnitas, así el mismo sistema `eq` tiene soluciones distintas si se resuelve respecto a las variables `x, y, z` o respecto a `x, y, z, a`.

Existe otro comando que conviene tener en cuenta: `eliminate` y cuya sintaxis es

```
eliminate ([eq1, eq2, ..., eqn], [x1, x2, ..., xk])
```

que intenta eliminar las variables x_1, \dots, x_k de las ecuaciones eq_1, \dots, eq_n ($k < n$). Por ejemplo:

```
(%i7) eq: [x-y=0, x^2+y^2=1];  

(%o7) [x-y=0, y^2+x^2=1]  

(%i8) eliminate(eq, [x]);  

(%o8) [2*y^2-1]  

(%i9) eliminate(eq, [y]);  

(%o9) [2*x^2-1]
```

4.1.3. Un poco más sobre la simplificación de expresiones.

A parte de los comandos de simplificación de expresiones algebraicas como `ratsimp`, `factor` y `expand`, que ya hemos visto en las secciones anteriores MAXIMA cuenta con comandos específicos de simplificación de funciones trigonométricas, exponenciales y logarítmicas. En el caso de usar wxMAXIMA se puede acceder a ellos través de la pestaña “Simplificar”. Veamos a continuación algunos ejemplos.

Comenzaremos discutiendo con más detalle el comando `ratsimp` que como vimos simplifica expresiones racionales de tipo polinómico. Hay que hacer notar los comandos de simplificación no solo simplifican expresiones como hemos visto hasta ahora sino cualquier objeto en general como por ejemplo, ecuaciones o listas. Así, tenemos

```
(%i1) ratsimp( sin(x/(x^2 + x)) = exp((log(x)+1)^2-log(x)^2) );  

(%o1) sin(1/(x+1))=%e*x^2
```

o

```
(%i2) ratsimp( ((x - 1)^(3/2) - (x+1)*sqrt(x-1))/sqrt((x-1)*(x+1)) );  

(%o2) -(2*sqrt(x-1))/sqrt(x^2-1)
```

Aquí vemos que `ratsimp` no puede simplificar las expresiones que están en las distintas raíces. Si tenemos expresiones que involucran radicales, exponenciales y logaritmos conviene usar en vez de `ratsimp` la orden `radcan`.

```
(%i3) radcan( % );
(%o3) -2/sqrt(x+1)
```

Por ejemplo, compárese la salida de

```
(%i4) radcan([sqrt(x+1)/sqrt(x^2-1), log(x^2-2*x+1)/log(x-1)]);
(%o4) [1/sqrt(x-1), 2]
```

con la correspondiente usando `ratsimp`.

Conviene también mencionar que `ratsimp` no siempre simplifica *hasta el final*. En esos casos conviene usar `fullratsimp`. Por ejemplo

```
(%i5) expr1: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
(%o5) ((x^(a/2)-1)^2*(x^(a/2)+1)^2)/(x^(a-1))
(%i6) ratsimp(expr1);
(%o6) (x^(2*a)-2*x^(a+1))/(x^(a-1))
(%i7) fullratsimp(expr1);
(%o7) x^(a-1)
```

Veamos algunos ejemplos más de expresiones que involucran radicales, exponenciales y logaritmos. Por ejemplo simplifiquemos las expresiones

$$\frac{(\log(x^2 + x) - \log x)^a}{(\log(x + 1))^{a/2}} \text{ y } \sin\left(\frac{(x^2 + x) e^{\frac{3\log(x+1)}{2} - \frac{\log(x-3)}{2}}}{x(2x^2 + 2x)}\right).$$

Comenzamos con la primera. En este caso `ratsimp` (tampoco lo hace `fullratsimp`) no hace nada mientras que `radcan` simplifica al máximo la expresión

```
(%i8) ratsimp((log(x+x^2)-log(x))^a/log(1+x)^(a/2));
(%o8) (log(x^2+x)-log(x))^a/log(x+1)^(a/2)
(%i9) fullratsimp((log(x+x^2)-log(x))^a/log(1+x)^(a/2));
(%o9) (log(x^2+x)-log(x))^a/log(x+1)^(a/2)
(%i10) radcan((log(x+x^2)-log(x))^a/log(1+x)^(a/2));
(%o10) log(x+1)^(a/2)
```

Para la segunda, tanto `ratsimp` como `fullratsimp` solo simplifican la parte racional como era de esperar

```
(%i11) expr2: sin((x^2+x)/x*exp((3*log(x+1))/2-log(x-3)/2)/(2*x^2+2*x));
(%o11) sin(((x^2+x)*%e^((3*log(x+1))/2-log(x-3)/2))/(x*(2*x^2+2*x)))
(%i12) ratsimp(expr2);
(%o12) sin(%e^((3*log(x+1))/2-log(x-3)/2)/(2*x))
(%i13) fullratsimp(expr2);
(%o13) sin(%e^((3*log(x+1))/2-log(x-3)/2)/(2*x))
```

No así `radcan` que da una simplificación más completa

```
(%i14) radcan(expr2);
(%o14) sin((x+1)^(3/2)/(2*sqrt(x-3)*x))
```

Nótese que si usamos `radcan` para simplificar la expresión de la entrada (%i5) obtenemos el mismo resultado que el de la orden `fullratsimp`

```
(%i15) radcan(expr1);
(%o15) x^a-1
```

En ocasiones las órdenes `ratsimp` y `fullratsimp` no son de ayuda como es el caso del siguiente ejemplo

```
(%i16) expr2: ((x+2)^20 - 2*y)/(x+y)^20 + (x+y)^(-19) - x/(x+y)^20;
(%o16) 1/(y+x)^19+((x+2)^20-2*y)/(y+x)^20-x/(y+x)^20
(%i17) ratsimp(expr2);
(%o17) -(y-x^20-40*x^19-760*x^18- ... -49807360*x^2-10485760*x-1048576)/
(y^20+20*x*y^19+190*x^2*y^18+ ... +190*x^18*y^2+20*x^19*y+x^20)
```

En este caso es mejor usar la orden `xthru(expr)` que combina todos los términos de `expr` (la cual debe ser una suma) sobre un común denominador sin expandir productos ni sumas elevadas a exponentes. Así se tiene

```
(%i18) xthru (expr2);
(%o18) ((x+2)^20-y)/(y+x)^20
```

Antes de continuar debemos mencionar que la forma de simplificar que tiene `radcan` está controlada por la variable `radexpand` con valores `true` (por defecto), `all` y `false`. Como ejemplo mostramos a continuación distintas salidas de la orden `radcan` para distintos valores de la variable `radexpand`:

```
(%i19) sqrt (x^2); radexpand;
(%o19) abs(x)
(%o20) true
(%i21) radexpand:all$ sqrt (x^2);
(%o22) x
(%i23) radexpand:false$ sqrt (x^2);
(%o24) sqrt(x^2)
(%i25) sqrt (16*x^2);
(%o25) sqrt(16*x^2)
(%i26) radexpand:all$ sqrt (16*x^2);
(%o27) 4*x
(%i28) radexpand:true$ sqrt (16*x^2);
(%o29) 4*abs(x)
```

Antes de pasar a discutir otros comandos debemos mencionar que el comando `radcan` consume muchos recursos por lo que muchas veces es conveniente intentar simplificar lo máximo posible antes de usarlo. Se recomienda al lector consultar la sección 14 sobre polinomios de manual [1] para más detalles sobre la manipulación de expresiones polinómicas y racionales así como la sección 9 sobre simplificación.

A parte de las órdenes discutidas anteriormente MAXIMA cuenta con comandos específicos para la manipulación de expresiones trigonométricas. Entre ellos tenemos a `trigsimp`, `trigreduce` y `trigrat`.

Veamos ahora la orden `exponentialize` que transforma en exponenciales todas las funciones que aparecen, en caso de ser posible. Por ejemplo, imaginemos que queremos desarrollar la expresión $4/((1 - e^{2b}) \operatorname{senh}^2 b)$. Si usamos el comando `expand`, este no hace nada, pero si la transformamos en exponenciales podemos simplificarla sin problemas

```
(%i30) kill(all)$
(%i1) expr:ratsimp( 4/(1-%e^(2*b))*sinh(b)^2 );
(%o1) -(4*sinh(b)^2)/(%e^(2*b)-1)
(%i2) exponentialize(%);
(%o2) -(%e^b-%e^(-b))^2/(%e^(2*b)-1)
(%i3) ratsimp(%);
(%o3) -%e^(-2*b)*(%e^(2*b)-1)
```

Para simplificar expresiones trigonométricas conviene usar comandos específicos. En nuestro ejemplo usaremos `trigexpand` para desarrollar la expresión.

```
(%i4) A*cos(theta+phi+psi);
(%o4) cos(theta+psi+phi)*A
(%i5) expand(%);
(%o5) cos(theta+psi+phi)*A
(%i6) trigexpand(%);
(%o6) (-cos(phi)*sin(psi)*sin(theta)-sin(phi)*cos(psi)*sin(theta)-
sin(phi)*sin(psi)*cos(theta)+cos(phi)*cos(psi)*cos(theta))*A
```

o bien, `trigreduce` para simplificarla³

```
(%i7) ratsimp(%);
(%o7) ((-cos(phi)*sin(psi)-sin(phi)*cos(psi))*sin(theta)+
(cos(phi)*cos(psi)-sin(phi)*sin(psi))*cos(theta))*A
(%i8) trigreduce(%);
(%o8) cos(theta+psi+phi)*A
```

Cuando uno trabaja con expresiones trigonométricas conviene tener en cuenta las siguientes opciones:

- `trigexpandplus` (valor por defecto `true`) que determina si las expresiones del tipo $\sin(x + y)$, y similares, son o no desarrolladas
- `trigexpandtimes` (valor por defecto `true`) determina si las expresiones del tipo $\sin(kx)$, $k \in \mathbb{N}$ y similares serán o no desarrolladas,
- `trigsign` (valor por defecto `true`) que incluye identidades del tipo $\sin(-x) = -\sin(x)$,
- `triginverses` controla la simplificación de las composiciones de funciones trigonométricas e hiperbólicas con sus funciones inversas. Puede tomar los valores `all` (valor por defecto), `true` y `false`,
- `halfangles` (valor por defecto, `false`), que controla si las fórmulas con ángulo mitad serán o no reescritas en términos del ángulo completo.

³Puede que se necesite ejecutar `trigreduce` dos veces.

Otra opción muy interesante de MAXIMA es que podemos definir el tipo de variable que estamos usando. Ello se hace con el comando `declare` (véase el manual [1] para más detalles sobre este versátil comando). Por ejemplo si hacemos

```
(%i9) kill(n,m)$
(%i10) [sin (%pi * n), cos (%pi * m), sin (%pi/2 * m), cos (%pi/2 * m)];
(%o10) [sin(%pi*n),cos(%pi*m),sin((%pi*m)/2),cos((%pi*m)/2)]
```

MAXIMA realiza la operación asumiendo que n y m son números cualesquiera. En cambio, si escribimos

```
(%i11) declare (n, integer, m, even)$
(%i12) [sin(%pi * n), cos(%pi * m), sin(%pi/2 * m),cos (%pi/2 * m)];
(%o12) [0,1,0,(-1)^(m/2)]
(%i13) facts();
(%o13) [kind(n,integer),kind(m,even)]
```

entonces MAXIMA entiende que n y m son enteros y que además m es par y procede a simplificar las expresiones.

Las posibilidades de simplificar expresiones con MAXIMA son enormes. Es conveniente echarle un vistazo a las secciones §9 y §14 del manual [1] antes mencionadas.

4.1.4. Trabajando con desigualdades.

Finalmente, mostremos como MAXIMA resuelve desigualdades. Para ello hay que cargar el paquete `fourier_elim`. La sintaxis del comando es

```
fourier_elim ([eq1, eq2, ...], [var1, var, ...])
```

donde `[eq1, eq2, ...]` es una lista de desigualdades lineales y `[var1, var, ...]`, son las correspondientes variables. Hay algunos casos de desigualdades no lineales que involucran a las funciones valor absoluto, mínimo y máximo, y a algunas expresiones que son productos o cocientes de términos lineales que también se pueden resolver con esta orden.

Como ejemplo resolvamos algunas desigualdades sencillas (las gráficas se incluyen como forma de comprobación visual y no las incluiremos aquí):

```
(%o14) kill(all)$
(%i1) load(fourier_elim)$
(%i2) ineq:(x-1)*(x-2)>0;
        fourier_elim([ineq],[x]);
(%o2) (x-2)*(x-1)>0
(%o3) [2<x] or [x<1]
(%i4) wxplot2d([(x-1)*(x-2)], [x,0,3])$
```

Como se ve en la gráfica MAXIMA encuentra los intervalos adecuados. En la desigualdad pueden aparecer igualdades:

```
(%i5) ineq1:(x-1/2)*(x-5/2)<=0;
```

```
(%o5) (x-5/2)*(x-1/2)<=0
(%i6) fourier_elim([ineq1],[x]);
(%o6) [x=1/2] or [x=5/2] or [1/2<x,x<5/2]
(%i7) wxplot2d([(x-1/2)*(x-5/2)], [x,0,3])$
```

También funciona para un sistema de desigualdades

```
(%i8) fourier_elim([ineq,ineq1],[x]);
(%o8) [x=1/2] or [x=5/2] or [2<x,x<5/2] or [1/2<x,x<1]
(%i9) wxplot2d([(x-1)*(x-2),(x-1/2)*(x-5/2)], [x,0,3])$
```

En lugar de una lista de inecuaciones, el primer argumento de `fourier_elim` puede ser una conjunción o disyunción lógica⁴

```
(%i10) fourier_elim((x + y < 5) and (x - y > 8),[x,y]);
(%o10) [y+8<x,x<5-y,y<-3/2]
(%i11) fourier_elim(((x + y < 5) and x < 1) or (x - y > 8),[x,y]);
(%o11) [y+8<x] or [x<min(1,5-y)]
```

Veamos como funciona en algunos casos *no lineales*:

```
(%i12) fourier_elim([(x+1)/(x-2) <= 3],[x]);
(%o12) [x=7/2] or [7/2<x] or [x<2]
(%i13) wxplot2d([(x+1)/(x-2),3], [x,-1.2,5], [y,-10,10])$
```

(%i14) fourier_elim([max(x,y) < 1,min(x,y)>-1],[x,y]);
(%o14) [-1<x,x<1,-1<y,y<1]
(%i15) plot3d([(abs(x-y)+x+y)/2,1,-1, [x,-4,4], [y,-4,4]])\$

En el último gráfico hemos usado la identidad $\max(x,y) = (|x - y| + x + y)/2$. Finalmente mostremos que no siempre MAXIMA es capaz de resolver el problema

```
(%i16) fourier_elim([x^2 - 1 # 0],[x]);
(%o16) [-1<x,x<1] or [1<x] or [x<-1]
(%i17) fourier_elim([x^3-1 # 0],[x]);
(%o17) [x<1,x^2+x+1>0] or [1<x,-(x^2+x+1)>0] or [1<x,x^2+x+1>0]
          or [x<1,-(x^2+x+1)>0]
(%i18) wxplot2d([(x^2+x+1)], [x,-3,3])$
```

Como vemos en el último gráfico la expresión $x^2 + x + 1$ siempre es positiva (de lo cual es fácil convencerse calculando sus raíces), por lo que la solución debería ser simplemente el intervalo $(-\infty, 1) \cup (1, +\infty)$.

4.2. Trabajando con matrices.

Las matrices en MAXIMA se introducen mediante el comando `matrix` especificando cada una de las filas de la misma, i.e.,

```
matrix(fila1,fila2,...)
```

⁴Nuevamente hemos usado el operador lógico `and` que discutiremos con más detalle en la página 163.

donde fila1, fila2, etc, son listas de la misma longitud.

```
(%i1) A:matrix([1,2],[2,1]);
(%o1) matrix([1,2],[2,1])
(%i2) B:matrix([1,-1,4],[3,2,-1],[2,1,-1]);
(%o2) matrix([1,-1,4],[3,2,-1],[2,1,-1])
```

Si se sabe una expresión para generar cada elemento se puede definir de forma parecida a las funciones. En primer lugar se definen cada uno de los elementos de la matriz. En nuestro ejemplo construiremos la matriz C cuyos elementos son $c_{i,j} = (i + j)^2$.

```
(%i3) C[i,j]:=(i+j)^2;
(%o3) C[i,j]:=(i+j)^2
```

Para generar la matriz usamos el comando⁵

```
genmatrix(C, No. de filas, No. de columnas)
```

donde debemos especificar el número de filas y columnas. Así para generar la matriz C de 4×3 usamos

```
(%i4) D:genmatrix(C,3,4);
(%o4) matrix([4,9,16,25],[9,16,25,36],[16,25,36,49])
```

El comando `submatrix` nos permite quedarnos con cualquier submatriz. Por ejemplo si de la matriz D anterior queremos eliminar la fila 2 y las columnas 1 y 4 escribimos

```
(%i5) E:submatrix(2,D,1,4);
(%o5) matrix([9,16],[25,36])
```

En general la sintaxis es

```
submatrix(filas a eliminar, matriz, columnas a eliminar)
```

Para multiplicar matrices con la definición habitual se usa el punto “.”

```
(%i6) A.E;
(%o6) matrix([59,88],[43,68])
```

Hay que tener cuidado pues $A * E$ también nos devuelve una salida que no tiene nada que ver con la anterior (`*` multiplica elemento a elemento).

```
(%i7) A*E;
(%o7) matrix([9,32],[50,36])
```

Así, por ejemplo, la operación

⁵El simple uso de `C[i,j]:=expresion` no genera la matriz sino un `array` como ya comentamos en la página 18.

```
(%i8) B.D;
(%o8) matrix([59,93,135,185],[14,34,62,98],[1,9,21,37])
```

está bien definida, mientras que

```
(%i9) B*D;
fullmap: arguments must have same formal structure.
-- an error. To debug this try: debugmode(true);
```

no lo está. Por supuesto que se pueden sumar matrices de las mismas dimensiones

```
(%i10) A+E;
(%o10) matrix([10,18],[27,37])
```

multiplicar por escalares (números)

```
(%i11) A-2*E;
(%o11) matrix([-17,-30],[-48,-71])
```

etc. Otra operación importante es elevar matrices a potencias enteras. Ello se hace con el comando A^{n} donde A es la matriz y n la potencia correspondiente. Es importante tener en cuenta que el símbolo $^$ se repite dos veces. Comprobar como ejercicio la diferencia entre A^n y A^{n} .

Podemos ver la k -ésima columna (o fila) de la matriz A con el comando `col(A,k)` (`row(A,k)`), respectivamente

```
(%i12) row(D,1);
(%o12) matrix([4,9,16,25])
(%i13) col(D,1);
(%o13) matrix([4],[9],[16])
```

En el caso de las filas también se puede invocar la orden $D[k]$ donde D es la variable que estamos usando para definir la matriz cuya fila queremos extraer. En nuestro ejemplo la orden $D[1]$ daría como resultado la misma salida (%o12) de antes.

Si queremos adicionar columnas o filas se usan los comandos `addcol` y `addrow`. Como ejercicio añadir una fila y una columna a la matriz D anterior (úse la ayuda de MAXIMA para ver la sintaxis de dichos comandos). Finalmente, podemos extraer el elemento (i,j) de una matriz D con el comando $D[i,j]$. Por ejemplo, para extraer los elementos $D_{1,1}$ y $D_{3,4}$ de la matriz D hacemos

```
(%i14) D[1,1]; D[3,4];
(%o14) 4
(%o15) 49
```

Dada una matriz A podemos calcular su transpuesta A^T con el comando `transpose(A)`

```
(%i16) E;
(%o16) matrix([9,16],[25,36])
(%i17) transpose(E);
(%o17) matrix([9,25],[16,36])
```

su determinante, con `determinant(A)`

```
(%i18) determinant(E);
(%o18) -76
```

su rango con `rank(A)`

```
(%i19) D; rank(D);
(%o19) matrix([4,9,16,25],[9,16,25,36],[16,25,36,49])
(%o20) 3
```

y su inversa A^{-1} con el comando `invert(A)`

```
(%i21) invA:invert(A);
(%o21) matrix([-1/3,2/3],[2/3,-1/3])
```

Nótese que la inversa es con respecto a la multiplicación habitual de matrices

```
(%i22) A.invA;
(%o22) matrix([1,0],[0,1])
```

Comprobar que $A \cdot invA$ no da la matriz identidad y repetir la operación intercambiando el orden $invA \cdot A$.

También podemos convertir nuestra matriz en una triangular mediante el método de eliminación de Gauss con el comando `triangularize`. Por ejemplo

```
(%i23) triangularize(B);
(%o23) matrix([1,-1,4],[0,5,-13],[0,0,-6])
(%i24) triangularize(D);
(%o24) matrix      ([4,9,16,25],[0,-17,-44,-81],[0,0,-8,-24])
```

Finalmente, podemos resolver el problema de autovalores $Av = \lambda v$, $v \neq 0$ con los comandos `eigenvalues` (encuentra los autovalores λ) y `eigenvectors` (encuentra los correspondientes autovectores v) de la siguiente forma:

```
eigenvalues(matriz)      eigenvectors(matriz)
```

donde la variable `matriz` es una matriz $N \times N$. Por ejemplo tenemos

```
(%i25) eigenvalues(B);
(%o25) [[3,-2,1],[1,1,1]]
(%i26) eigenvectors(B);
(%o26) [[[3,-2,1],[1,1,1]],[[[1,2,1]],[[1,-1,-1]],[[1,-4,-1]]]]
```

La salida de `eigenvalues` son dos listas: la primera corresponde a los autovalores distintos de la matriz, y la segunda a la multiplicidad algebraica de los mismos. La salida de `eigenvectors` son dos listas, la primera coincide con la salida de `eigenvalues`, es decir una lista compuesta por dos listas: la de los autovalores y la de su multiplicidad. La segunda lista es la lista de los autovectores correspondientes a cada autovalor. Así, por ejemplo, en el caso de la matriz B anterior vemos que tiene 3 autovalores distintos cada uno de los cuales tiene multiplicidad algebraica 1. Como ejercicio comprobar que en ejemplo anterior efectivamente $Bv - \lambda v = 0$ para cada uno de los autovalores y autovectores.

No siempre los autovalores son simples. Tal es el caso de la matriz F definida a continuación

```
(%i27) F:matrix([1,1,0],[0,1,0],[0,2,1]);
(%o27) matrix([1,1,0],[0,1,0],[0,2,1])
(%i28) eigenvalues(F);
(%o28) [[1],[3]]
(%i29) eigenvectors(F);
(%o29) [[[1],[3]],[[[1,0,0],[0,0,1]]]]
```

que tiene un único autovalor 1 con multiplicidad 3 al que le corresponden 2 autovectores. Conviene saber que la orden `eigenvalues` invoca al comando `solve` por lo que cuando este falla es preciso utilizar algún método numérico. Veamos un ejemplo.

Ante todo generamos una matriz de 5 por 5 con entradas enteras entre 0 y 0 y intentamos calcular los autovalores:

```
(%i30) kill(all)$
( %i1) C[i,j]:= random(10);
      D:genmatrix(C,5,5);
(%o1) C[i,j]:=random(10)
(%o2) matrix([6,9,0,8,8],[8,3,9,0,6],[3,2,3,0,9],[9,4,4,5,7],[6,2,1,0,4])
(%o3) eigenvalues(D);
"eigenvalues: solve is unable to find the roots of the characteristic
      polynomial."
(%o3) []
```

Así que tenemos que resolver el problema de forma numérica. Una opción es calcular el polinomio característico y encontrar sus raíces

```
(%i4) charpoly(D,x)$ expand(%);
(%o5) -x^5+21*x^4+58*x^3-396*x^2+2608*x-8182
(%i6) define(p(x),expand (charpoly (D, x)));
(%o6) p(x):=-x^5+21*x^4+58*x^3-396*x^2+2608*x-8182
(%i7) solve(p(x)=0,x);
(%o7) [0=-x^5+21*x^4+58*x^3-396*x^2+2608*x-8182]
(%i8) eign:allroots(p(x));
(%o8) [x=3.891131442362021*%i+0.8728783161156408,...,
      x=-6.448764124655439,22.96090841096163]
```

para lo cual, hemos usado la orden `allroots` que mencionamos en el apartado de resolución de ecuaciones. El próximo paso sería resolver los sistemas $(D - x_i I)v = 0$, $i = 1, 2, 3, 4, 5$, donde I es la matriz identidad cinco por cinco. Vamos a resolverlo para el caso del primer autovalor (para el resto es análogo) para lo que definimos la matriz $M = D - x_1 I$ donde para generar la matriz identidad I usamos el comando `identfor(matriz)` que genera una matriz identidad⁶ de las mismas dimensiones que `matriz`:

```
(%i9) eign1:second(eign[1]);
(%o9) 3.891131442362021*%i+0.8728783161156408
(%o10) M:D-eign1*identfor(D)$ b:[0,0,0,0,0]$
```

y preparamos el sistema de ecuaciones para resolverlo con `linsolve`:

⁶También podemos usar la orden `ident(n)`, siendo n el orden de la matriz.

```
(%i12) eq:M.[x1,x2,x3,x4,x5];
(%o12) matrix([8*x5+8*x4+9*x2+(5.127121683884-3.8911314423620*i)*x1],
..., [(3.1271216838843-3.8911314423620*i)*x5+x3+2*x2+6*x1])
(%i14) ratprint:false$
makelist(eq[k][1],k,1,5)$
(%i15) linsolve(%,[x1,x2,x3,x4,x5]);
(%o15) [x1=0,x2=0,x3=0,x4=0,x5=0]
```

De esta forma descubrimos que `linsolve` solo nos encuentra la solución trivial. Podemos usar un comando que contiene el paquete `linearalgebra` que no está explicado en el manual: la orden `linsolve_by_lu(A,b)` que encuentra la solución del sistema $Ax = b$ mediante la descomposición *LU* de la matriz cuadrada A :

```
(%i16) linsolve_by_lu(M,b);
(%o16) [matrix([0],[0],[0],[0],[0]),false]
```

cuya salida nuevamente es la trivial. ¿Qué hacer? está claro que hay que usar un método específico para la resolución del problema de autovalores. Para ello MAXIMA cuenta con la implementación de la librería LAPAC⁷ (*Linear Algebra PACKAGE*) originalmente escrita en FORTRAN.

Comenzaremos calculando los autovalores con la orden `dgeev`. En general `dgeev` tiene la siguiente sintaxis

```
dgeev(matriz, vec-der, vec-izq)
```

que lo que hace es calculas los autovalores de la matriz `matriz` y los autovectores por la derecha cuando `vec-der=true`, y los autovectores por la izquierda cuando `vec-izq=true`. Los autovectores por la derecha son las soluciones no triviales del sistema $Av_{der} = \lambda v_{der}$ mientras que por lo izquierdo lo son del sistema $v_{der}^H A = \lambda v_{der}^H$ donde v^H denota la transpuesta conjugada de v . La salida de `dgeev` una lista de tres elementos. El primer elemento es una lista con los autovalores. El segundo elemento es `false` o la matriz de autovectores por la derecha. El tercer elemento es `false` o la matriz de autovectores por la izquierda. Si escribimos `dgeev(matriz)` la salida es simplemente la lista de autovalores seguida de dos `false` (no muestra las matrices de los correspondientes autovectores). Así, tenemos

```
(%i17) load ("lapack")$%
(%i18) dgeev(D);
(%o18) [[22.96090841096165,-6.936883286868907,
3.891131442362015*i+0.8728783161156397,
0.8728783161156397-3.891131442362015*i,
3.23021824367599],false,false]
```

Ahora resolvemos el problema de autovalores por la derecha (nótese que hemos puesto la opción `false` para la tercera entrada). La salida la guardamos como lista de tres elementos por comodidad:

```
(%i19) [L,Vd,Vi]:dgeev(D,true,false)$
```

⁷Para más detalles visitar su web oficial: <http://www.netlib.org/lapack/>

Comprobamos ahora que la solución es correcta calculando la matriz $A v_k - \lambda_k v_k$ para cada una de las columnas de la matriz de autovalores. En esta salida mostramos el caso del tercer autovalor, que al ser complejo nos da soluciones complejas a las que calcularemos su valor absoluto:

```
(%i22) k:3$ expand(D.col(Vd,k)-L[k]*col(Vd,k)); abs(%);
(%o21) matrix([-2.886579864025407*10^-15*i-5.995204332975845*10^-15] ,
..., [3.552713678800501*10^-15*i+8.881784197001253*10^-16])
(%o22) matrix([6.653932544407802*10^-15], ..., [3.66205343881779*10^-15])
```

Como vemos las salidas son prácticamente cero (con 16 cifras significativas). Lo anterior lo podemos incluir en un bucle con la orden `for`. Dentro de sus muchas opciones, las cuales conviene consultar en el manual [1, §37], usaremos la siguiente:

```
for variable valor inicial step incremento thru valor final do orden
```

Si el paso es 1 se puede omitir. La orden `do` da paso a la operación que queremos repetir.

```
(%o23) for k:1 thru 5 do print(expand(D.col(Vd,k)-L[k]*col(Vd,k)));
```

La salida es muy engorrosa y no la incluiremos aquí.

Otra opción para comprobar si los cálculos son correctos es ver los valores de la norma de la matriz (o vector en este caso). Ello se puede hacer con el comando `dlangue` cuya sintaxis es

```
dlangue (norm, matriz)
```

donde `norm` es la opción de la norma a usar cuyas posibilidades son

1. `max` que calcula $\max_{i,j} |a_{i,j}|$, siendo $a_{i,j}$ los elementos de la matriz (no necesariamente cuadrada) `matriz`. Nótese que esta función no es una norma matricial.
2. `one_norm` que calcula el máximo de la suma de los valores absolutos de los elementos de cada columna.
3. `inf_norm` que calcula el máximo de la suma de los valores absolutos de los elementos de cada fila.
4. `frobenius` que calcula la norma de Frobenius de A, i.e., la raíz cuadrada de la suma de los cuadrados de los elementos de la matriz.

Así, por ejemplo podemos calcular los valores del máximo elemento en valor absoluto:

```
(%i24) for k:1 thru 5 do
    print( dlangue(max,expand(D.col(Vd,k)-L[k]*col(Vd,k))) )$
```

1.77635683940025*10^-14" "
7.549516567451065*10^-15" "
6.653932544407802*10^-15" "
6.206335383118182*10^-15" "
5.995204332975845*10^-15" "

o la norma Frobenius:

```
(%i25) for k:1 thru 5 do
      print(dlange(frobenius,expand(D.col(Vd,k)-L[k]*col(Vd,k))))$  

2.162855789913095*10^-14  

8.826098480675078*10^-15  

Maxima encountered a Lisp error:  

Condition in MACSYMA-TOP-LEVEL [or a callee]: INTERNAL-SIMPLE-ERROR:  

Wrong type error  

Automatically continuing.  

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

En este caso obtenemos un error debido a que los valores para los autovectores tercero y cuarto son complejos. Para subsanarlo simplemente repetimos la orden tomando primero valores absolutos:

```
(%i26) for k:1 thru 5 do print(
      dlange(frobenius,abs(expand(D.col(Vd,k)-L[k]*col(Vd,k)))))$  

2.162855789913095*10^-14  

8.826098480675078*10^-15  

8.297013916595377*10^-15  

8.199882085642889*10^-15  

8.004055474699982*10^-15
```

Como se ve son casi cero.

Hagamos ahora la comprobación matricial para lo cual creamos la matriz diagonal de los autovalores con `diag_matrix(lista)`:

```
(%i28) Dia : apply (diag_matrix, L)$
      expand((D-Dia).Vd);
(%o28) matrix([-2.664535259100375*10^-15, 21.49279947090898, . . .], . . .)
(%i29) dlange(max,%);
(%o29) 21.49279947090898
```

que como vemos no da la matriz nula. ¿Qué ocurre? pues ya hemos comprobado que el problema estaba bien resuelto.

La razón, como no, es la no commutatividad del producto matricial. Hagamos un breve paréntesis en nuestro estudio para mostrar lo anterior. Para ello definiremos una matriz general de tres por tres y la multiplicaremos por la izquierda y por la derecha por una diagonal con elementos distintos:

```
(%i30) A:matrix([a1,a2,a3],[b1,b2,b3],[c1,c2,c3]);
(%o30) matrix([a1,a2,a3],[b1,b2,b3],[c1,c2,c3])
(%i31) Dd:diag_matrix(d1,d2,d3);
(%o31) matrix([d1,0,0],[0,d2,0],[0,0,d3])
(%i32) Dd.A;
(%o32) matrix([a1*d1,a2*d1,a3*d1],[b1*d2,b2*d2,b3*d2],[c1*d3,c2*d3,c3*d3])
(%i33) A.Dd;
(%o33) matrix([a1*d1,a2*d2,a3*d3],[b1*d1,b2*d2,b3*d3],[c1*d1,c2*d2,c3*d3])
```

Como se ve el resultado es diferente. Se puede demostrar que si queremos comprobar matricialmente nuestro problemas de autovalores lo que tenemos que calcular no es $(D \cdot \text{Dia}) \cdot Vd = D \cdot Vd - \text{Dia} \cdot Vd$ sino $D \cdot Vd - Vd \cdot \text{Dia}$:

```
(%i34) expand(D.Vd-Vd.Dia);
(%i35) dlange(max,%);
(%o35) 1.77635683940025*10^-14
```

El paquete `lapack` tiene otras muchas utilidades como por ejemplo rutinas para resolver el problema de valores singulares de matrices, descomposición *QR* de una matriz, entre otros. Para más detalle el lector puede consultar el manual [1].

Llegados a este punto vale la pena destacar que podemos construir matrices cuyos elementos sean objetos cualesquiera, por ejemplo, funciones:

```
(%i30) kill(all)$
(%i1) define(polm(x),matrix([1,x],[x,x^2]));
(%o1) polm(x):=matrix([1,x],[x,x^2])
```

Dicho polinomio matricial podemos⁸ derivarlo, integrarlo, ...

```
(%i2) diff(polm(x),x);
(%o3) matrix([0,1],[1,2*x])
```

aplicarle funciones

```
(%i4) exp(polm(x));
(%o4) matrix([\%e,\%e^x],[\%e^x,\%e^x^2])
```

Nótese que todas estas operaciones las hace elemento a elemento, lo cual es importante a tener en cuenta a la hora de trabajar con funciones de matrices que se definen de manera completamente distinta. Un ejemplo de lo anterior es el cálculo de exponencial de una matriz que mostraremos en el apartado 6.2.

Para terminar este apartado vamos mencionar algunos otros comandos a tener en cuenta cuando se trabaja con matrices.

En primer lugar, aunque no es recomendable, podemos introducir una matriz elemento a elemento interactivamente. Ello se hace con el comando `entermatrix(n,m)`, donde *n* y *m* son las dimensiones de la matriz.

```
(%i5) kill(all);
(%i1) entermatrix(2,2);
Is the matrix 1. Diagonal 2. Symmetric 3. Antisymmetric 4. General
Answer 1, 2, 3 or 4 : 2;
Row 1 Column 1: 3;
Row 1 Column 2: 4;
Row 2 Column 2: 1;
Matrix entered.
(%o1) matrix([3,4],[4,1])
```

⁸Entenderemos que la derivada de un vector o una matriz es la derivada término a término y lo mismo

Nótese que si pedimos a MAXIMA escribir una matriz cuadrada este decide preguntar por varios tipos especiales de matrices, no así para matrices en general. Probar que ocurre si queremos introducir una matriz que no sea cuadrada, como por ejemplo, `entermatrix(2,3)`.

A parte de las matrices diagonales (que construimos con `diag_matrix(lista)`) y la identidad `ident` que ya hemos usado antes hay otras matrices especialmente importantes como es el caso de las matrices nulas $n \times m$ y las proporcionales a la identidad que se generan mediante los comandos `zeromatrix(n,m)` y `diagmatrix(n,x)`, respectivamente y donde para estas últimas n es el orden de la matriz y x el factor de proporcionalidad:

```
(%i2) ceroM:zeromatrix(3,2);
(%o2) matrix([0,0],[0,0],[0,0])
(%i3) diagmatrix(2,x);
(%o3) matrix([x,0],[0,x])
```

La manipulación de las matrices es muy sencilla. Para cambiar un elemento determinada basta con redefinirlo.

```
(%i4) ceroM[1,2]:=1$ ceroM;
(%o5) matrix([0,1],[0,0],[0,0])
```

Si por ejemplo queremos cambiar la segunda fila de la matriz `ceroM` basta con definir la lista correspondiente

```
(%i6) ceroM[2]:[1,2]$ ceroM;
(%o7) matrix([0,1],[1,2],[0,0])
```

Para cambiar una columna podemos usar la idea anterior junto a la operación transposición:

```
(%i8) ceroM:transpose(ceroM)$ ceroM[2]:[1,1,1]$ 
      ceroM:transpose(ceroM);
(%o10) matrix([0,1],[1,1],[0,1])
```

Otro tipo de matrices de gran importancia son las matrices $n \times m$ que tienen un único elemento no nulo en la posición (i,j) y cuyo valor sea, digamos x . Este tipo de matrices se puede generar con el comando `ematrix(m,n,x,i,j)`

```
(%i11) ematrix(3,2,x,1,2);
(%o11) matrix([0,x],[0,0],[0,0])
```

Otros comandos útiles son

- `minor(A,i,j)` que calcula el menor (i,j) de la matriz A (es decir elimina la fila i y la columna j de A)
- `adjoint(A)` que calcula la matriz adjunta de A
- `setelmx (x, i, j, A)` que asigna el valor x a la entrada (i,j) de la matriz A

- `mattrace(A)` calcula la traza de la matriz A (requiere que se cargue el paquete `nchrpl`).
- `gramschmidt(A)` realiza el proceso de ortogonalización de Gram-Schmidt sobre A , que puede ser una lista de listas (vectores) o una matriz. En este último caso toma como vectores las filas. Requiere el paquete `eigen`.

Este último es especialmente útil para construir bases ortonormales

```
(%o12) kill(all)$  
(%i1) load("eigen")$  
(%i2) A:matrix([2,0,1],[3,0,0],[5,1,1]);  
(%o2) matrix([2,0,1],[3,0,0],[5,1,1])  
(%i3) ss:ratsimp(gramschmidt(A));  
(%o3) [[2,0,1],[3/5,0,-6/5],[0,1,0]]
```

Comprobemos que efectivamente los vectores son ortogonales para lo cual calculamos el producto escalar entre ellos usando la orden `innerproduct(u,v)` donde u y v son los correspondientes vectores:

```
(%i4) innerproduct(ss[1],ss[2]); innerproduct(ss[1],ss[3]);  
      innerproduct(ss[2],ss[3]);  
(%o4) 0  
(%o5) 0  
(%o6) 0
```

Finalmente, si queremos construir los vectores unitarios, i.e., de norma 1, usamos el comando `unitvector(v)` (para que funcione hay que cargar el paquete `eigen`)

```
(%i7) unitvector(ss[1]);unitvector(ss[2]);unitvector(ss[3]);  
(%o7) [2/sqrt(5),0,1/sqrt(5)]  
(%o8) [1/sqrt(5),0,-2/sqrt(5)]  
(%o9) [0,1,0]
```

En este caso como la matriz es real también podríamos haber usado como producto escalar las secuencias `ss[1].ss[2]; ss[1].ss[3]; ss[2].ss[3]`; lo que no es posible hacer en el caso complejo. Como ejemplo compárese las salidas de ambas opciones para la matriz

$$\begin{pmatrix} i+2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1-2i & 1 \end{pmatrix}.$$

Para más detalle sobre como usar MAXIMA para resolver problemas de álgebra lineal se recomienda consultar los apartados §23 (Matrices y Álgebra lineal), 65 (lapack) y 68 (linearalgebra) del manual de MAXIMA [1].

4.3. Tratamiento de datos.

MAXIMA tiene un paquete muy completo para el tratamiento de datos: el paquete `descriptive`. Antes de pasar a ver algunas de sus posibilidades vamos a ver como se introducen los datos. Normalmente se tiene un fichero ASCII con una lista de los datos ordenados por columnas. En nuestro ejemplo es el fichero `datos80.dat` que contiene los datos de crecimiento de la población mundial según datos de la Oficina del Censo de los Estados Unidos que luego usaremos en algunos ejercicios. Para cargarlo lo primero que hay que asegurarse es que MAXIMA reconoce dónde está para lo cual usamos el comando `file_search_maxima` que ya comentamos en la página 13. A continuación cargamos el paquete `descriptive`

```
(%i1) load (descriptive)$
```

e introducimos los datos.

El fichero `datos80.dat` de nuestro ejemplo consta de cuatro columnas con 31 elementos cada una. Para poder incluir los datos en la sesión de MAXIMA tenemos que usar una combinación de los comandos `file_search` y `read_matrix`. El primero encuentra el fichero de datos y el segundo lo lee e importa los datos como una matriz⁹ (si queremos que los importe como una lista, lo cual es muy cómodo para ficheros con una sola columna de datos hay que usar `read_list`).

```
(%i3) mm:read_matrix (file_search ("datos80.dat"))$
```

A partir de este momento podemos trabajar con la variable `mm` como una matriz, por lo que podemos usar cualquiera de los comandos válidos para el tratamiento de matrices. Aquí nos interesará tener los vectores columna 1 y 2 de los datos.

```
(%i4) xx:col(mm,1)$ yy:col(mm,2)$
```

Por ejemplo, si queremos saber el número de filas usamos

```
(%i5) length(mm);length(xx);
(%o5) 31
(%o6) 31
```

o que nos muestre la fila 31

```
(%i7) row(mm,31);row(xx,31);
(%o7) matrix([1980,4452557135,1.7,76294721])
(%o8) matrix([1980])
```

Podemos calcular la media y la varianza de los datos de una determinada columna con el comando `mean` y `var`, respectivamente

```
(%i9) float( mean( col(mm,3) ) );
(%o9) [1.845483870967742]
(%i10) var(col(mm,3));
(%o10) [0.046650572320499]
```

⁹Para MAXIMA el formato de los datos del fichero `datos80.dat` es una matriz de 4 cuatro columnas y 31

Dichos comandos también funcionan cuando la entrada es una matriz, dando como resultado el cálculo de la media o la varianza de cada una de las columnas. Por ejemplo

```
(%i12) float(mean( mm ));  
(%o12) [1965,3.41281317235483*10^9,1.8454838709677,6.36418213870967*10^7]
```

calcula la media de cada una de las columnas.

Si queremos representar los datos gráficamente tenemos que convertir cada columna en una *lista* de datos. Para ello usamos el comando `makelist` que ya hemos en la página [14](#) y cuya sintaxis es

```
makelist( expr , k , k0 , kfin )
```

y que nos genera una lista con los valores de la expresión `expr` para cada `k` desde `k0` inicial hasta `kfin` final

```
(%i10) lx:makelist(mm[k,1] ,k,1,length(mm))$  
ly:makelist(mm[k,2] ,k,1,length(mm))$
```

y usamos cualquiera de los comandos para representar gráficas, como por ejemplo `plot2d`

```
(%i14) plot2d([discrete,lx,ly],[x,1945,1985],[logy],[style,points,lines],  
[color, red ], [point_type, circle],[legend, "datos"],  
[xlabel, "año"], [ylabel, "individuos"])$  
(%t14) << Graphics >>
```

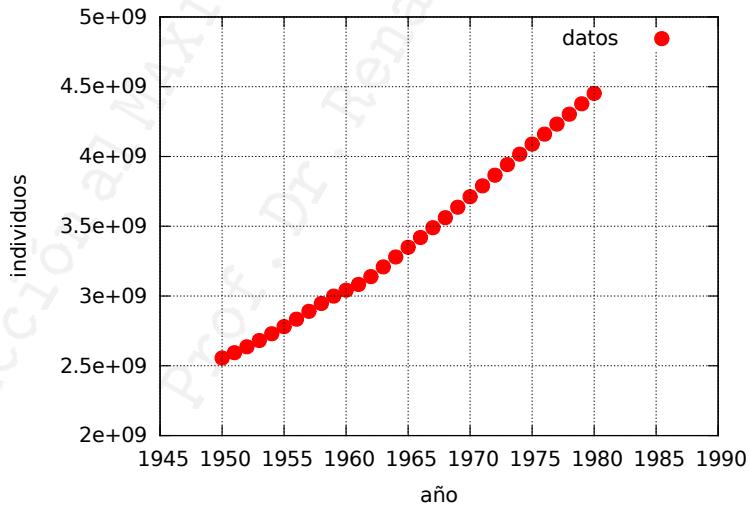


Figura 4.2: Datos de la población del censo de la Oficina del Censo de los Estados Unidos.

De la gráfica [4.2](#) parece razonable pensar que los datos se pueden aproximar mediante una recta de regresión. Imaginemos que queremos aproximar los datos de la segunda

columna mediante la expresión $y = ax + b$, donde los valores de a y b los vamos a determinar a partir de los propios datos de forma tal que la desviación media cuadrática sea mínima, i.e., usando en método de los mínimos cuadrados. Para ello MAXIMA cuenta con el paquete lsquares

```
(%i15) load (lsquares)$
```

Ante todo tenemos que generar una matriz de dos columnas, la primera serán las entradas x y la segunda las y

```
(%i16) datos:submatrix(mm,3,4)$
      length(datos);
(%o17) 31
```

A continuación usamos el comando lsquares_mse(datos, var , ec) que genera la expresión del error cuadrático medio de los datos contenidos en la matriz datos según la ecuación ec en la lista de variables var. En nuestro caso tenemos dos variables así que escribimos

```
(%i18) kill(a,b);
      mse : lsquares_mse (datos, [x, y], y= a+b*x);
(%o18) done
(%o19) sum((datos[i,2]-b*datos[i,1]-a)^2,i,1,31)/31
```

Ahora usamos el comando lsquares_estimates_approximate que calcula numéricamente los valores de los parámetros (en nuestro ejemplo a y b) que minimizan el error cuadrático medio obtenido mediante el comando anterior lsquares_mse

```
(%i20) sol:lsquares_estimates_approximate(mse, [a,b]);
*****
N=      2   NUMBER OF CORRECTIONS=25
INITIAL VALUES
F=  1.198414876567487D+19  GNORM=  1.342269164073287D+13
*****
I  NFN  FUNC          GNORM          STEPLENGTH
1    11  7.764309441907280D+18  1.072344568144780D+13  2.603985916947699D-08
2    12  3.191414544668561D+17  5.121509730845318D+06  1.000000000000000D+00
3    30  2.463241440306616D+17  5.604701804329961D+10  2.290649224500000D+10
4    31  2.658193297849049D+15  1.784676183886853D+10  1.000000000000000D+00
5    32  2.636725183420313D+15  6.894361531609709D+08  1.000000000000000D+00
6    33  2.636694396735505D+15  6.830942264400961D+04  1.000000000000000D+00
THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
(%o20) [[a=-1.2359822742463164*10^11,b=6.4636661890076451*10^7]]
```

Si queremos que MAXIMA intente resolver analíticamente el problema —lo cual no siempre es capaz de hacer— hay que usar en vez de lsquares_estimates_approximate, el comando lsquares_estimates, cuya sintaxis es

```
lsquares_estimates(datos,lista de variables,ecuacion,lista de parametros)
```

donde datos es una matriz de datos, lista de variables, es una lista con el nombre de filas.

las variables correspondiente a cada columna de la matriz de datos, lista de parametros es una lista con los parámetros del modelo y ecuacion es la ecuación que vamos usar para el ajuste de datos.

```
(%i21) sol:lsquares_estimates(datos, [x,y],y= a+b*x,[a,b]);
(%o21) [[a=-15326178001631/124,b=40074724771/620]]
```

Nótese que la salida son números racionales (y no en coma flotante), es decir la solución es exacta. Finalmente dibujamos la gráfica de la función $y = ax + b$ y la comparamos con nuestros datos –ver gráfica de la izquierda en la figura 4.3—

```
(%i21) aa:rhs(sol[1][1]);bb:rhs(sol[1][2]);
(%o21) -1.2359822742463164*10^11
(%o22) 6.4636661890076451*10^7
(%i23) define(f(x), aa+bb*x);
(%o23) f(x):=6.4636661890076451*10^7*x-1.2359822742463164*10^11
(%i24) wxplot2d([[discrete,lx,ly],f(x)], [x,1945,1987],
               [style,points,[lines,3]], [y,2*10^9,5*10^9], [yticks,5*10^8],
               [color, red, blue], [point_type,bullet],
               [legend,"datos", "curva de ajuste"],
               [ylabel, "individuos"], [xlabel, ""])$
(%t24) << Graphics >>
```

Si queremos representar los gráficos en escala logarítmica incluimos [logy], [logx], o ambas en la lista de opciones (en nuestro caso solo la usaremos para el eje de las y). Los resultados están representados en la gráfica de la derecha de la figura 4.3.

```
(%i25) wxplot2d([[discrete,lx,ly],f(x)], [x,1945,1987], [color,red,blue],
               style,points,[lines,3]], [point_type,bullet],
               [legend,"datos","curva de ajuste"], [logy],
               [ylabel, "individuos"], [xlabel, ""])$
(%t25) << Graphics >>
```

Para terminar mostraremos como podemos crear diagramas de pastel e histogramas a partir de listas de datos discretos. Para ello comenzaremos generando tres listas que corresponden a las calificaciones de dos exámenes parciales de una asignatura (que generaremos usando el comando `random` que genera números aleatorios) y a la media entre ambas (usamos el comando `floor` para calcular la parte entera¹⁰ de la semisuma). A continuación usamos el comando `barsplot` que cuenta las coincidencias y dibuja cada uno de los rectángulos (la lista de datos ha de ser números enteros o datos no numéricos). Comenzamos con las notas del examen:

```
(%i26) l1:makelist(random(10),k,1,100)$
      l2:makelist(random(10),k,1,100)$
      l3: floor((l1+l2)/2)$
(%i29) barsplot( l1,l2,l3, box_width=1, fill_density = 1,
                  bars_colors=[blue,green,red], xlabel="", ylabel="",
                  sample_keys=["Parcial 1", "Parcial 2", "Final"])$
```

¹⁰Véase también los comandos `ceiling` y `entier`.

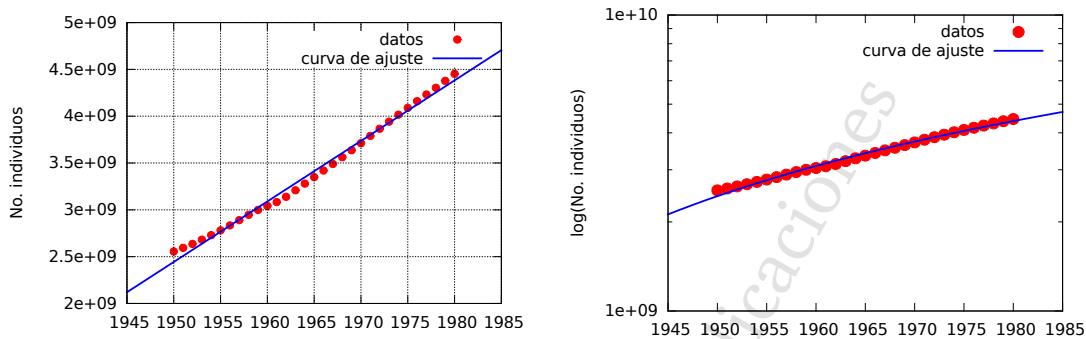


Figura 4.3: Comparación de los datos y la función $y = ax + b$ calculado por el método de los mínimos cuadrados. A la izquierda en escala normal, a la derecha en escala logarítmica en y .

La gráfica la podéis ver en la figura 4.4.

Dibujemos ahora un diagrama de pastel con las notas de un examen de matemáticas del primer curso (ver gráfica izquierda de la figura 4.5):

```
(%i30) n:makelist([suspenso,aprobado,notable,sobresaliente]
                  [random(4)+1], k,1,50)$
(%i31) piechart (n,title="Notas Mates Curso 1",dimensions=[1000,1000],
                  xtics=none,ytics=none,axis_top=false,axis_right=false,
                  axis_left=false,axis_bottom=false,yrange=[-1.4,1.4],
                  xrange = [-1.34, 1.34])$
```

Ahora generamos tres listas con las notas cualitativas del examen de una asignatura durante tres cursos y las dibujamos (ver gráfica derecha de la figura 4.5):

```
(%i32) ne1:makelist([suspenso,aprobado,notable,sobresaliente] [random(4)+1],
                  k,1,50)$
      ne2:makelist([suspenso,aprobado,notable,sobresaliente] [random(4)+1],
                  k,1,50)$
      ne3:makelist([suspenso,aprobado,notable,sobresaliente] [random(4)+1],
                  k,1,50)$
(%i35) barsplot(ne1,ne2,ne3,
                 title = "notas Mates",ylabel="Notas",groups_gap=4,fill_density=0.5,
                 bars_colors = [black,blue,red], ordering=ordergreatp,
                 dimensions=[1000,1200],yrange=[0,20],font="Arial", font_size=15,
                 sample_keys=["Curso 1","Curso 2","Curso 3"])$
```

Si queremos analizar un elevado número de datos numéricos distribuidos en un rango muy grande, o simplemente si queremos que la longitud de las clases sean mayores, podemos distribuirlos en un número de intervalos dado usando el comando

```
continuous_freq(datos,No. de intervalos)
```

Por ejemplo, imaginemos que queremos presentar los resultados de un examen de un grupo de cien alumnos. Para crear las notas usaremos un generador de números aleatorios

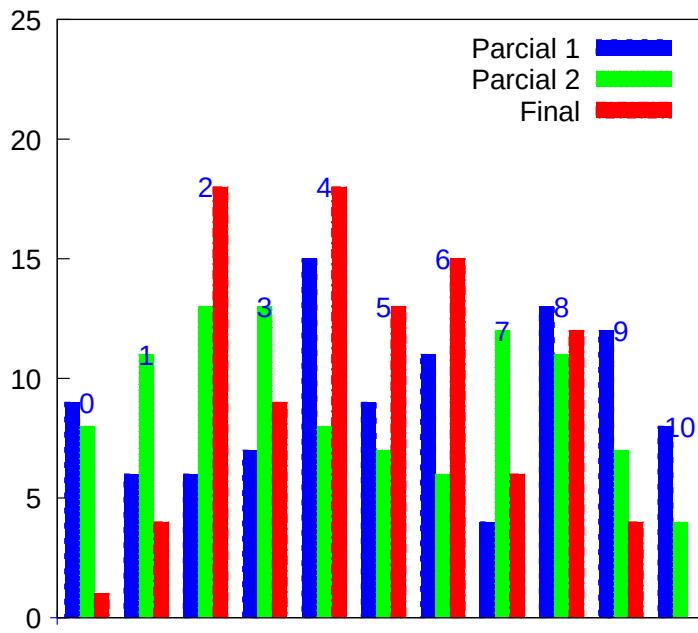


Figura 4.4: Histograma con las notas del examen de la salida (%i29).

```
(%i36) mm3:makelist(random(11),k,1,100)$
```

Podemos representar los datos usando un `barsplot`

```
(%i37) barsplot(mm3, box_width=1,fill_density=1,
bars_colors=[blue],sample_keys=[["Nota del examen"]])$
```

Esta claro que un `barsplot` no es una buena opción en el caso de que queramos agrupar más los datos, digamos en 5 clases. Para ello podemos usar la orden `histogram` que construye el histograma con los datos distribuidos en el número de clases que fijemos

```
(%i38) histogram(mm3,nclasses=10,fill_color=blue,fill_density = 0.6)$
(%i39) histogram(mm3,nclasses=5,fill_color=blue,fill_density = 0.6)$
```

Como se ve de los gráficos generados por los comandos anteriores, incluso para representar los datos sin agrupar es mucho más conveniente, al menos visualmente, usar la orden `histogram` en vez de `barsplot`.

Si queremos representar las notas del examen en un diagrama de pastel escribimos la secuencia

```
(%i40) piechart( mm3 , title = "Notas del examen",dimensions = [500,500],
xtics=none,ytics=none,axis_top=false, axis_right=false,
axis_left=false,font="Arial",font_size=20,axis_bottom=false,
yrange = [-1.4,1.4] , xrange=[-1.34,1.34])$
```

Un vistazo a la gráfica izquierda de la figura 4.7 nos indica claramente que es mejor dividir los datos en clases más grandes. Agrupemos los datos en cinco clases

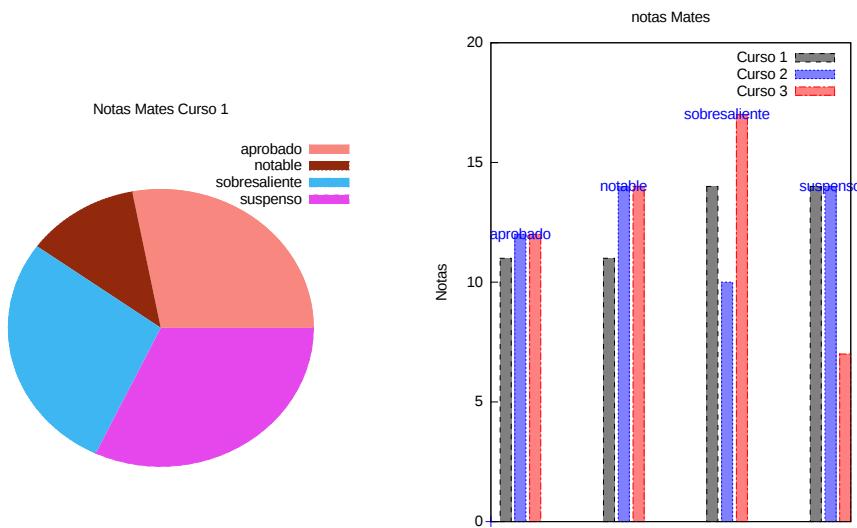


Figura 4.5: Gráfico de la entrada (%i31) de un diagrama de pastel, y del histograma de la entrada (%i35), respectivamente.

```
(%i41) clases:5$  
    contar: continuous_freq (mm3, clases);  
    pies:contar[2];  
    sum(pies[k],k,1,length(pies));  
(%o42) [[0,2,4,6,8,10],[23,13,29,13,22]]  
(%o43) [23,13,29,13,22]  
(%o44) 100
```

La primera lista de la variable contar nos da las clases: la primera entrada contiene los extremos de los subintervalos en que se han dividido los datos que en nuestro caso son en [0,2] (ambos incluidos), el (2,4], (4,6], (6,8] y (8,10]. La segunda lista nos dice cuantos datos hay en cada uno de los intervalos. Así en el primer intervalo (clase) hay 23, en el segundo 13, etc. Lo que vamos a hacer para dibujar el diagrama de pastel es construir una nueva lista que contenga los valores del 1 al 5 (nuestras clases) repetidos tantas veces como aparecen en la lista pies (*¿por qué?*) para luego usar la orden `continuous_freq (lista,[inic,fin,No. clases])` que genera dos listas, la primera contiene los intervalos definidos por No. clases comenzando por `inic` y terminando por `fin`, y la segunda el número de elementos que contiene cada intervalo. Así tendremos

```
(%i45) kill(aa)$ listapies:[]$  
for l:1 thru clases step 1 do aa[l]:makelist(l,k,1,pies[l])$  
for l:1 thru clases step 1 do listapies:append(listapies,aa[l])$  
listapies$  
continuous_freq (listapies,[0,clases,clases]);  
(%o50) [[0,1,2,3,4,5],[23,13,29,13,22]]
```

Finalmente, dibujamos el diagrama de pastel (ver la gráfica derecha de la figura 4.7):

```
(%i51) piechart(listapies,title ="Notas del examen",dimensions=[500,500],  
    xtics=none,ytics=none,axis_top=false,axis_right=false,
```

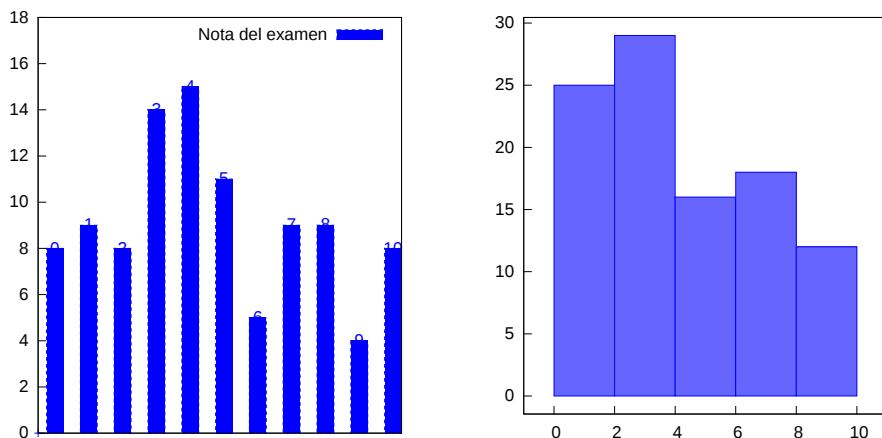


Figura 4.6: Representación de las notas del examen usando `barplot` y `histogram`, respectivamente.

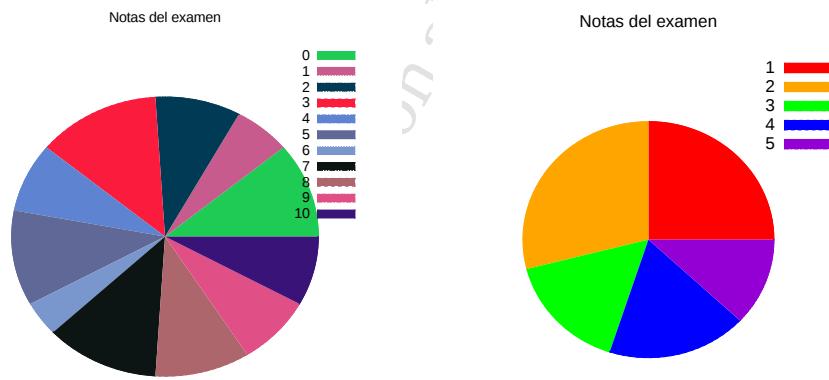


Figura 4.7: Representación de las notas del examen usando diagramas de pastel. A la izquierda sin redefinir las clases y a la derecha después de agrupar los datos convenientemente.

```
axis_left=false,font="Arial",font_size=20,axis_bottom=false,
yrange=[-1.4,1.6],xrange=[-1.34,1.34],
sector_colors=[red,orange,green,blue,dark_violet])$
```

Desgraciadamente los comandos `histogram` y `piechart` no tienen muchas opciones¹¹. Véase el manual [1] para más detalles.

4.4. El problema de interpolación.

Veamos ahora otro problema relacionado con el tratamiento de datos: el problema de interpolación.

¹¹Como comentario extra hemos de destacar que en el momento de escribir estas notas ha sido imposible exportar a pdf la salida de `barplot`.

La interpolación consiste en establecer la función original, que muchas veces es desconocida, a partir de un número finito de sus valores¹², conocidos de antemano y que están contenidos en cierto dominio o intervalo fijo. Así tenemos el siguiente problema: Dado un intervalo (a, b) , comúnmente denominado *dominio de interpolación*, y Sean $x_i, i = 1, 2, \dots, n$ ciertos valores prefijados del mismo a los cuales le corresponden los valores $y_i, i = 1, 2, \dots, n$. Llamaremos *función de interpolación* o *interpolante* a la función $f(x)$ que coincide en los puntos x_i con los valores y_i , osea $f(x)$ es tal que:

$$f(x_i) = y_i, \quad i = 1, 2, \dots, n. \quad (4.4.1)$$

Existen varias formas de elegir la interpolante $f(x)$. El modo más sencillo es suponer que $f(x)$ es un polinomio, en cuyo caso se puede comprobar que dados $n + 1$ puntos distintos siempre existe un único polinomio de interpolación de orden n que cumple con (4.4.1). En este caso, el problema de interpolación se le denomina *problema de interpolación polinómica*. Si suponemos que $f(x)$ es una función racional entonces tendremos un problema de interpolación racional; si $f(x)$ está definida a trozos, un problema de Spline-interpolación o interpolación por splines, etc. En este apartado discutiremos algunos de estos problemas.

4.4.1. El polinomio de interpolación de Lagrange.

Comenzaremos por la interpolación polinómica. Busquemos al polinomio de interpolación de la forma:

$$P_{int}(x) = \sum_{k=0}^n c_k x^k.$$

Puesto que $P_{int}(x_i) = y_i$ (ver (4.4.1)), obtenemos un sistema de n ecuaciones con n incógnitas (los coeficientes c_k) cuyo determinante siempre es diferente de cero (es un determinante de Vandermonde cuyos coeficientes son todos distintos) y por tanto tiene una única solución. Aunque formalmente podemos calcular el polinomio interpolador por el método antes descrito, ello es tremadamente ineficiente si el número de puntos a interpolar es alto (hay que invertir matrices grandes con pocos ceros). Una forma de evitar esto es usar el método de interpolación de Lagrange que consiste en encontrar un polinomio, que denotaremos por $L_m(x)$, de orden a lo más $m = n$, que coincide en los puntos x_i con los valores y_i , osea $L_m(x)$ es tal que:

$$L_m(x_i) = y_i, \quad i = 1, 2, \dots, n + 1. \quad (4.4.2)$$

A dicho polinomio $L_m(x)$ le llamaremos *polinomio de interpolación de Lagrange*. Se puede comprobar que el polinomio de interpolación de Lagrange tiene la forma:

$$P_{int}(x) \equiv L_n(x) = \sum_{k=1}^{n+1} y_k l_k(x), \text{ con } l_k(x) = \prod_{\substack{i=1 \\ i \neq k}}^{n+1} \frac{x - x_i}{x_k - x_i}. \quad (4.4.3)$$

Para ello es suficiente utilizar el hecho de que $l_k(x_i) = \delta_{ki}$, donde δ_{ki} es el símbolo de Kronecker: $\delta_{ki} = 1$ si $i = k$ y 0 si $i \neq k$. Es sencillo comprobar que el polinomio de interpolación de Lagrange (4.4.3) cumple con las condiciones (4.4.2) y por tanto es realmente una de las soluciones del problema de interpolación planteado. Además, por la unicidad del problema de interpolación polinómica, toda solución polinómica del mismo coincide con él.

¹²En general este conjunto de valores pueden aparecer como resultado de un experimento o bien son

Numéricamente el polinomio de interpolación se puede calcular mediante la fórmula de Newton. Describiremos brevemente este algoritmo.

Busquemos el polinomio de interpolación de la siguiente forma:

$$p_n(x) = a_0 + a_1(x - x_1) + \cdots + a_n(x - x_1)(x - x_2) \dots (x - x_n), \quad (4.4.4)$$

entonces los coeficientes a_k son la solución del siguiente sistema:

$$\begin{aligned} y_1 &= a_0 \\ y_2 &= a_0 + a_1(x_2 - x_1) \\ &\vdots && \vdots \\ y_{n+1} &= a_0 + a_1(x_{n+1} - x_1) + a_2(x_{n+1} - x_1)(x_{n+1} - x_2) + \\ &&& + \cdots + a_n(x_{n+1} - x_1)(x - x_2) \dots (x_{n+1} - x_n). \end{aligned}$$

De la fórmula anterior se sigue que al agregar un nuevo punto de interpolación (x_{n+2}, y_{n+2}) el polinomio de interpolación cumple con la propiedad:

$$p_{n+1}(x) = p_n(x) + a_{n+1}(x - x_1)(x - x_2) \dots (x - x_n)(x - x_{n+1}).$$

Resolviendo el sistema encontramos los coeficientes:

$$a_0 = y_1, \quad a_1 = \frac{y_2 - y_1}{x_2 - x_1}, \quad a_2 = \left[\frac{y_3 - y_1}{x_3 - x_1} - \frac{y_2 - y_1}{x_2 - x_1} \right] \frac{1}{x_3 - x_2}, \quad \dots$$

A estos coeficientes a_k se les llama *diferencias divididas de orden k* y se les denota por: $a_k = [y_1, y_2, \dots, y_{k+1}]$. Comparando la fórmula (4.4.4) con (4.4.3) concluimos que el coeficiente c_n de la potencia x^n coincide con a_n , por tanto

$$a_n = [y_1, y_2, \dots, y_{n+1}] = \sum_{k=1}^{n+1} \frac{y_k}{\prod_{\substack{i=1 \\ i \neq k}}^{n+1} (x_k - x_i)}.$$

Si los números y_k son los valores de cierta función $f(x)$ en los puntos x_k entonces (4.4.4) se puede escribir de la forma:

$$P_n(f, x) = \sum_{k=1}^{n+1} [f(x_1), f(x_2), \dots, f(x_{k+1})] (x - x_1)(x - x_2) \dots (x - x_k).$$

A esta forma de escribir el polinomio de interpolación se le conoce por *serie finita de Newton* para la función $f(x)$.

Veamos un ejemplo

```
(%i52) kill(all)$
(%i1) load(interp)$ load(draw)$
(%i3) datos: [[1,2], [3/2,1], [2,0], [5/2,-1], [3,1/2], [4,1]]; length(datos);
(%o3) [[1,2], [3/2,1], [2,0], [5/2,-1], [3,1/2], [4,1]]
(%o4) 6
(%i7) lagrange(datos)$ /* Polinomio de Lagrange */
ratsimp(%)$ define(lag(x),%);
(%o7) lag(x):=-(52*x^5-580*x^4+2435*x^3-4835*x^2+4620*x-1764)/36
```

El polinomio de interpolación de Lagrange no siempre es una buena aproximación. Se puede comprobar que al aumentar el número de puntos de interpolación este deja de parecerse a la función original, especialmente fuera del intervalo definido por el conjunto de puntos. Además hay que tener en cuenta otro problema añadido y es que cuando aumentamos en número de puntos, automáticamente aumenta el grado del polinomio y es un hecho bien conocido que la evaluación numérica de los valores de un polinomio de grado alto en coma flotante puede ser muy inestable numéricamente hablando.

Por ejemplo, dada la función $f(x) = x + \sin(x)$ construimos la lista de las x y $f(x)$ en el intervalo $[0, \pi]$ con 10 puntos.

```
(%i8) define(f(x),x+sin(x));
      kill(x)$ a:0$ b:%pi$ Nu:10$ h:(b-a)/(Nu)$
      x[0]:a$ x[n]:=x[0]+n*h;
      pp10:makelist( float([x[k],f(x[k])]),k,0,Nu)$
(%o9) f(x):=sin(x)+x
(%o16) x[n]:=x[0]+n*h
```

y la dibujamos

```
(%i17) define(lag10(x),lagrange(pp10))$
(%i18) wxdraw2d(color=red, key="polinomio de Lagrange",line_width=3,
      explicit(lag10(x),x,0,%pi),yrange=[-.2,4.3],
      point_type = filled_diamant, point_size = 3,
      color = blue, key = "Puntos", points(pp10))$
(%t18) (Graphics)
```

La gráfica la vemos en la figura 4.8 (izquierda). No obstante si aumentamos el intervalo está claro que el polinomio de Lagrange se aleja de la función original tal y como se ve en la figura 4.8 (derecha) donde hemos ampliado el intervalo a $[-1,3\pi, 2,3\pi]$.

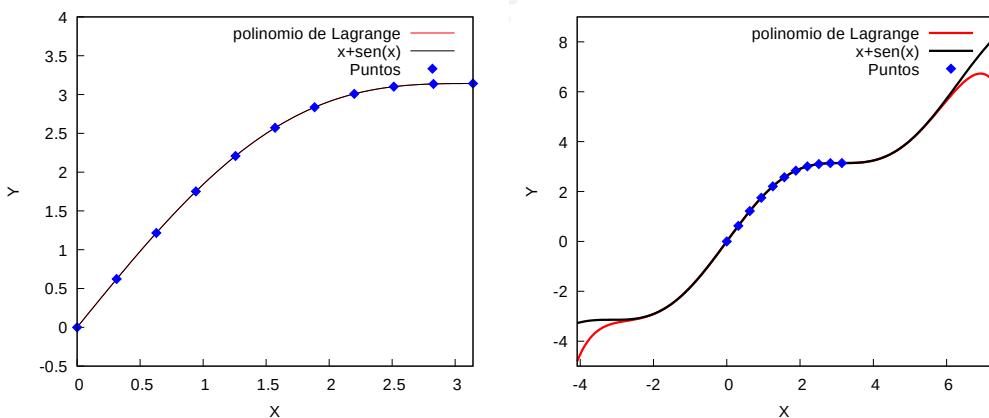


Figura 4.8: Polinomio interpolador de Lagrange de la función $f(x) = x + \sin(x)$ definido a partir de 10 puntos en $[0, \pi]$. A la izquierda comparación entre el polinomio y la función en $[0, \pi]$ y a la derecha en $[-1,3\pi, 2,3\pi]$.

Como ejercicio comprobar que ocurre si aumentamos el número de puntos (por ejemplo tomando $Nu=20$ o $Nu=40$).

resultados de ciertos cálculos.

Consideremos ahora la función de Runge $f(x) = 1/(1 + 25x^2)$ y estudiemos la interpolación de Lagrange. Comenzaremos con 10 puntos:

```
(%t19) kill(all)$
(%i1) load(interpoly)$ load(draw)$
(%i3) kill(x,pp,Nu,h)$ /* Chebyshev */
define(f(x),1/(1+25*x^2));
a:-1$ b:1 $Nu:20$ h:(b-a)/(Nu)$
x[0]:a$ x[n]:=x[0]+n*h;
pp10:makelist(float([x[k],f(x[k])]),k,0,Nu)$
(%o4) f(x):=1/(25*x^2+1)
(%o10) x[n]:=x[0]+n*h
```

A continuación calculamos el polinomio de Lagrange y lo dibujamos

```
(%i12) ratprint:false$
define(lag10(x),ratsimp(lagrange(pp10)))$
(%i14) wxdraw2d(color=red, key="polinomio de Lagrange",line_width=1,
      explicit(lag10(x),x,-1,1), key="1/(1+25 x^2)",yrange=[-2,5.5],
      color= black,explicit(f(x),x,-1,1),point_type=filled_diamond,
      point_size=1, color=blue, key = "Puntos", points(pp10))$
(%t14) (Graphics)
```

A la hora de simplificar con `ratsimp` MAXIMA primero transforma todos los números en coma flotante por sus correspondientes fracciones y en cada caso genera un comentario del tipo:

```
rat: replaced -0.001209807608535 by -484610/400567823 = -0.001209807608535
```

Esto se debe a que la variable `ratprint` está definida por defecto como `true`, por lo que la hemos redefinido para que tome el valor `false` y así eliminar dichos comentarios. Como se ve en la figura 4.9 hay una enorme oscilación cerca de los extremos.

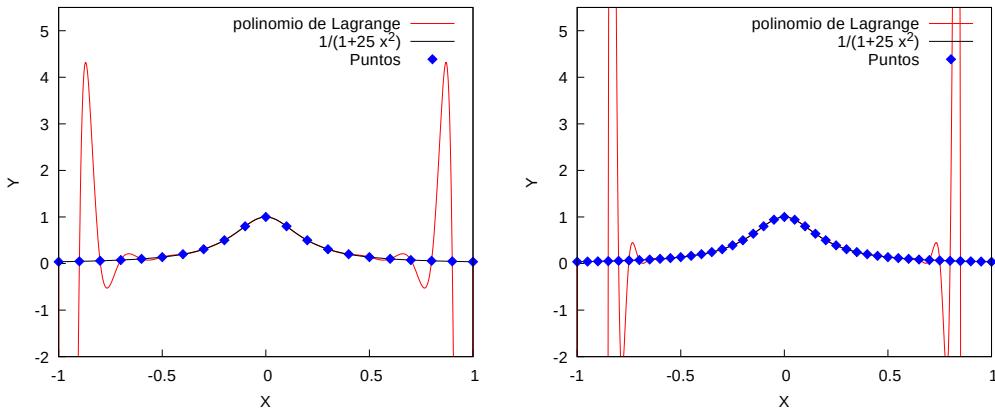


Figura 4.9: Polinomio interpolador de Lagrange de la función de Runge $f(x) = 1/(1 + 25x^2)$ definido a partir de 10 puntos equidistantes en $[-1, 1]$ (izquierda) y 20 puntos (derecha), respectivamente.

Esto empeora, como se ve en la gráfica derecha de la figura figura 4.9, si aumentamos el número de puntos de interpolación. Por ejemplo tomemos 20 puntos:

```
(%i15) kill(x,pp,Nu,h)$ /* Chebyshev */
      define(f(x),1/(1+25*x^2));
      a:-1$ b:1$ Nu:40$ h:(b-a)/(Nu)$
      x[0]:a$ x[n]:=x[0]+n*h;
      pp20:makelist(float([x[k],f(x[k])]),k,0,Nu)$
(%o16) f(x):=1/(25*x^2+1)
(%o22) x[n]:=x[0]+n*h
```

y calculemos el polinomio interpolador de Lagrange

```
(%i24) ratprint:false$
      define(lag20(x),ratsimp(lagrange(pp20)))$
(%i26) wxdraw2d(color=red, key="polinomio de Lagrange",line_width=1,
      explicit(lag20(x),x,-1,1),key="1/(1+25 x^2)",yrange=[-2,5.5],
      color= black,explicit(f(x),x,-1,1),point_type=filled_diamant,
      point_size = 1, color = blue, key = "Puntos", points(pp20))$
(%t26) (Graphics)
```

4.4.2. El polinomio de Lagrange con nodos de Chebyshev.

Para mejorar este resultado podemos usar en vez de nodos equidistantes los nodos de Chebyshev definidos en un intervalo $[a, b]$ mediante la expresión:

$$x(k) = \frac{a+b}{2} + \frac{a-b}{2} \cos \left[(2k-1) \frac{\pi}{2n} \right],$$

Así, preparamos la lista de los valores de $f(x)$ en dichos nodos:

```
(%i27) kill(chex,Nu)$ Nu:10$
      chex[k]:=(a+b)/2+((a-b)/2)*cos((2*k-1)*%pi/(2*Nu));
      pp10c:makelist(float([chex[k],f(chex[k])]),k,1,Nu)$
(%o29) chex[k]:=(a+b)/2+(a-b)/2*cos((2*k-1)*%pi)/(2*Nu)
(%i31) pp10c:makelist(float([chex[k],f(chex[k])]),k,1,Nu)$
```

Luego calculamos el polinomio interpolador y dibujamos el resultado:

```
(%i33) ratprint:false$
      define(lag10c(x),ratsimp(lagrange(pp10c)))$
(%i34) wxdraw2d(color=red, key="polinomio de Lagrange",line_width=1,
      explicit(lag10c(x),x,-1,1), key="1/(1+25 x^2)", yrange=[-.2,1.2],
      color= black,explicit(f(x),x,-1,1), point_type = filled_diamant,
      point_size = 1, color = blue, key = "Puntos", points(pp10c))$
(%t34) (Graphics)
```

La gráfica la podemos ver en la figura 4.10 (izquierda). A diferencia del caso con nodos equidistantes, la aproximación mejora notablemente si aumentamos en número de puntos tal y como se ve en la figura 4.10 (derecha):

```
(%i35) kill(chex,Nu)$ Nu:20$
      chex[k]:=(a+b)/2+((a-b)/2)*cos((2*k-1)*%pi/(2*Nu))$
```

```

ppc:makelist(float([chex[k],f(chex[k])]),k,1,Nu)$
define(lagc(x),ratsimp(lagrange(ppc)))$ 
wxdraw2d(color=red, key="polinomio de Lagrange",line_width=1,
explicit(lagc(x),x,-1,1), key="1/(1+25 x^2)", yrangle=[-.2,1.2],
color= black, explicit(f(x),x,-1,1), point_type = filled_diamond,
point_size = 1, color = blue, key = "Puntos", points(ppc))$ 
(%t40) (Graphics)

```

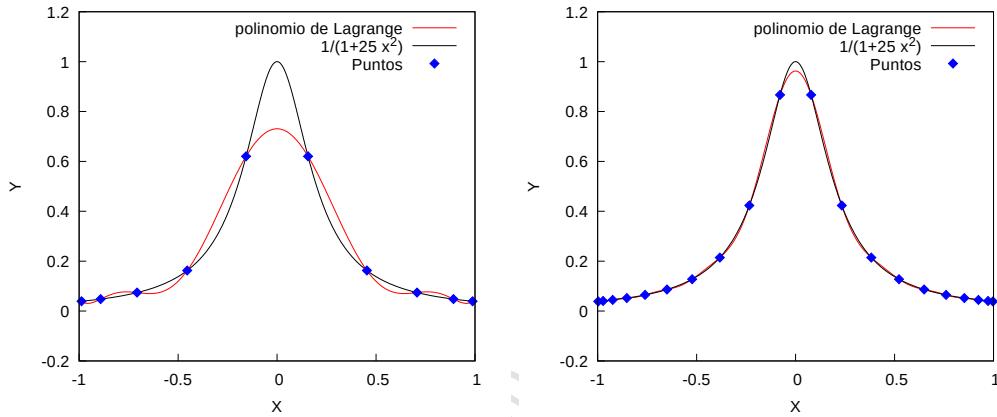


Figura 4.10: Polinomio interpolador de Lagrange de la función de Runge $f(x) = 1/(1 + 25x^2)$ definido a partir de 10 nodos de Chebyshev en $[-1, 1]$ (izquierda) y 20 nodos (derecha), respectivamente.

4.4.3. Splines.

Hay otra forma mucho mejor de interpolar que la polinómica consistente en usar ciertas funciones a trozos. Esta forma de interpolar se conoce como interpolación por *splines*. Vamos a comentar brevemente la forma más común: los splines cúbicos que denotaremos por $S(x)$. La idea es la siguiente. Imaginemos que tenemos en el intervalo $[a, b]$ los nodos de interpolación $a = x_0, x_1, \dots, x_{n-1}, x_n = b$ a los que le corresponden las imágenes $y_k = f(x_k)$. La función $S(x)$ va a ser una función definida a trozos en el intervalo $[a, b]$ definida en cada subintervalo $[x_k, x_{k+1}]$, $k = 0, \dots, n-1$, mediante un polinomio de grado 3, $S_k(x)$, $k = 0, \dots, n-1$ tal que $S_k(x_k) = y_k$, $k = 0, \dots, n-1$. Vamos ahora a imponer que $S(x)$ sea continua en $[a, b]$. Ello implica que los polinomios $S_{k-1}(x)$ y $S_k(x)$ deben tomar el mismo valor en los puntos t_{k+1} , i.e.,

$$S_k(t_{k+1}) = y_{k+1} = S_{k+1}(t_{k+1}), \quad 0 \leq k \leq n.$$

Lo mismo haremos con la primera y segunda derivada de $S(x)$ en $[a, b]$ es decir, se tiene

$$S'_k(t_{k+1}) = S'_{k+1}(t_{k+1}), \quad S''_k(t_{k+1}) = S''_{k+1}(t_{k+1}), \quad 0 \leq k \leq n.$$

Todo lo anterior conduce a un sistema de $n - 1$ ecuaciones lineales (que no escribiremos) con $n + 1$ incógnitas que tiene, en general, infinitas soluciones. Para obtener un sistema determinado se suelen agregar condiciones sobre los valores de $S''(x_0)$ y $S''(x_n)$ (normalmente indeterminados en el problema). Usualmente se toman los valores $S''(x_0) = S''(x_n) = 0$ en cuyo caso se dice que $S(x)$ es el *spline cúbico natural*. Justo

este spline es el que calcula MAXIMA por defecto, a no ser que se le especifiquen los valores de la primera derivada en los extremos, en cuyo caso usa esos valores para determinar $S''(x_0)$ y $S''(x_n)$.

La orden que calcula los splines cúbicos en MAXIMA es:

```
cspline(datos) o cspline(datos,d1=val1,dn=val2))
```

donde los datos han de ser una lista del tipo $[[x_1, y_1], \dots, [x_n, y_n]]$ o $[y_1, \dots, y_n]$, en cuyo caso se asume que $x_1 = 1, \dots, x_n = n$. Los valores val1 y val2 son los valores de la primera derivada en los puntos extremos x_1 y x_n que por defecto toman los valores 'unknown' (indeterminados).

Veamos un par de ejemplos. Comencemos nuevamente con los datos que usamos en la interpolación de Lagrange con un spline natural:

```
(%i41) datos: [[1,2],[3/2,1],[2,0],[5/2,-1],[3,1/2],[4,1]];
(%o41) [[1,2],[3/2,1],[2,0],[5/2,-1],[3,1/2],[4,1]]
(%i45) cspline(datos)$ /* Splin cúbico natural */
ratsimp(%)$
define(fs(x),%)$ 
wxdraw2d(color=red, key="spline cúbico natural",line_width=2,
explicit(fs(x),x,0.5,5), point_type=plus, point_size=4,
color = blue, key = "Puntos", points(datos))$ 
(%t45) (Graphics)
```

o asumiendo que las derivadas en $x = 1$ y $x = 4$ son cero:

```
(%i46) cspline(datos,d1=0,dn=0)$
ratsimp(%)$
define(fs1(x),%)$ 
wxdraw2d(color=red, key="spline cúbico",line_width=2,
explicit(fs1(x),x,0.5,5), point_type=plus, point_size=4,
color = blue, key = "Puntos", points(datos))$ 
(%t49) (Graphics)
```

Los resultados se pueden ver en las gráficas de la figura 4.11.

Veamos ahora como aproximan los splines a la función de Runge:

```
(%i50) kill(f,chex,Nu,ppc)$
define(f(x),1/(1+25*x^2));
Nu:20$
chex[k]:=(a+b)/2+((a-b)/2)*cos((2*k-1)*%pi/(2*Nu))$ 
ppc:makelist(float([chex[k],f(chex[k])]),k,1,Nu)$
(%o51) f(x):=1/(25*x^2+1)
(%i55) define(srun(x),cspline(ppc))$ 
wxdraw2d(color=red, key="spline cúbico",line_width=2,
explicit(srun(x),x,-1,1), key="1/(1+25 x^2)", yrange=[-.2,1.2],
color= black, explicit(f(x),x,-1,1), point_type=filled_diamant,
point_size = 1, color = blue, key = "Puntos", points(ppc))$ 
(%t56) (Graphics)
```

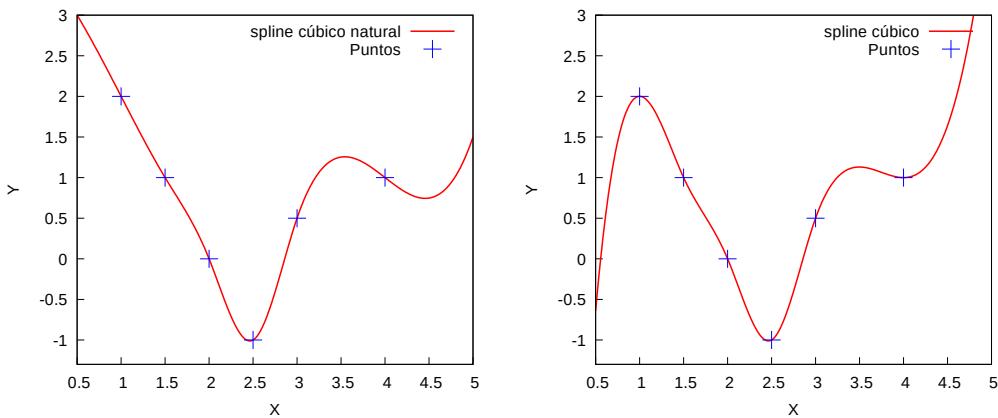


Figura 4.11: Spine cúbico natural obtenido a partir de la lista data (izquierda) y fijando los valores de la derivada en los extremos iguales a cero (derecha). Nótese que en el segundo caso $S(x)$ tiene extremos locales en los puntos $x = 1$ y $x = 4$.

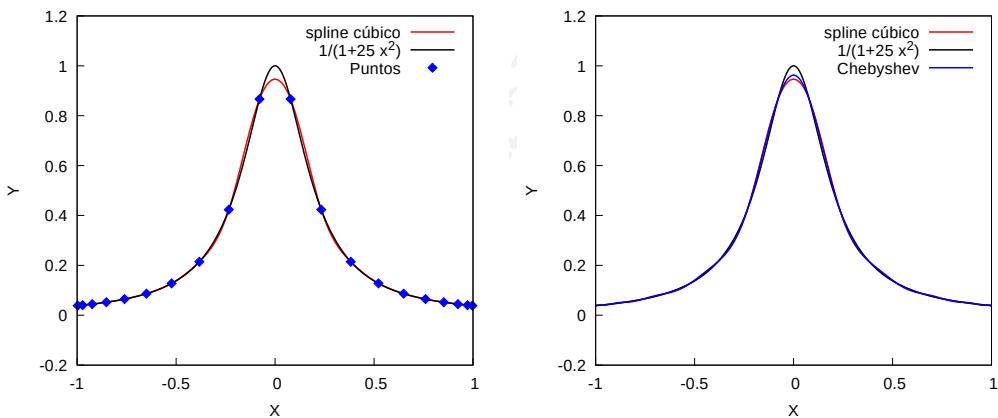


Figura 4.12: Spline cúbico natural para la función de Runge $f(x) = 1/(1 + 25x^2)$ en 20 puntos definidos a partir de 20 nodos de Chebyshev en $[-1, 1]$ (izquierda) y la comparación entre ambas aproximaciones —polinómica y splines— (derecha).

Nótese que aunque en la gráfica de la derecha de la figura 4.12 las curvas del spline y la del polinomio interpolador son casi idénticas, el spline es mucho más útil ya que no tenemos el problema de las inestabilidades si el número de nodos es muy grande (en cuyo caso el grado del polinomio interpolador también es muy elevado). Esto es especialmente visible si aumentamos el número de puntos de interpolación de la función, por ejemplo a 50. Elegiremos además una función muy sencilla, la función $\sin(x)$ en el intervalo $[0, 1]$:

```
(%i57) kill(x,pp,Nu,h)$ /* sin(x) 50 puntos */
      define(f(x),sin(x));
      a:0$ b:1$ Nu:50$ h:(b-a)/(Nu)$
      x[0]:a$ x[n]:=x[0]+n*h;
      pp:makelist(float([x[k],f(x[k])]),k,0,Nu)$
      ratprint:false$
      define(lagsin(x),ratsimp(lagrange(pp)))$
```

Evaluemos ahora el polinomio interpolador en los puntos de los nodos de forma exacta (trabajando simbólicamente):

```
(%i67) makelist(lagsin(x[4*k]),k,0,Nu/4)$ float(%);
(%o67) [0.0,0.07991469396917263,0.1593182066142462,0.2377026264271343,...,
0.7173560908995246,0.7707388788989682,0.8191915683010019]
```

Ahora repetimos la operación numéricamente:

```
(%i68) makelist(lagsin(float(x[4*k])),k,0,Nu/4);
(%o68) [0.0,0.07991469396918213,0.1593182072747862,0.2377048479005349,...,
5.479044778849127*10^7,3.05675556109827*10^8,7.931400833012784*10^9]
```

Nótese como los últimos valores comienzan a ser cada vez mayores. En cambio, con los splines no ocurre lo mismo:

```
(%o69) kill(all)$
(%i1) load(interp)$
(%i2) kill(x,pp,Nu,h,splsin)$
define(f(x),sin(x));
a:0$ b:1 $Nu:50$ h:(b-a)/(Nu)$
x[0]:a$ x[n]:=x[0]+n*h;
pp:makelist([x[k],f(x[k])],k,0,Nu)$
define(splsin(x),ratsimp(cspline(pp)))$
(%o3) f(x):=sin(x)
(%o9) x[n]:=x[0]+n*h

(%i13) makelist(splsin(x[4*k]),k,0,Nu/4)$ float(%);
(%o13) [0.0,0.07991469396917705,0.1593182066143341,0.2377026264267633,...,
0.7173560909092605,0.7707388787911007,0.8191915684317678]
(%i14) makelist(float(splsin(float(x[4*k]))),k,0,Nu/4);
(%o14) [0.0,0.07991469396917705,0.1593182066143341,0.2377026264267633,...,
0.7173560909092605,0.7707388787911007,0.8191915684317678]
```

Como ejercicio proponemos al lector que encuentre el spline para la función de Runge en 20 puntos equidistantes y compare los resultados con los aquí discutidos.

Finalmente debemos mencionar que el paquete `interp` cuenta también con el comando `ratinterp` que genera una función racional interpoladora con el grado del numerador fijado por el usuario.

Veamos un ejemplo muy sencillo:

```
(%i1) kill(all)$
load(interp)$
(%i2) datos: [[1,2],[3/2,1],[2,0],[5/2,-1],[3,1/2],[4,1]] ;
ratinterp(datos,2)/* Funcion racional */
ratsimp(%)$
define(fr(x),%);
(%o2) [[1,2],[3/2,1],[2,0],[5/2,-1],[3,1/2],[4,1]]
(%o5) fr(x):=(39*x^2-189*x+222)/(10*x^3-50*x^2+58*x+18)
(%i6) wxdraw2d(color=red, key="Interpolacion racional",line_width=3,
explicit(fr(x),x,0.5,4.5), point_type = filled_diamant,
point_size=2, color=blue, key ="Puntos", points(datos))$
```

A continuación la comparamos con la interpolación de Lagrange y mediante splines:

```
(%i7) wxdraw2d(color=green, key="Lagrange", line_width=3,
               explicit(lagrange(datos),x,0.5,4.5),
               color=red, key="Splines", line_width=3,
               explicit(cspline(datos),x,0.5,4.5),
               color=blue, key="Función racional", line_width=3,
               explicit(fr(x),x,0.5,4.5),
               point_type = plus, point_size=3, color=black, key="Puntos",
               points(datos), yrange = [-2,5])$
```

Las salidas se pueden ver en la figura 4.13.

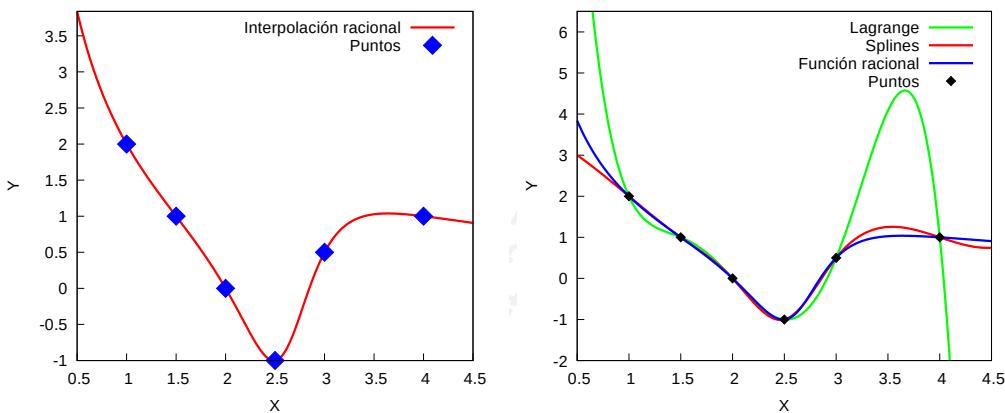


Figura 4.13: Interpolación racional de los datos $[1, 2], [3/2, 1], [2, 0], [5/2, -1], [3, 1/2], [4, 1]$ usando el comando `ratinterpol` (izquierda) y comparación con la interpolación por splines y el polinomio de Lagrange (derecha).

Para más detalle sobre este comando recomendamos al lector que consulte el manual [1].

4.5. Las funciones anónimas y su uso en MAXIMA.

Para terminar este capítulo vamos a discutir brevemente una función muy especial: la función anónima `lambda`. Este es un concepto muy usado en programación que consiste en crear funciones que no tienen un nombre especificado. Por ejemplo, en MAXIMA ya hemos visto un sinnúmero de funciones matemáticas teniendo cada una de ellas un nombre determinado. Hay ciertas situaciones donde es conveniente definir una función pero sin asignarle un nombre específico. Es ahí donde juegan un papel importante estas funciones. Más adelante usaremos una en la resolución de un problema muy concreto por lo que en este apartado vamos a discutir brevemente su uso.

Su sintaxis más usual es

```
lambda ([x_1, ..., x_m], expr_1, ..., expr_n)
```

donde la lista `[x_1, ..., x_m]` son sus argumentos y `expr_1, ..., expr_n` es cierta secuencia de órdenes o expresiones. Veamos unos ejemplos sencillos. Por ejemplo, definimos por un lado la función $f(x) = x^a$ de una variable x con a fijo, y por otro la función de dos variables $g(x, a) = x^a$:

```
(%i1) f:lambda([x],x^a);
(%o1) lambda([x],x^a)
(%i2) f(x);
(%o2) x^a
(%i3) f(2);
(%o3) 2^a
(%i4) g: lambda([x,a],x^a);
(%o4) lambda([x,a],x^a)
(%i5) g(x,a);
(%o5) x^a
(%i6) g(x,a);
(%o6) x^a
(%i7) g(2,3);
(%o7) 8
(%i8) f(t);
(%o8) t^a
```

El valor que devuelve la función es la salida de la última orden `expr_n`, es decir fijados los argumentos o variables (locales) las órdenes `expr_1`, ..., `expr_n` se ejecutan secuencialmente:

```
(%i9) h: lambda([x,a,b],y:x^b,a*y);
(%o9) lambda([x,a,b],y:x^b,a*y)
(%i10) h(x,a,b);
(%o10) a*x^b
```

¿Qué ocurre cuando una de las variables toma cierto valor global?

```
(%i11) x:2;
(%o11) 2
(%i12) f(x);
(%o12) 2^a
(%i13) g(3,a);
(%o13) 3^a
(%i14) g(x,a);
(%o14) 2^a
(%i15) g(t,a);
(%o15) t^a
```

En muchas ocasiones `lambda` viene acompañado del operador `apply` cuya sintaxis es

```
apply (F, [x_1, ..., x_n])
```

y que lo que hace es *evaluar el nombre* de la función *F* en sus argumentos. Por ejemplo, si tenemos una función propia de MAXIMA (el seno en nuestro ejemplo) podemos aplicarla a una lista directamente

```
(%i16) l:[a,b,c];
(%o16) [a,b,c]
(%i17) sin(l);
(%o17) [sin(a),sin(b),sin(c)]
```

o podemos usar el operador apply

```
(%i19) apply(sin,[l]);
(%o19) [sin(a),sin(b),sin(c)]
```

A veces al usar apply podemos tener el siguiente error:

```
(%i20) exp (2*%pi*%i*z);
(%o20) %e^(2*%i*%pi*z)
(%i21) apply (demoivre, [exp (%i * x)]);
apply: found false where a function was expected.
-- an error. To debug this try: debugmode(true);
```

En este caso la razón se debe a que `demoivre` tiene dos posibles usos: uno como función (para escribir en forma trigonométrica los números complejos) y la otra como una variable global de MAXIMA. Si queremos evitar la evaluación usamos la comilla simple '

```
(%i22) apply ('demoivre, [exp (2*%pi*%i*z)]);
(%o22) %i*sin(2*%pi*z)+cos(2*%pi*z)
```

Volvamos a la función `lambda` y veamos que efectivamente es anónima. Para ello usemos la función anónima *f* definida en la entrada (%i1) y preguntemos a MAXIMA por su definición (lo que se hace usando el comando `disfun`)

```
(%i23) apply (f , [z]);
(%o23) z^a
(%i24) dispfun(f);
fundef: no such function: f
-- an error. To debug this try: debugmode(true);
(%i25) apply(disfun,[f]);
fundef: no such function: lambda([x],x^a)
-- an error. To debug this try: debugmode(true);
```

Como vemos MAXIMA no reconoce que hayamos definido ninguna función *f*, a diferencia del caso cuando usamos la definición directa:

```
(%i26) ff(x):=z^a;
(%o26) ff(x):=z^a
(%i27) dispfun(ff);
(%t27) ff(x):=z^a
(%o27) [%t27]
(%i28) apply (dispfun, [ff]);
(%t28) ff(x):=z^a
(%o28) [%t28]
```

Incluso, si ejecutamos el comando `functions` sólo obtenemos la función ordinaria:

```
(%i29) functions;
(%o29) [ff(x)]
```

Conviene hacer notar la existencia de otro comando muy útil cuando se quiere aplicar una misma función a varios elementos de una lista de forma independiente. Ya nos encontramos con él en la página 53 al dibujar la región definida entre dos funciones. Su sintaxis es

```
map (f, expr_1, ..., expr_n)
```

Esencialmente lo que hace `map` es aplicar `f` a cada una de las subpartes de las expresiones, donde `f` puede ser tanto el nombre de una función de n argumentos como una expresión lambda de n argumentos.

Este procedimiento es muy útil en el caso cuando la expresión sobre la que se va a actuar es muy extensa y el uso directo sobre ella pudiese agotar el espacio de almacenamiento durante el transcurso del cálculo. Así, en el primer caso aplicamos la función `gg` a cada sumando de la expresión de la derecha (si queremos aplicarla sobre la suma se ha de proceder de forma distinta), en el segundo están involucradas dos expresiones (en este caso listas), y finalmente lo usamos en conjunción con una función `lambda`

```
(%i1) map(gg,x+a*y+b*z);
(%o1) gg(b*z)+gg(a*y)+gg(x)
(%i2) map(gg,x+a*y+b*z);
(%o2) gg(b*z)+gg(a*y)+gg(x)
(%i3) map(gg,[x+a*y+b*z]);
(%o3) [gg(b*z+a*y+x)]
(%i4) map("^", [a,b,c],[1,2,3]);
(%o4) [a,b^2,c^3]
(%i5) map ( lambda ([x], sin(x)*x^2), [a, b, c, d, e]);
(%o5) [a^2*sin(a),b^2*sin(b),c^2*sin(c),d^2*sin(d),e^2*sin(e)]
```

Por último vamos a mostrar que en efecto `map` actúa por separado en cada una de las subpartes de las expresiones de la derecha. Para ello mostraremos las diferentes salidas a la hora de simplificar una expresión algebraica con el comando `ratsimp`:

```
(%i5) ratsimp(x/(x^2+x)+(y^2+y)/y);
(%o5) ((x+1)*y+x+2)/(x+1)
(%i6) map(ratsimp, x/(x^2+x)+(y^2+y)/y);
(%o6) y+1/(x+1)+1
(%i7) map(ratsimp, [x/(x^2+x),(y^2+y)/y]);
(%o7) [1/(x+1),y+1]
```

Conviene destacar que esta función tiene muchas cosas en común con `maplist` cuando se aplica a listas. Para más detalles remitimos al lector al manual de MAXIMA [1].

Introducción al MAXIMA CAS con algunas aplicaciones
Prof. Dr. Renato Álvarez Nodarse

Capítulo 5

Resolviendo EDOs con MAXIMA.

5.1. Soluciones analíticas.

Las ecuaciones diferenciales juegan un papel muy importante en la modelación de fenómenos reales (véase e.g. [3, 4, 6]). MAXIMA cuenta con varios comandos para resolver analíticamente las ecuaciones diferenciales ordinarias (EDOs).

Comencemos comentando la orden `desolve` que resuelve sistemas de EDOs mediante la transformada de Laplace. Su sintaxis es

```
desolve([eq1, eq2, ..., eqn], [y1, y2, ..., yn])
```

donde `eq1, ..., eqn` denota las ecuaciones diferenciales (lineales) e `y1, ..., yn` las funciones desconocidas. Para definir la ecuación debemos usar la sintaxis correcta que consiste en escribir las funciones con sus variables pues en otro caso obtendremos errores. Por ejemplo, la ecuación $y' = y$ la tenemos que escribir de la siguiente forma:

```
(%i1) edo:diff(y(x),x,1) = y(x)$
```

es decir, escribiendo explícitamente la dependencia de x . Entonces hacemos

```
(%i2) desolve(edo,y(x));  
(%o2) y(x)=y(0)*%e^x
```

Si ahora queremos resolver el problema de valores iniciales (PVI), es decir, la ecuación $y' = f(x, y)$ con la condición inicial $y(x_0) = y_0$, usamos el comando `atvalue` para indicar el valor de la función y en $x = x_0$:

```
(%i3) atvalue(y(x),x=0,1);  
      desolve(edo,y(x));  
(%o3) 1  
(%o4) y(x)=%e^x
```

Es importante saber que para que la orden `desolve` funcione correctamente hay que especificar para las funciones desconocidas sus correspondientes variables, pues en caso contrario MAXIMA nos da un error:

```
(%i5) kill(y)$
      edo1:diff(y,x,1) = y;
      desolve(edo,y);
(%o6) 0=y
      length: argument cannot be a symbol; found y
      -- an error. To debug this try: debugmode(true)
```

En este caso lo que ocurre es que MAXIMA asume que y es una constante y por tanto $y' = 0$. Podríamos pensar que si le decimos a MAXIMA que y depende de x , para lo que podemos usar el comando `depends` que ya vimos antes en la página [36](#)

```
depends(y,[x]); diff(y,x,1) = y;
```

el problema va a desaparecer pues MAXIMA nos devuelve la ecuación `'diff(y,x,1)=y`. Sin embargo, si intentamos resolver la ecuación con `desolve(%,y(x))` obtenemos un error `desolve: can't handle this case` es decir MAXIMA nos comunica que no puede tratar esa ecuación. Es decir, es imprescindible que al introducir la función a calcular y la escribamos como $y(x)$.

Resolvamos ahora la ecuación $y' = -1/2y + \sin(x)$ con la condición $y(0) = 1$ y dibujemos su solución. Si usamos sólo el comando `desolve` obtenemos la solución general de la ODE pero no la solución del PVI

```
(%i8) kill(y)$
      edo2:diff(y(x),x,1) = -1/2*y(x)+sin(x);
      desolve(edo2,y(x))$%
      expand(%);
(%o9) 'diff(y(x),x,1)=sin(x)-y(x)/2
(%o11) y(x)=(2*sin(x))/5-(4*cos(x))/5+y(0)*%e^(-x/2)+(4*%e^(-x/2))/5
```

así que usamos la combinación con `atvalue`

```
(%i12) atvalue(y(x),x=0,1);
      desolve(edo2,y(x));
(%o12) 1
(%o13) y(x)=(2*sin(x))/5-(4*cos(x))/5+(9*%e^(-x/2))/5
```

Para definir la función solución y luego trabajar con ella usamos, como ya hemos visto, la orden `define`

```
(%i14) define(soledo2(x),second(%));
(%o14) soledo2(x):=(2*sin(x))/5-(4*cos(x))/5+(9*%e^(-x/2))/5
```

Además hemos usado un comando muy útil `second` que, dada una ecuación

```
miembro izquierdo = miembro derecho
```

extrae el miembro de la derecha de la misma¹ Finalmente, dibujamos la solución (ver la gráfica de la izquierda de la figura [5.1](#)):

```
(%i15) wxplot2d(soledo2(x),[x,0,18],[ylabel,"y"])$
(%t15) << Graphics >>
```

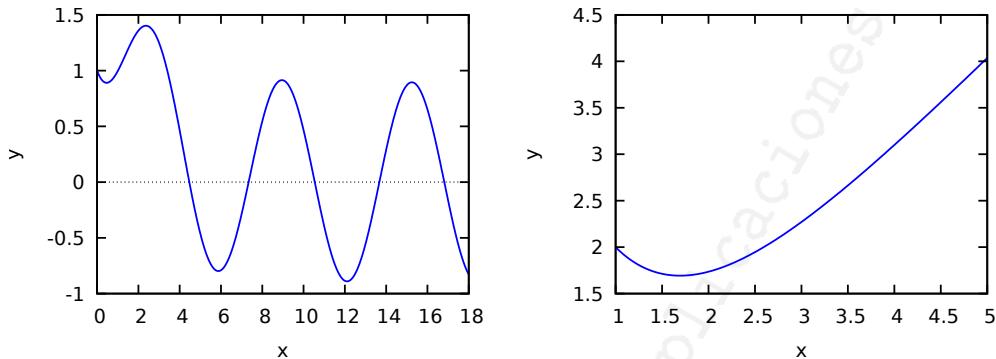


Figura 5.1: Solución del PVI $y' = -1/2y + \sin(x)$, $y(0) = 1$ (izquierda) y $z' = -z + x$, $z(1) = 2$ (derecha).

El comando `desolve` no siempre funciona. Si lo aplicamos a la ecuación separable $y' = 1 + y^2$, nos da

```
(%i16) edo3:diff(w(x),x,1) = 1+(w(x))^2;
       desolve(edo3,w(x));
(%o16) 'diff(w(x),x,1)=w(x)^2+1
(%o17) w(x)=ilt(((laplace(w(x)^2,x,g36165)+w(0))*g36165+1)/g36165^2,g36165,x)
```

donde `ilt` significa transformada inversa de Laplace (en sus siglas inglesas *inverse Laplace transform*), pero no nos permite obtener ningún valor de la misma. En este caso es conveniente utilizar otro comando: el comando `ode2` cuya sintaxis es

```
ode2(eqn, variable dependiente, variable independiente)
```

y que resuelve EDOs de primer y segundo orden intentando reconocer alguna de las ecuaciones tipo más usuales: lineales, separables, de Bernoulli, etc. o reducibles a ellas.

Por ejemplo, resolvamos la EDO $z' = -z + x$:

```
(%i18) 'diff(z,x)=x-z;
       ode2('diff(z,x)=x-z,z,x)$
       expand(%);
(%o18) 'diff(z,x,1)=x-z
(%o20) z=%c*%e^(-x)+x-1
```

Si queremos resolver el PVI $z' = -z + x$, $z(1) = 2$ hay que usar el comando `ic1` cuya sintaxis es

```
ic1(solución, valor de x, valor de y)
```

¹Si queremos el izquierdo podemos usar el comando `first`. También hacen lo mismo los comandos `rhs`

donde `solución` es la solución general que da el comando `ode2` y el valor de x y el valor de y , son los valores que toma la y cuando $x = x_0$, i.e., los valores iniciales. Así tenemos

```
(%i21) expand(ic1(% ,x=1,z=2));
(%o21) z=2*%e^(1-x)+x-1
(%i22) define(s1(x),second(%));
(%o22) s1(x):=2*%e^(1-x)+x-1
```

que podemos representar en una gráfica (ver figura 5.1, gráfica de la derecha)

```
(%i23) wxplot2d(s1(x),[x,1,5],[ylabel,"y"])$
(%t23) << Graphics >>
```

Nótese que en la entrada (%i18) hemos escrito '`diff`' y no `diff`. La razón es que el apóstrofe '`'` delante del comando `diff` obliga a MAXIMA a no ejecutar la orden (probar sin el apóstrofe)². Otra posibilidad es decirle a MAXIMA que z depende de x , lo que podemos hacer, como hemos visto anteriormente, usando el comando `depends`. Como ejercicio recomendamos comprobar que ocurre si usamos la secuencia

```
depends(z,x);
ode2('diff(z,x)=x-z,z,x);
expand(ic1(% ,x=1,z=2));
```

Usemos el comando `ode2` al PVI $y' = 1 + y^2$, $y(0) = 0$ que no pudimos resolver con la orden `desolve`

```
(%i23) kill(w)$
'diff(w,x)=1+w^2;
ode2('diff(w,x)=1+w^2,w,x)$
sol:expand(%);
expand(ic1(sol,x=0,w=0));
(%o24) 'diff(w,x,1)=w^2+1
(%o26) atan(w)=x+%c
(%o27) atan(w)=x
(%i28) sol:solve(% ,w);
(%o28) [w=tan(x)]
```

La última salida (entre "[]") es una lista. Las listas son especialmente importantes en MAXIMA como ya hemos comentado en el apartado 2.1 (ver página 14). En este caso simplemente nos interesa extraer un elemento dado de la lista lo que se hace, como ya vimos, usando el comando `part(lista,nº del elemento)` (o bien, en nuestro caso escribiendo `sol[1]`) así que hacemos

```
(%i29) part(% ,1);
(%o29) w=tan(x)
(%i30) define(solw(x),second(%));
(%o30) solw(x):=tan(x)
```

(del inglés "right hand side", miembro derecho) y `lhs` ("left hand side") miembro izquierdo, respectivamente.

²Véase el apartado §8.1 del manual [1] para más detalles.

Como ejercicio resolver las siguientes ecuaciones:

- Ecuación separable:

$$x^3(y-1)\left(\frac{dy}{dx}\right) + (x-1)y^3 = 0.$$

- Ecuación homogénea:

$$3xy^2\left(\frac{dy}{dx}\right) + y^3 + x^3 = 0.$$

- Reducible a homogénea:

$$\frac{dy}{dx} = \frac{y+x-1}{-y+x-1}.$$

- Ecuación exacta:

$$(8x - 4y^3)\left(\frac{dy}{dx}\right) + 8y + 4x^3 = 0.$$

- Ecuación de Bernoulli:

$$\frac{dy}{dx} - y + \sqrt{y} = 0.$$

5.2. Soluciones numéricas.

La orden `ode2` no siempre funciona. Por ejemplo si intentamos resolver la EDO $z' = x - \sin z$ obtenemos como salida `false`

```
(%i31) ode2('diff(z,x)=x-sin(z),z,x);
(%o31) false
```

La razón es que MAXIMA no reconoce ningún patrón en esta ecuación. Cuando esto ocurre no queda más remedio que usar algún método numérico. Más adelante explicaremos el funcionamiento de un par de métodos muy sencillos, mientras que aquí nos restringiremos a mostrar como se pueden resolver numéricamente las EDOs usando MAXIMA.

Comenzaremos usando el comando `runge1` que permite resolver numéricamente PVI de primer orden del tipo $y' = f(x, y)$, $y(x_0) = y_0$ con MAXIMA usando en método de Runge-Kutta. Para ello lo primero que tenemos que hacer es cargar el paquete numérico `diffEq` y luego ejecutar dicho comando. La sintaxis de `runge1` es la siguiente:

```
runge1(f, x0, x1, h, y0)
```

donde f es la función $f(x, y)$ de la ecuación $y' = f(x, y)$, x_0 y x_1 los valores inicial, x_0 , y final, x_1 , de la variable independiente, respectivamente, h es la longitud (o paso) de los subintervalos e y_0 es el valor inicial y_0 que toma y en x_0 . El resultado es una lista que a su vez contiene tres listas: la primera contiene las abscisas x , la segunda las ordenadas y y tercera las correspondientes derivadas y' .

Como ejemplo consideremos el PVI $y' = 1 + y$, $y'(0) = 1$. Ante todo limpiaremos todas las variables y cargaremos el paquete `diffEq`

```
(%i32) kill(all);
(%o0) done
(%i1) load(diffeq);
(%o1) /usr/share/maxima/5.20.1/share/numeric/diffeq.mac
```

A continuación definimos la función f , y el paso h , para, a continuación, invocar la orden `runge1`

```
(%i2) f(x,y):=1+y; h:1/20;
(%o2) f(x,y):=1+y
(%o3) 1/20
(%i4) solnum:runge1(f,0,1,h,1)
(%o4) [[0.0,0.05,...,0.95],[1.0,1.10210416666667,...,4.150987618241528],
      [2.0,2.10210416666667,...,5.150987618241528]]
(%i5) wxplot2d([discrete,solnum[1],solnum[2]])$
(%t5) << Graphics >>
```

Como esta ecuación es exactamente resoluble podemos comparar sus gráficas. Para ello primero usamos `ode2` e `ic1` para resolver analíticamente el PVI:

```
(%i6) ode2('diff(w,x)=1+w,w,x)$
      sol:expand(%);
      expand(ic1(sol,x=0,w=1));
      define(solw(x),second(%));
(%o7) w=%c*%e^x-1
(%o8) w=2*%e^x-1
(%o9) solw(x):=2*%e^x-1
```

Y ahora dibujamos ambas gráficas

```
(%i10) wxplot2d([[discrete,solnum[1],solnum[2]],solw(x)], [x,0,1],
                [y,1,5], [legend,"sol. numerica","solucion exacta"],
                [xlabel,"x"], [ylabel,"y"], [color,red,blue], [point_type,plus],
                [style,[points,5],[lines,2]])$
(%t10) << Graphics >>
```

La gráfica la podemos ver en la figura 5.2 (izquierda).

Finalmente resolvemos numéricamente el PVI $y' = x - \sin(y)$, $y(0) = 1$.

```
(%i11) g(x,y):=x-sin(y);
      sol2:runge1(g,0,1,1/20,1);
(%o11) g(x,y):=x-sin(y)
(%o12) [[0.0,0.05,...,0.9,0.95],
      [1.0,0.95944094204897,...,0.75740012409521,0.7689229050892],
      [-0.8414709848079,-0.76887080939197,...,0.25463842295662]]
(%i13) wxplot2d([discrete,sol2[1],sol2[2]], [x,0,1.01],
                [point_type,diamond], [legend, "y(x) con runge1"],
                [xlabel, "x"], [ylabel, "y"], [color,red],
                [style,[points,3]])$
(%t13) << Graphics >>
```

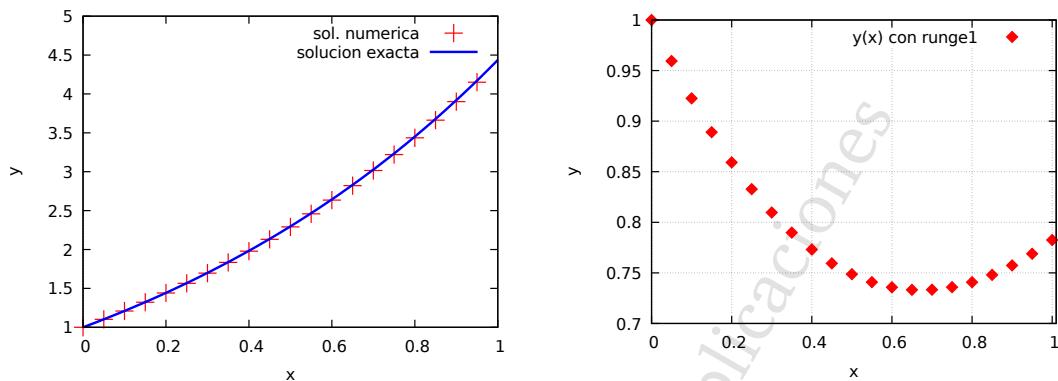


Figura 5.2: Comparación de la solución numérica y analítica del PVI $y' = 1 + y$, $y(0) = 1$ (izquierda) y solución numérica del PVI $y' = x - \sin(y)$, $y(0) = 1$ (derecha).

La gráfica la podemos ver en la figura 5.2 (derecha).

Además de la orden `runge1 MAXIMA` dispone de un comando alternativo para resolver numéricamente una ecuación diferencial pero que además es extensible a sistemas de ecuaciones de primer orden lo cual nos será de utilidad más adelante. Se trata de comando `rk` del paquete `dynamics` que no es más que otra implementación de un método de Runge-Kutta. Su sintaxis, para el caso del PVI $y' = f(x, y)$, $y(x_0) = y_0$ es la siguiente:

`rk(f, y, y0, [x, x0, x1, h])`

donde f es la función $f(x, y)$ de la ecuación $y' = f(x, y)$, x_0 y x_1 los valores inicial, x_0 , y final, x_1 , de la variable independiente, respectivamente, h es la la longitud de los subintervalos e y_0 es el valor inicial y_0 que toma y en x_0 . El resultado es una lista con los pares $[x, y]$ de las abscisas x y las ordenadas y .

Así tenemos la secuencia de órdenes

```
(%i14) load(dynamics)$
(%i15) h:1/20; kill(x,y)$
      numsolrk:rk(x-sin(y),y,1,[x,0,1,h])$
```

(%o15) 1/20

```
(%i17) numsolrk;
(%o18) [[0, 1], [0.05, 0.95973997169251], [0.1, 0.92308155305544], ...
```

```
[0.95, 0.77210758398484], [1.0, 0.78573816934072]]
```

La salida de `rk` es justo una lista que entiende perfectamente el comando `plot2d` por lo que podemos dibujar la solución y comparar ambas salidas numéricas.

```
(%i19) wxplot2d([discrete,numsolrk],[legend, "y"],
               [xlabel, "x"], [ylabel, "y"], [color,blue])$
```

```
(%t19) << Graphics >>
```

```
(%i20) wxplot2d([[discrete,sol2[1],sol2[2]], [discrete,numsolrk]],
               [x,0,1.05], [style, [points,5], [points,5]],
               [color,red,blue], [point_type, diamond,asterisk],
               [xlabel,"x"], [ylabel,"y"], [legend,"runge1","rk"])$
```

```
(%t20) << Graphics >>
```

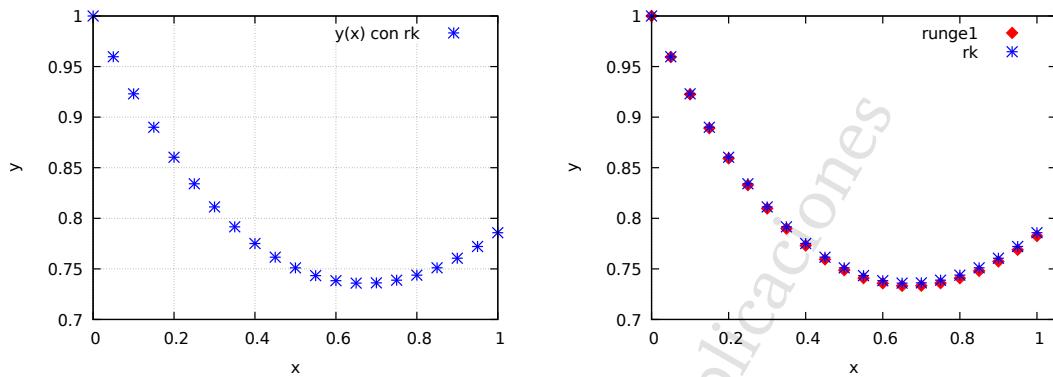


Figura 5.3: A la izquierda vemos la solución numérica usando el comando `rk` mientras que a la derecha están representados los resultados usando ambos comandos (`runge1` con rombos rojos y `rk` con asteriscos azules) para el PVI $y' = x - \sin(y)$, $y(0) = 1$.

Las gráficas las podemos ver en la figura 5.3.

5.2.1. Exportando ficheros de datos.

Para terminar este apartado mostraremos como se pueden exportar a un fichero los resultados de una simulación realizada con MAXIMA. Para ello usaremos la orden `write_data(salida,fichero)`. Como ejemplo exportaremos la salida del comando `rk` que ya hemos visto antes (ver página 5.2) y que genera una lista de datos. Concretamente resolveremos la ecuación

$$x'' = \sin(x) + (\cos(\omega t) + \cos(2\omega t))(1 - \exp(t/\tau)), \quad x(0) = x'(0) = 0, \quad t \in [0, 100], \quad (5.2.1)$$

o, equivalentemente, el sistema de ecuaciones de primer orden

$$\begin{cases} x' = y, \\ y' = (\sin(2x) + \cos(\omega t) + \cos(2\omega t))(1 - \exp(t/\tau)), \end{cases}$$

con $x(0) = y(0) = 0$. Así, cargamos el paquete y definimos la función y los parámetros del problema:

```
(%i1) load("dynamics")$  
kill(w,f,x,y,t)$  
w:float(sqrt(2)/10);tau:100/w;  
define(f(t),(cos(w*t)+cos(2*w*t))*(1-exp(t/tau)));  
(%o3) 0.1414213562373095  
(%o4) 707.1067811865476  
(%o5) f(t):=(cos(0.282842712474619*t)+cos(0.1414213562373095*t))*  
(1-%e^(0.001414213562373095*t))
```

para, a continuación, resolverla la ecuación usando el método de Runge-Kutta:

```
(%o6 sol:rk( [y,sin(2*x)+f(t)], [x,y], [0,0], [t, 0, 100, 0.1] )$  
write_data (sol, "/home/renato/solnum.dat" );  
(%o7) done
```

Como comentario adicional queremos mencionar que hay una orden que nos permite conocer el tiempo que tarda MAXIMA en realizar las operaciones. Ello es interesante cuando tenemos un programa que requiere muchas operaciones y queremos estimar el tiempo, o bien si lo estamos ejecutando en segundo plano tal y como explicaremos a continuación. El comando en cuestión es `time`. Si en nuestro ejemplo queremos saber cuanto demora la resolución de la ecuación (véase la salida `%o6`) escribimos

```
(%i8) time(%o6);
(%o8) [0.11]
```

que indica que ha tardado 0.11 segundos. También podemos escribir una serie de salidas

```
(%i9) time(%o6,%o7);
(%o9) [0.11,0.03]
```

y `time` nos devuelve una lista con los tiempos que usa MAXIMA para tratar cada una de ellas. También hay una variable que controla de forma global la impresión de los tiempos: `showtime`. De hecho si hacemos `showtime:true` MAXIMA devuelve el tiempo de ejecución de cada una de las líneas de salida.

Como comentamos antes, es posible ejecutar un programa de MAXIMA en segundo plano (*background*). En el caso de LINUX esto lo podemos hacer desde la pantalla de comandos de manera muy simple. Primero creamos un fichero que contenga el programa anterior con el nombre `output-edo.max` (usando cualquier editor de texto) y luego lo ejecutamos en la pantalla de comandos de LINUX de la siguiente forma:

```
$ maxima < output-edo.max &
```

La salida, en este ejemplo, aparecerá guardada en el fichero `solnum.dat` del directorio raíz del usuario “renato”.

Para recuperar los datos usamos las órdenes `read_matrix` y `file_search` que ya vimos en la página [83](#)

```
(%i10) kill(all)$
(%i1) mm:read_matrix (file_search ("~/home/renato/solnum.dat"))$
(%i2) xx1:makelist(col(mm,1)[k][1],k,1,length(mm))$
      yy1:makelist(float(col(mm,2)[k][1]),k,1,length(mm))$
      yyp:makelist(float(col(mm,3)[k][1]),k,1,length(mm))$
(%i5) wxplot2d([discrete,xx1,yy1],[xlabel,"t"],[ylabel,"x(t)"]);
(%i6) wxplot2d([discrete,xx1,yyp],[xlabel,"t"],[ylabel,"x'(t)"]);
```

Las gráfica las vemos en la figura [5.4](#)

Como ejercicio representar el diagrama de fase de las ecuación ([5.2.1](#)).

5.3. Implementando un método numérico con MAXIMA.

Como ya hemos visto son pocas las EDOs que se pueden resolver analíticamente, es por ello que se necesita de métodos fiables para obtener la solución de una EDO numérica.

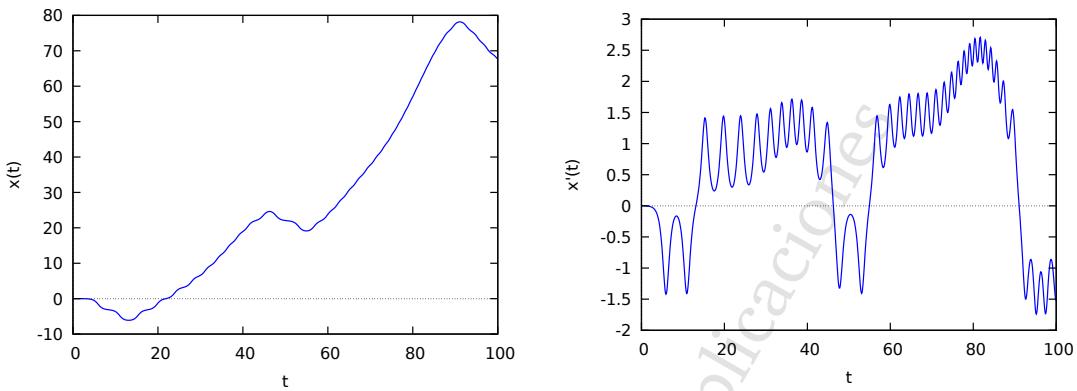


Figura 5.4: A la izquierda vemos la solución numérica del PVI (5.2.1) usando el comando `rk` mientras que a la derecha están representados los valores de la primera derivada en función de t .

camente. Aunque MAXIMA cuenta con las órdenes `runge1` y `rk` comentadas en el apartado anterior vamos a mostrar aquí como implementar un método numérico con MAXIMA. Por simplicidad implementaremos el método de Euler y una de sus variantes más sencillas.

Supongamos que queremos resolver el problema de valores iniciales

$$\frac{dy(x)}{dx} = f(x, y), \quad y(x_0) = y_0. \quad (5.3.1)$$

Es obvio que usando un ordenador sólo podremos resolver el problema de valores iniciales en un intervalo acotado, digamos $[x_0, x_0 + l]$ (aunque l podría ser muy grande). Para ello vamos a dividir el intervalo en N subintervalos $[x_0, x_1] \cup [x_1, x_2] \cup \dots \cup [x_{N-1}, x_N]$, $x_N = x_0 + l$. Supongamos que hemos encontrado los valores de y en los puntos x_0, x_1, \dots, x_N , que denotaremos por y_0, y_1, \dots, y_N . Entonces, para encontrar una solución aproximada $\hat{y}(x)$ podemos unir los puntos (x_i, y_i) , $i = 0, 1, \dots, N$ mediante líneas rectas (ver figura 5.5). Es evidente que si el valor y_i es bastante cercano al valor real $y(x_i)$ para todos los $i = 0, 1, \dots, N$, entonces, al ser \hat{y} e y funciones continuas, la solución aproximada $\hat{y}(x)$ estará “muy cercana” a la solución real $y(x)$ en cada uno de los intervalos $[x_i, x_{i+1}]$.

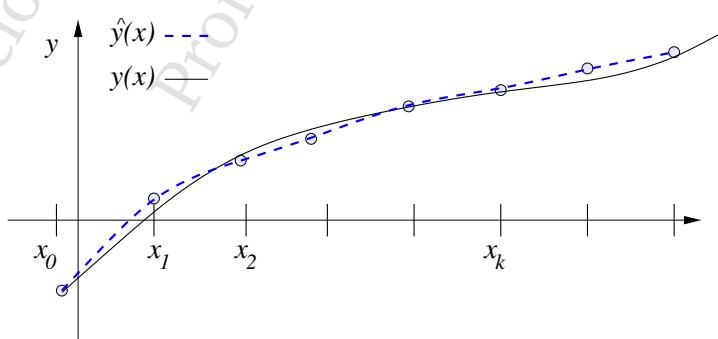


Figura 5.5: Construcción de un esquema numérico.

Vamos a usar por simplicidad intervalos iguales, es decir, vamos a escoger los *nodos* x_i equidistantes. Lo anterior se conoce en la teoría de métodos numéricos como una red

equiespaciada o uniforme de paso $h = l/N$. Así pues tendremos las siguientes ecuaciones: $x_k = x_0 + kh = x_0 + k(\frac{l}{N})$, $k = 0, 1, \dots, N$, $x_{k+1} = x_k + h$. Además, está claro que la única información que tenemos para calcular los valores y_i , aparte de la condición inicial, es la propia EDO que satisface nuestra incógnita $y(x)$. ¿Cómo encontrar entonces los valores y_i ? La idea es como sigue:

1. Usando la EDO y la condición inicial calculamos el valor de y_1 en el punto $x_1 = x_0 + h$
2. A continuación, usando los valores de y_0 e y_1 calculamos el valor aproximado y_2 de $y(x)$ en x_2 , y así sucesivamente.
3. Conocido los valores y_0, y_1, \dots, y_k encontramos el valor y_{k+1} de $y(x)$ en x_{k+1} .

El método de Euler.

Entre las muchas posibilidades para resolver el problema de encontrar en valor de $y(x_{k+1})$ conocidos los valores anteriores $y(x_j)$, $j = 0, \dots, k$, podemos optar, por ejemplo, por usar el teorema de Taylor

$$y(x_{k+1}) = y(x_k + h) = y(x_k) + y'(x_k)h + \frac{y''(x_k)}{2!}h^2 + \dots \quad (5.3.2)$$

Como $y'(x_k) = f(x_k, y(x_k))$, entonces

$$\begin{aligned} y''(x_k) &= \frac{d}{dx} \left(\frac{dy}{dx} \right) \Big|_{x=x_k} = \frac{d}{dx} (f(x, y(x))) \Big|_{x=x_k} = \frac{\partial f(x, y)}{\partial x} + \frac{\partial f(x, y)}{\partial y} \frac{\partial y}{\partial x} \Big|_{(x,y)=(x_k,y(x_k))} \\ &= \frac{\partial f(x, y)}{\partial x} + f(x, y) \frac{\partial f(x, y)}{\partial y} \Big|_{(x,y)=(x_k,y(x_k))}. \end{aligned}$$

Luego, (5.3.2) nos da

$$y(x_{k+1}) = y(x_k) + hf(x_k, y(x_k)) + \left[\frac{\partial f}{\partial x}(x_k, y(x_k)) + f(x_k, y(x_k)) \frac{\partial f}{\partial y}(x_k, y(x_k)) \right] \frac{h^2}{2!} + \dots$$

La aproximación más sencilla es por tanto cuando nos quedamos en la serie anterior con el término de primer orden, o sea, cuando tenemos el esquema numérico

$$y_1 = y_0 + hf(x_0, y_0), \quad y_2 = y_1 + hf(x_1, y_1), \quad \dots, \quad y_{k+1} = y_k + hf(x_k, y_k), \quad (5.3.3)$$

donde $y_0 = y(x_0)$.

El esquema anterior se conoce por el nombre de *esquema o método de Euler* y es, quizás, el método más sencillo para resolver numéricamente una EDO de primer orden. Nótese que dicho esquema necesita en cada paso del valor $y(x_k)$, por tanto cuanto más cercano sea el valor y_k calculado del $y(x_k)$ real más preciso será el método. Obviamente en cada paso arrastramos el error del cálculo del paso anterior. En efecto, para calcular y_1 usamos el valor real y_0 pero cuando calculamos y_2 , sustituimos el valor exacto $y(x_1)$ desconocido por su valor aproximado y_1 , para calcular y_3 sustituimos el valor $y(x_2)$ por su valor aproximado y_2 , y así sucesivamente.

Veamos algunos ejemplos.

Comenzaremos con una ecuación que sepamos resolver exactamente. Por ejemplo, estudiemos el problema de valores iniciales

$$y' + y = x, \quad y(0) = 1, \quad x \in [0, 1],$$

cuya solución exacta es $y(x) = 2e^{-x} - 1 + x$. Escogeremos una discretización equidistante con paso $h = 1/20$ (20 subintervalos iguales).

Vamos a implementarlo con MAXIMA. Para ello usaremos la notación $x[n]$ que es esencialmente la forma de definir sucesiones con MAXIMA.

Comenzaremos definiendo el intervalo $[x_0, x_n]$, cuya longitud denotaremos por $l = x_n - x_0$ y que en nuestro caso será igual a 1.

```
(%i1) x[0]:0;
      1:1;Nu:20;h:1/Nu;
      x[n]:=x[0]+n*h;
(%o1) 0
(%o2) 1
(%o3) 20
(%o4) 1/20
(%o5) x[n]:=x[0]+n*h
```

A continuación limpiaremos la variable f que definirá nuestra función f del PVI (5.3.1)

```
(%i6) kill(f)$ y[0]:1;
      define(f[x,y],-y+x);
(%o7) 1
(%o8) f[x,y]:=x-y
```

Ya estamos listos para pedirle a MAXIMA que encuentre nuestra solución numérica. Para ello le pediremos que calcule los valores de $y(x_k)$ mediante la fórmula (5.3.3) y luego crearemos una lista que podamos representar gráficamente. Para crear la lista usamos el comando makelist

```
(%i9) y[k]:=float(y[k-1]+h*f[x[k-1],y[k-1]]);
      sol:makelist(float([x[k],y[k]]),k,0,Nu);
(%o9) y[k]:=float(y[k-1]+h*f[x[k-1],y[k-1]])
(%o10) [[0.0,1.0],[0.05,0.95],[0.1,0.905],[0.15,0.86475], ...,
      [0.95,0.70470720507062],[1.0,0.71697184481708]]
```

Los resultados están escritos en la tabla 5.1 o dibujados en la gráfica 5.6 (izquierda). Para dibujarlos hemos usado el comando plot2d

```
(%i11) wxplot2d([[discrete,sol]], [style, [points,2,2]],
      [legend,"y(x)"], [xlabel,"x"], [ylabel,"y"])$
(%t11) << Graphics >>
```

Comparemos ahora la solución exacta $y(x) = 2e^{-x} - 1 + x$ del PVI con los valores numéricos que nos da en método de Euler y dibujemos ambas

k	x_k	$\hat{y}(x_k)$	$y(x_k)$	$\hat{y}(x_k) - y(x_k)$
0	0	1.	1.	0
1.	0.05	0.95	0.952459	-0.00245885
2.	0.1	0.905	0.909675	-0.00467484
3.	0.15	0.86475	0.871416	-0.00666595
4.	0.2	0.829012	0.837462	-0.00844901
5.	0.25	0.797562	0.807602	-0.0100397
6.	0.3	0.770184	0.781636	-0.0114527
7.	0.35	0.746675	0.759376	-0.0127016
8.	0.4	0.726841	0.74064	-0.0137992
9.	0.45	0.710499	0.725256	-0.0147575
10.	0.5	0.697474	0.713061	-0.0155874
11.	0.55	0.6876	0.7039	-0.0162994
12.	0.6	0.68072	0.697623	-0.0169031
13.	0.65	0.676684	0.694092	-0.0174074
14.	0.7	0.67535	0.693171	-0.0178206
15.	0.75	0.676582	0.694733	-0.0181506
16.	0.8	0.680253	0.698658	-0.0184046
17.	0.85	0.686241	0.70483	-0.0185892
18.	0.9	0.694429	0.713139	-0.0187107
19.	0.95	0.704707	0.723482	-0.0187748
20.	1.	0.716972	0.735759	-0.018787

Cuadro 5.1: Solución numérica de la ecuación $y' + y = x$, $y(0) = 1$ en el intervalo $[0, 1]$ para $N = 20$ según el esquema de Euler.

```
(%i12) expon(x):=-1+2*exp(-x)+x$  
wxplot2d([[discrete,sol],expon(x)], [x,0,1], [style, [points,2,2],  
[lines,1,1]], [legend,"y(x) aproximado","y(x) real"],  
[ xlabel,"x"], [ ylabel,"y"])$  
(%t13) << Graphics >>
```

Finalmente, podemos calcular los errores cometidos en cada paso y representarlos gráficamente –ver figura 5.6 (derecha)–.

```
(%i14) compsol:makelist(float([x[k],abs(y[k]-expon(x[k]))]),k,0,Nu)$  
wxplot2d([[discrete,compsol]], [style, [points,2,3]],  
[legend,false], [ xlabel,"x"], [ ylabel,"y(x)-y"])$  
(%t15) << Graphics >>
```

Si hacemos un simple cambio como aumentar en número de subintervalos hasta 40 (el doble) vemos que la precisión mejora notablemente. Como ejercicio realizar el programa para $N = 40$ y comparar los resultados con el caso anterior (dibuja las correspondientes gráficas).

Consideremos ahora el problema

$$y' - 1 - y^2 = 0, \quad y(0) = 0, \quad x \geq 0.$$

Vamos a usar un esquema de 80 puntos en el intervalo $[0, 2]$ y vamos a representar la solución numérica en el intervalo $[0, 0.6]$ en la gráfica 5.7 (izquierda) mediante “•”. Como la solución exacta de la ecuación es $y(x) = \tan x$, vamos a dibujar ambas en la gráfica 5.7 (izquierda) siendo la línea la solución exacta. Se puede ver que prácticamente coinciden.

```
(%i16) kill(x,y)$ kill(f)$  
1:2$ Nu:80$ h:1/Nu$
```

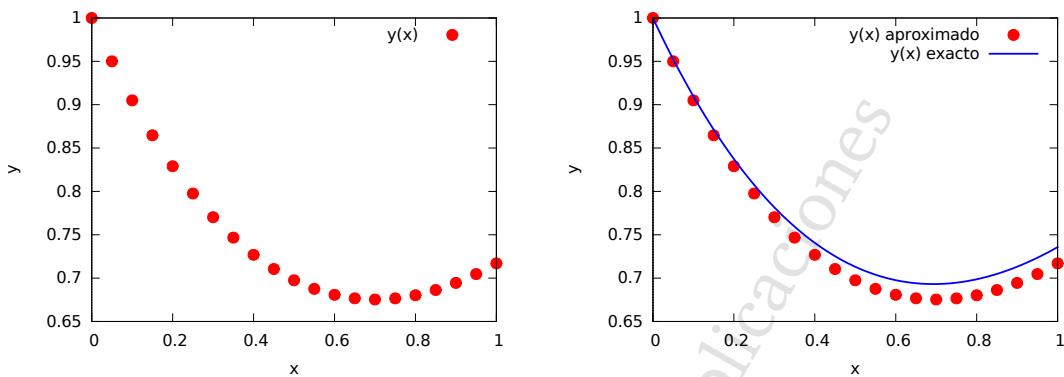


Figura 5.6: Solución numérica para $N = 20$ (●) (izquierda) y comparación con la solución exacta $y' + y = x$, $y(0) = 1$ (línea) (derecha)

```
x[0]:=0$ x[n]:=x[0]+n*h$ y[0]:=0$
define(f[x,y],1+y*y);
y[k]:=y[k-1]+h*f[x[k-1],y[k-1]]$ 
solt:makelist(float([x[k],y[k]]),k,0,Nu)$
wxplot2d([[discrete,solt],tan(x)],[x,0,.6],
          [style, [points,2,2], [lines,1,1]], [xlabel,"x"],
          [ylabel,"y"], [legend,"y(x) aproximado","y(x) real"])$
(%o24) f[x,y]:=x^2+1
(%t27) << Graphics >>
```

¿Qué ocurre si dibujamos las soluciones hasta llegar a $x = 1,52$?

```
(%i28) wxplot2d([[discrete,solt],tan(x)],[x,0,1.52], [xlabel,"x"],
               [ylabel,"y"], [style, [points,2,2], [lines,1,1]],
               [legend,"y(x) aproximado","y(x) real"])$
(%t28) << Graphics >>
```

El resultado está representado en la gráfica 5.7 (derecha). En dicha gráfica se ve claramente que la solución numérica está lejos de la exacta. La razón principal de dicha divergencia es obvia: la solución $\tan x$ no está definida en $x = \pi/2 \approx 1,57$. Una pregunta natural es, por tanto, ¿cómo decidir a priori si el método de Euler (o cualquier otro que usemos) nos está dando la solución correcta? y ¿en qué intervalo podemos asegurar que \hat{y} es una buena aproximación de y ?

Las preguntas anteriores apuntan a la necesidad de tener al menos condiciones suficientes que garanticen la existencia y unicidad de las soluciones de una EDO: los conocidos teoremas de existencia y unicidad de soluciones, que no trataremos aquí pero que el lector puede consultar por ejemplo en [4, 6].

El método de Euler mejorado.

El método de Euler es el más sencillo pero tiene un problema, es un método de orden uno, o sea, es “poco preciso”. ¿Cómo mejorarlo?

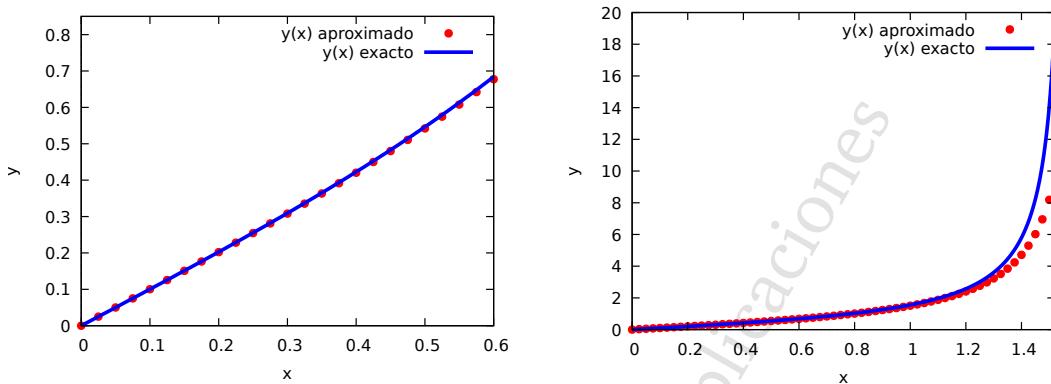


Figura 5.7: Comparación de las soluciones numérica y exacta del PVI $y' - 1 - y^2 = 0$, $y(0) = 0$ en los intervalos $x \in [0, 0,6]$ (izquierda) y $x \in [0, 1,52]$ (derecha).

Una posibilidad es truncar la serie de Taylor (5.3.2) en el tercer orden, de forma que tengamos

$$y_{k+1} = y_k + h f(x_k, y_k) + \left[\frac{\partial f}{\partial x}(x_k, y_k) + f(x_k, y_k) \frac{\partial f}{\partial y}(x_k, y_k) \right] \frac{h^2}{2}, \quad y_0 = y(x_0).$$

La ecuación anterior se conoce como el método de la serie de Taylor de tres términos y, aunque es más preciso que el de Euler (se puede probar que es un método de orden 2), es algo incómodo sobre todo si f es una función “complicada”. Por ello se suele usar una modificación del mismo.

Para ello escribamos la EDO original $y' = f(x, y)$ en el intervalo $x_k, x_k + h$ en su forma integral

$$y(x_{k+1}) = y(x_k) + \int_{x_k}^{x_k+h} f(x, y(x)) dx.$$

Para resolver este problema aproximamos la integral mediante un rectángulo de altura $f(x_k, y_k)$ (ver figura 5.8 izquierda)

$$\int_{x_k}^{x_k+h} f(x, y(x)) dx \approx h f(x_k, y_k),$$

lo que nos conduce nuevamente al esquema de Euler (5.3.3)

Esta aproximación es muy burda. Una mejor aproximación es usar la regla de los trapecios (ver figura 5.8 derecha) para aproximar la integral, es decir

$$\int_{x_k}^{x_k+h} f(x, y(x)) dx \approx \frac{h}{2} [f(x_k, y_k) + f(x_{k+1}, y_{k+1})],$$

de donde se deduce el esquema *implícito*

$$y_{k+1} = y_k + \frac{h}{2} [f(x_k, y_k) + f(x_{k+1}, y_{k+1})], \quad k = 0, 1, \dots, N, \quad y_0 = y(x_0).$$

El esquema anterior es muy incómodo pues hay que resolver la ecuación implícita para hallar y_{k+1} . Una forma de obviar esta dificultad es usar la predicción que da el método de Euler $y_{k+1} = y_k + h f(x_k, y_k)$, de esta forma obtenemos el *método de Euler mejorado*

$$y_{k+1} = y_k + \frac{h}{2} [f(x_k, y_k) + f(x_{k+1}, y_k + h f(x_k, y_k))], \quad k = 0, 1, \dots, N, \quad y_0 = y(x_0).$$

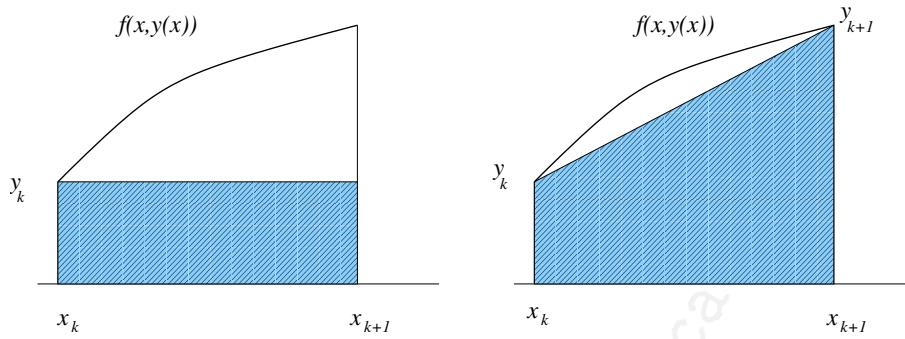


Figura 5.8: Regla del rectángulo (izquierda) y del trapecio (derecha) para aproximar una integral

Como ejemplo resolvamos nuevamente el PVI

$$y' + y = x, \quad y(0) = 1, \quad x \in [0, 1],$$

cuya solución exacta es $y(x) = 2e^{-x} - 1 + x$. Escogeremos una discretización equidistante con paso $h = 1/20$ (20 subintervalos iguales).

```
(%i29) kill(x,ym)$
ym[0]:1$ ym[0]:1$ l:1:Nu:20$ h:1/Nu$
x[0]:0$ x[n]:=x[0]+n*h$
define(f[x,y],-y+x)$
ym[k]:=float(ym[k-1]+(h/2)*(f[x[k-1],ym[k-1]]+
f[x[k], ym[k-1]+h*f[x[k-1],ym[k-1]]]))$
solm:makelist(float([x[k],ym[k]]),k,0,Nu)$
```

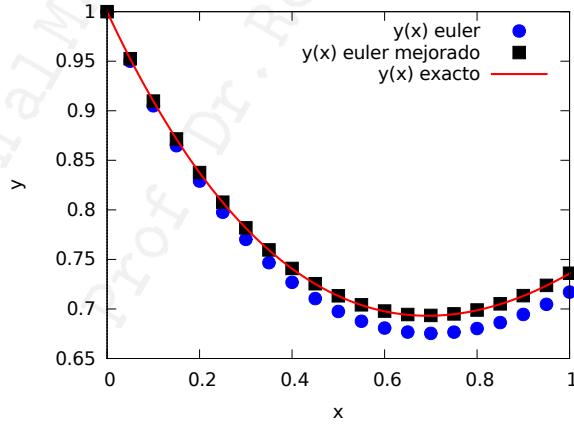


Figura 5.9: Comparación gráfica de las soluciones numéricas y exacta (línea) de $y' + y = x$, $y(0) = 1$ para $N = 20$ usando los métodos de Euler (\bullet) y Euler mejorado (\circ).

Los resultados los vemos en la gráfica 5.9.

```
wxplot2d([[discrete,sol],[discrete,solm],expon(x)],[x,0,1],
```

```
[style, [points,2,2], [points,2,5], [lines,1,1]],
[legend,"y(x) euler","y(x) euler mejorado","y(x) real"],
[xlabel,"x"], [ylabel,"y"])$
(%t40) << Graphics >>
```

k	x_k	$\hat{y}(x_k)$	$y(x_k)$	$\hat{y}(x_k) - y(x_k)$
0	0	1.	1.	0
1.	0.05	0.95125	0.952459	-0.00120885
2.	0.1	0.907314	0.909675	-0.00236077
3.	0.15	0.867958	0.871416	-0.00345845
4.	0.2	0.832957	0.837462	-0.00450443
5.	0.25	0.8021	0.807602	-0.00550115
6.	0.3	0.775186	0.781636	-0.00645092
7.	0.35	0.75202	0.759376	-0.00735595
8.	0.4	0.732422	0.74064	-0.00821835
9.	0.45	0.716216	0.725256	-0.00904012
10.	0.5	0.703238	0.713061	-0.00982318
11.	0.55	0.69333	0.7039	-0.0105693
12.	0.6	0.686343	0.697623	-0.0112803
13.	0.65	0.682134	0.694092	-0.0119578
14.	0.7	0.680567	0.693171	-0.0126034
15.	0.75	0.681515	0.694733	-0.0132186
16.	0.8	0.684853	0.698658	-0.0138047
17.	0.85	0.690467	0.70483	-0.0143632
18.	0.9	0.698244	0.713139	-0.0148955
19.	0.95	0.708079	0.723482	-0.0154026
20.	1.	0.719873	0.735759	-0.0158858

Cuadro 5.2: Solución numérica de la ecuación $y' + y = x$, $y(0) = 1$ con $N = 20$ usando el método de Euler mejorado.

Como ejercicio, construir la matriz de errores similar a la de la tabla 5.1 pero para el caso del método de Euler mejorado (ver resultado en la tabla 5.2).

Hemos de mencionar que la forma que usamos aquí usa los arrays pues usamos las asignaciones de la forma `+ x[n]:=x[0]+n*h`. Ello nos puede llevar a errores si queremos resolver una ecuación distinta, así que es conveniente limpiar la memoria con un `kill(all)`. El problema radica si queremos pintar en un mismo gráfico las soluciones de dos problemas distintos. Entonces, o bien usamos distintas variables para el array, por ejemplo, `+ xx[n]:=xx[0]+n*h`, o bien usamos un método distinto.

Como ejemplo resolveremos el problema de valores iniciales usando el método de Euler mejorado

$$v' = g - \kappa v^r, \quad v(0) = v_0, \quad (5.3.4)$$

que modeliza la velocidad de la caída de un cuerpo en un medio “viscoso” (el aire o el agua). Tomaremos los siguientes valores: $g = 9,8$, $\kappa = 2$, $v_0 = 0$ y $r = 1, 1,5$ y 2 . Lo implementaremos usando una secuencia distinta (y más recomendable que la anterior que corresponde a un array)

```
kill(all)$
x0:0;l:4;Nu:40;h:l/Nu;
x(k):=x0+k*h;
kk:1$ g:9.8$ w:sqrt(kk/g)$
f(x,y,p):=g*(1- w^2*y^p) ;
p:1$
atvalue(y(k),[k=0],1); y(0);
for m:1 thru Nu do atvalue(y(k),[k=m],y(m-1)+h*f(x(m-1),y(m-1),p))$
```

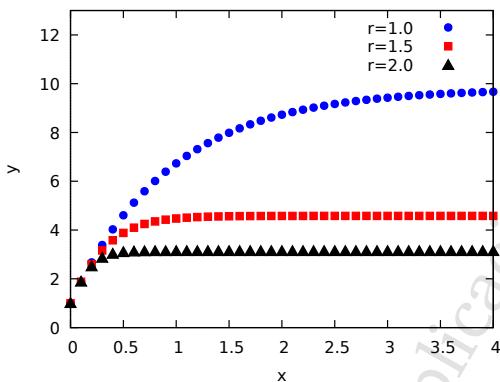


Figura 5.10: Modelización de la caída de un cuerpo. La línea superior representa el caso $r = 1$, la del medio $r = 1,5$ y la inferior $r = 2$.

```

sol1:makelist(float([x(k),y(k)]),k,0,Nu)$
p:1.5$
for m:1 thru Nu do atvalue(y(k),[k=m],y(m-1)+h*f(x(m-1),y(m-1),p))$
sol15:makelist(float([x(k),y(k)]),k,0,Nu)$
p:2$
for m:1 thru Nu do atvalue(y(k),[k=m],y(m-1)+h*f(x(m-1),y(m-1),p))$
sol2:makelist(float([x(k),y(k)]),k,0,Nu)$

```

Los resultados los representaremos en la figura 5.10.

```

wxplot2d([[discrete,sol1],[discrete,sol15],[discrete,sol2]],
[style, [points,2,1], [points,2,2], [points,3,5]], [y,0,13],
[legend,"r=1.0", "r=1.5", "r=2.0"], [xlabel,"x"], [ylabel,"y"])$

```

Antes de terminar este apartado conviene hacer notar que existen muchos métodos para resolver numéricamente las ecuaciones diferenciales. En general conviene que el método usado tenga dos características fundamentales: que sea preciso y estable. Las dos versiones del método de Runge-Kutta que usa MAXIMA cumplen muy bien ambas características no así el método de Euler que hemos descrito en este apartado y que suele ser inestable, no obstante hay problemas donde conviene usar otros métodos adaptados al tipo de problema a resolver, pero estos quedan fuera del objetivo de estas notas.³

³EL lector interesado puede consultar la magnífica monografía de A Quarteroni y F. Saleri, *Cálculo Científico con MATLAB y Octave*, Springer-Verlag, 2006.

Capítulo 6

Resolviendo sistemas de EDOs lineales.

6.1. Sistemas lineales de EDOs

Vamos a dedicar este apartado a los sistemas de EDOs (SEDOs) lineales, i.e., los sistemas de la forma:

$$\begin{cases} y'_1(x) = a_{11}(x)y_1(x) + a_{12}(x)y_2(x) + \cdots + a_{1n}(x)y_n(x) + b_1(x), \\ y'_2(x) = a_{21}(x)y_1(x) + a_{22}(x)y_2(x) + \cdots + a_{2n}(x)y_n(x) + b_2(x), \\ \vdots \\ y'_n(x) = a_{n1}(x)y_1(x) + a_{n2}(x)y_2(x) + \cdots + a_{nn}(x)y_n(x) + b_n(x). \end{cases}$$

El sistema anterior se suele escribir en la forma matricial

$$Y'(x) = A(x)Y(x) + B(x), \quad (6.1.1)$$

donde

$$A(x) = \begin{pmatrix} a_{11}(x) & a_{12}(x) & a_{13}(x) & \cdots & a_{1n}(x) \\ a_{21}(x) & a_{22}(x) & a_{23}(x) & \cdots & a_{2n}(x) \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{n1}(x) & a_{n2}(x) & a_{n3}(x) & \cdots & a_{nn}(x) \end{pmatrix}, \quad B(x) = \begin{pmatrix} b_1(x) \\ b_2(x) \\ \vdots \\ b_n(x) \end{pmatrix},$$

Cuando $B(x) = 0$, o sea cuando todas las componentes del vector B son cero, diremos que el SEDO es homogéneo, y si al menos una componente de B es no nula, no homogéneo. En el caso de un sistema homogéneo se tiene el siguiente resultado:

Teorema 1 Si $Y_1(x), \dots, Y_n(x)$ son soluciones linealmente independientes del sistema $Y' = A(x)Y$, entonces toda solución de $Y' = A(x)Y$ se puede expresar como una única combinación lineal de dichas soluciones. O sea, existen unos únicos coeficientes c_1, \dots, c_n tales que

$$Y(x) = c_1Y_1(x) + c_2Y_2(x) + \cdots + c_nY_n(x).$$

En particular, la solución del PVI, $Y' = A(x)Y$, $Y(x_0) = Y_0$ se expresa de manera única como una combinación lineal de las soluciones del sistema homogéneo correspondiente.

Así que, para encontrar la solución general del SEDO $Y' = A(x)Y$, basta encontrar n soluciones independientes del mismo. Una de las opciones más comunes es buscar la solución de $Y' = A(x)Y$ en la forma

$$Y(x) = e^{\lambda x}v, \quad (6.1.2)$$

donde λ es cierta constante y v un vector constante (independiente de x). Sustituyendo (6.1.2) en (6.1.1) tenemos, como v es constante¹

$$Y'(x) = (e^{\lambda x}v)' = \lambda e^{\lambda x}v \implies \lambda e^{\lambda x}v = A e^{\lambda x}v.$$

Es decir, (6.1.2) es solución del sistema homogéneo si y sólo si λ es un autovalor de A y v su autovector asociado, por tanto para resolver nuestro sistema lineal basta encontrar n autovectores v_1, \dots, v_n linealmente independientes en cuyo caso la solución general es

$$Y(x) = \sum_{k=1}^n c_k e^{\lambda_k x} v_k,$$

donde c_1, \dots, c_n son constantes arbitrarias y $\lambda_k, k = 1, \dots, n$, son los autovalores asociados a los autovectores $v_k, k = 1, \dots, n$.

En caso no homogéneo es algo más complicado pues se precisa del concepto de exponencial de una matriz: Sea A una matriz $n \times n$ y $x \in \mathbb{R}$. Definiremos la función $\exp(xA)$ mediante la serie formal

$$\exp(xA) = \sum_{k=0}^{\infty} \frac{x^k A^k}{k!} = I_n + xA + \frac{x^2 A^2}{2} + \dots + \frac{x^n A^n}{n!} + \dots,$$

donde la convergencia² la entenderemos elemento a elemento. La serie anterior converge para todo $x \in \mathbb{R}$ quienquiera sea la matriz A (de hecho converge en todo \mathbb{C}).

Teorema 2 *La función $\exp(xA)$ satisface las siguientes propiedades:*

1. Para toda matriz A , $\exp(0A) = I_n$ y para todo $x \in \mathbb{R}$ $\exp(xO_n) = I_n$, donde O_n es la matriz nula.
2. Para toda matriz A , $\frac{d}{dx} \exp(xA) = A \exp(xA) = \exp(xA)A$.
3. Para todos $x, t \in \mathbb{R}$ y toda matriz A , $\exp[(x+t)A] = \exp(xA) \exp(tA)$.
4. Para toda matriz A , la inversa $[\exp(xA)]^{-1} = \exp(-xA)$.
5. Para todas las matrices A, B con $AB = BA$, $\exp[x(A+B)] = \exp(xA) \exp(xB)$.
6. Para todo $x \in \mathbb{R}$, $\exp(xI_n) = e^x I_n$.

Dado cualquier vector constante $v \in \mathbb{R}^n$ se cumple

$$\frac{d}{dx} [\exp(xA)v] = A[\exp(xA)v],$$

por tanto $\exp(xA)v$ es solución de la ecuación homogénea $Y' = AY$ y $Y(0) = v$. Si escogemos v sucesivamente como e_i , $i = 1, 2, \dots, n$ obtenemos n soluciones v_1, \dots, v_n

¹Recordemos que la derivada de un vector o una matriz es la derivada término a término.

²Es más apropiado definir el correspondiente espacio normado pero eso se sale de nuestros propósitos.

del PVI, $Y' = AY$, $Y(0) = e_i$ que además son linealmente independientes y por tanto constituyen una base del espacio de soluciones del sistema homogéneo correspondiente.

Otro concepto importante es el de matriz fundamental: Una matriz $V \in \mathbb{R}^{n \times n}$ es una matriz fundamental del sistema $Y' = AY$ si sus n columnas son un conjunto linealmente independiente de soluciones de $Y' = AY$. Obviamente la matriz exponencial es una matriz fundamental del sistema $Y' = AY$.

Vamos a usar MAXIMA para encontrar la solución de un sistema homogéneo así como la exponencial de una matriz. Por sencillez nos centraremos en el caso de matrices de 2x2.

6.2. Resolviendo sistemas con MAXIMA.

Comenzaremos de la forma más simple: usando el comando `desolve` (ver el inicio de la sección 5.1, página 105) podemos resolver cualquier sistema de ecuaciones lineales de forma simbólica (analítica). Lo primero que haremos es introducir la matriz A que necesitaremos más adelante. Para ello se usa la orden `matrix` cuya sintaxis es

```
matrix(fila1,fila2,...,finaN)
```

donde `fila1`, `fila2`, etc. son los vectores filas (listas de números a_1 , a_2 , ... de la forma $[a_1, a_2, \dots, a_M]$)³

Veamos como ejemplo la resolución del sistema

$$Y' = \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix} Y.$$

Para ello hacemos

```
(%i1) A:matrix([1,12],[3,1]);
(%o1) matrix([1,12],[3,1])
(%i2) kill(y1,y2,x)$
      desolve(
      [diff(y1(x),x)=y1(x)+12*y2(x), diff(y2(x),x)=3*y1(x)+y2(x)],
      [y1(x),y2(x)]);
(%o3) [y1(x)=((2*y2(0)+y1(0))*%e^(7*x))/2-((2*y2(0)-y1(0))*%e^(-5*x))/2,
      y2(x)=((2*y2(0)+y1(0))*%e^(7*x))/4+((2*y2(0)-y1(0))*%e^(-5*x))/4]
```

y obtenemos la solución general. Nótese que en la salida de MAXIMA aparecen los valores $y_1(0)$ e $y_2(0)$ que apriori son desconocidos. Si queremos resolver el PVI

$$Y' = \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix} Y, \quad Y(0) = \begin{pmatrix} 1 \\ 2 \end{pmatrix},$$

entonces debemos usar además el comando `atvalue`

```
(%i4) kill(y1,y2,x)$
```

³La salida de `matrix([1,12],[3,1])` es en realidad de la forma $\begin{bmatrix} 1 & 12 \\ 3 & 1 \end{bmatrix}$.

```

atvalue(y1(x),x=0,1)$
atvalue(y2(x),x=0,2)$
desolve(
[diff(y1(x),x)=y1(x)+12*y2(x), diff(y2(x),x)=3*y1(x)+y2(x)] ,
[y1(x),y2(x)]);
(%o7) [y1(x)=(5*%e^(7*x))/2-(3*%e^(-5*x))/2,
      y2(x)=(5*%e^(7*x))/4+(3*%e^(-5*x))/4]

```

Para encontrar la matriz exponencial usaremos la propiedad de que las columnas E_k de la misma son las soluciones de los PVI $E'_k(x) = AE_k(x)$, $E_k(0) = e_k$, donde e_k es la base canónica de \mathbb{R}^n . Así resolvemos primero el PVI

$$Y' = \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix} Y, \quad Y(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

```

(%i8) kill(y1,y2,x)$
atvalue(y1(x),x=0,1)$
atvalue(y2(x),x=0,0)$
col1:desolve(
[diff(y1(x),x)=y1(x)+12*y2(x), diff(y2(x),x)=3*y1(x)+y2(x)] ,
[y1(x),y2(x)]);
(%o11) [y1(x)=%e^(7*x)/2+%e^(-5*x)/2, y2(x)=%e^(7*x)/4-%e^(-5*x)/4]
(%i12) ec1:makelist(second(col1[k]),k,1,length(col1));
(%o12) [%e^(7*x)/2+%e^(-5*x)/2, %e^(7*x)/4-%e^(-5*x)/4]

```

y luego el PVI

$$Y' = \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix} Y, \quad Y(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

```

(%i13) kill(y1,y2,x)$
atvalue(y1(x),x=0,0)$
atvalue(y2(x),x=0,1)$
col2:desolve(
[diff(y1(x),x)=y1(x)+12*y2(x), diff(y2(x),x)=3*y1(x)+y2(x)] ,
[y1(x),y2(x)]);
(%o16) [y1(x)=%e^(7*x)-%e^(-5*x), y2(x)=%e^(7*x)/2+.%e^(-5*x)/2]
(%i17) ec2:makelist(second(col2[k]),k,1,length(col2));
(%o17) [%e^(7*x)-%e^(-5*x), %e^(7*x)/2+.%e^(-5*x)/2]

```

Dado que las salidas $ec1$ y $ec2$ son vectores filas, los convertimos en columna simplemente transponiendo la matriz con el comando `transpose`

```

(%i18) define(expA(x),transpose(matrix(ec1,ec2)));
(%o18) expA(x):=matrix([%e^(7*x)/2+.%e^(-5*x)/2,%e^(7*x)-%e^(-5*x)],
                         [%e^(7*x)/4-.%e^(-5*x)/4,%e^(7*x)/2+.%e^(-5*x)/2])

```

que nos da como salida la matriz exponencial:

$$e^{x \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix}} = \begin{pmatrix} \frac{e^{7x}}{2} + \frac{e^{-5x}}{2} & e^{7x} - e^{-5x} \\ \frac{e^{7x}}{4} - \frac{e^{-5x}}{4} & \frac{e^{7x}}{2} + \frac{e^{-5x}}{2} \end{pmatrix}$$

Finalmente comprobamos que $(e^{xA})' = Ae^{xA}$ y que $e^{0A} = 0$

```
(%i19) expA(0);
(%o19) matrix([1,0],[0,1])
(%i20) ratsimp(diff(expA(x),x)-A.expA(x));
(%o20) matrix([0,0],[0,0])
```

Resolvamos el problema ahora usando la técnica descrita al principio de esta sección, es decir usando los autovalores y autovectores. Para ello usamos el comando `eigenvectors` (ver página 75). La salida de este comando, como ya vimos, son dos listas, la primera de ellas son dos vectores fila, el primero contiene los autovectores y el segundo la correspondiente multiplicidad algebraica de los mismos. La segunda lista es una lista de vectores fila que se corresponden con los autovalores. Así, para nuestra matriz $\begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix}$ tenemos

```
(%i21) vec:eigenvectors(A);
(%o21) [[[[-5,7],[1,1]],[[[1,-1/2]],[[1,1/2]]]]
(%i22) vec[1]; vec[2];
(%o22) [[-5,7],[1,1]]
(%o23) [[[1,-1/2]],[[1,1/2]]]
```

La primera lista $[-5, 7], [1, 1]$ nos indica que el autovalor $\lambda = -5$ tiene multiplicidad 1 y que el $\lambda = 7$ tiene multiplicidad 1. Además, a $\lambda = -5$ le corresponde el autovector $(1, -1/2)^T$ y a $\lambda = 7$ el autovector $(1, 1/2)^T$, respectivamente. Para extraer los autovalores y autovectores por separado usamos los `[]`. Así, si hacemos `vec[1]` obtenemos la primera de las dos listas (la de los autovalores y multiplicidades), `vec[1][1]` nos da el primer elemento de esa lista, o sea la de los autovalores, y `vec[1][1][2]` nos imprime el segundo autovalor. De forma similar sacamos los correspondientes autovectores lo que nos permitirá definir la dos soluciones linealmente independientes $s_1(x)$ y $s_2(x)$

```
(%i24) av1:vec[1][1][1]; av2:vec[1][1][2];
      v1:vec[2][1][1]; v2:vec[2][2][1];
(%o24) -5
(%o25) 7
(%o26) [1,-1/2]
(%o27) [1,1/2]
(%i28) s1(x):=%e^(av1*x)*v1$ s1(x);
      s2(x):=%e^(av2*x)*v2$ s2(x);
(%o29) [%e^(-5*x),-%e^(-5*x)/2]
(%o31) [%e^(7*x),%e^(7*x)/2]
```

es decir

$$Y_1(x) = \begin{pmatrix} e^{-5x} \\ -\frac{1}{2}e^{-5x} \end{pmatrix}, \quad Y_2(x) = \begin{pmatrix} e^{7x} \\ \frac{1}{2}e^{7x} \end{pmatrix}.$$

Lo anterior nos permite definir una matriz fundamental

$$V(x) = [Y_1(x) Y_2(x)] = \begin{pmatrix} e^{-5x} & e^{7x} \\ -\frac{1}{2}e^{-5x} & \frac{1}{2}e^{7x} \end{pmatrix}.$$

```
(%i32) define(v(x),transpose(matrix(s1(x),s2(x))));
(%o32) v(x):=matrix([\%e^(-5*x),\%e^(7*x)],[-\%e^(-5*x)/2,\%e^(7*x)/2])
```

Para obtener la matriz exponencial usamos la fórmula

$$\exp(xA) = V(x)V^{-1}(0).$$

Además comprobamos que, efectivamente, $(e^{xA})' = Ae^{xA}$ y $e^{0A} = 0$ y que obviamente es la misma matriz que obtuvimos antes

```
(%i33) define(e(x),v(x).invert(v(0)));
(%o33) e(x):=matrix([%e^(7*x)/2+%e^(-5*x)/2,%e^(7*x)-%e^(-5*x)],
                   [%e^(7*x)/4-%e^(-5*x)/4,%e^(7*x)/2+;%e^(-5*x)/2])
(%i34) ratsimp(diff(e(x),x)-A.e(x));
(%o34) matrix([0,0],[0,0])
(%i35) expA(x)-e(x);
(%o35) matrix([0,0],[0,0])
```

Dado que ya sabemos calcular la matriz exponencial ahora podemos resolver fácilmente el problema no homogéneo. Para ello usamos el resultado

Teorema 3 La solución del problema de valores iniciales $Y'(x) = AY(x) + B(x)$, $Y(x_0) = Y_0$, cualquiera sea $Y_0 \in \mathbb{R}^n$ existe y es única y se expresa mediante la fórmula

$$Y(x) = \exp[(x - x_0)A]Y_0 + \int_{x_0}^x \exp[(x - t)A]B(t)dt. \quad (6.2.1)$$

Vamos por tanto a resolver el sistema

$$Y' = \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix} Y + \begin{pmatrix} e^{-x} \\ 0 \end{pmatrix}, \quad Y(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Ante todo definimos el término independiente B (usaremos la b), luego multiplicamos $e^{(x-t)A} \cdot B(t)$ y definimos la función $\int_0^x \exp[(x-t)A]B(t)dt$. Aquí es importante recordar que la multiplicación⁴ habitual de matrices en MAXIMA se ejecuta con “.”.

```
(%i36) b:transpose([exp(-t),0]);
(%o36) matrix([%e^(-t)], [0])
(%i37) ratsimp(e(x-t).b); expand(%);
define(sol(x),expand(integrate(% , t , 0 , x)));
(%o37) matrix([(e^(-5*x-8*t)*(e^(12*x)+e^(12*t)))/2 ,
               [(e^(-5*x-8*t)*(e^(12*x)-e^(12*t)))/4]])
(%o38) matrix([(e^(7*x-8*t)/2+e^(4*t-5*x))/2 ,
               [%e^(7*x-8*t)/4-%e^(4*t-5*x)/4]])
(%o39) sol(x):=matrix([(e^(7*x)/16+e^(-x)/16-%e^(-5*x)/8] ,
                       [%e^(7*x)/32-(3*e^(-x))/32+e^(-5*x)/16])
```

Luego calculamos $\exp[(x)A]Y_0$ y definimos la solución según la fórmula (6.2.1)

⁴Recordemos que las operaciones $A \cdot B$ con $A * B$ son distintas. Si bien la primera es la multiplicación habitual de matrices, la segunda es una multiplicación elemento a elemento. Por ejemplo

$$A = \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad A \cdot B = \begin{pmatrix} 13 & 13 \\ 4 & 4 \end{pmatrix} \text{ pero } A * B \begin{pmatrix} 1 & 12 \\ 3 & 1 \end{pmatrix}.$$

```
(%i40) e(x).transpose([0,1]);
(%o40) matrix([\%e^(7*x)-\%e^(-5*x)], [\%e^(7*x)/2+\%e^(-5*x)/2])
(%i41) define(solt(x),sol(x)+e(x).transpose([0,1]));
(%o41) solt(x):=matrix([(17*\%e^(7*x))/16+\%e^(-x)/16-(9*\%e^(-5*x))/8],
 [(17*\%e^(7*x))/32-(3*\%e^(-x))/32+(9*\%e^(-5*x))/16])
```

Finalmente, comprobamos que nuestra solución efectivamente satisface la ecuación original con las condiciones iniciales.

```
(%i42) solt(0);
(%o42) matrix([0], [1])
(%i43) ratsimp(diff(solt(x),x)-A.solt(x));
(%o43) matrix([\%e^(-x)], [0])
```

El caso de autovalores múltiples es más complicado y lo omitiremos ya que su resolución no aporta nada nuevo desde el punto de vista de MAXIMA y además se puede resolver por el primer método sin problemas.

6.3. Un ejemplo de un sistema no lineal: el sistema Lotka-Volterra.

Como culminación de este apartado resolvamos el siguiente problema de valores iniciales:

$$\frac{du}{dx} = u(1 - v), \quad \frac{dv}{dx} = \alpha v(u - 1), \quad u(x_0) = u_0, \quad v(x_0) = v_0. \quad (6.3.1)$$

Nótese que este es un sistema acoplado⁵ de ecuaciones diferenciales no lineales. Intentemos estudiar sus soluciones. Ante todo notemos que dividiendo ambas ecuaciones se tiene que

$$\frac{du}{dv} = \alpha \frac{v(u-1)}{u(1-v)},$$

cuya solución es $\alpha u + v - \log u^\alpha v = H = \text{const.}$ Si dibujamos los valores de u respecto a v obtenemos las *trayectorias de las soluciones del sistema*. Así pues, si $H > 1 + \alpha$ (que es el mínimo de H que se alcanza para $u = v = 1$), entonces las *trayectorias* definidas por $\alpha u + v - \log u^\alpha v = H$ son cerradas lo que implica que u y v son funciones periódicas. Dichas trayectorias están representadas en el gráfico 6.1. Para construir dicha gráfica necesitamos el comando `implicit_plot` (pues v está definida en función de u de forma implícita) contenido el paquete `implicit_plot` que ya describimos en el apartado 3 y cuya sintaxis es

```
implicit_plot(F(u,v)=0, [u, uini, ufin], [v, vini, vfin])
```

donde $F(u, v) = 0$ es la ecuación implícita que queremos dibujar en los intervalos de las variables u desde u_{ini} hasta u_{fin} y v desde v_{ini} hasta v_{fin} . Así, hacemos

```
(%i44) load(implicit_plot)$
(%i45) implicit_plot ([
```

Más aún si v es un vector 2×1 , $A.v$ esta bien definido pero $A * v$ no.

⁵Quiere decir que las ecuaciones no son independientes.

```
0.5*u+v-log(v*u^0.5)=1.6, 0.5*u+v-log(v*u^0.5)=2, 0.5*u+v-log(v*u^0.5)=2.3,
0.5*u+v-log(v*u^0.5)=2.5, 0.5*u+v-log(v*u^0.5)=3, 0.5*u+v-log(v*u^0.5)=3.5],
[u,0.01,7], [v,0.01,6], [legend, "H1=1.6", "H2=2.0", "H3=2.3", "H4=2.5",
" H5=3.0", "H6=3.5"] )$
```

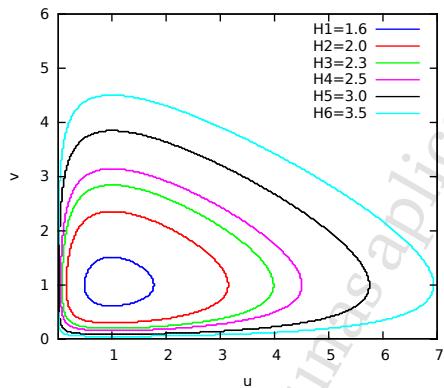


Figura 6.1: Trayectorias definidas por el sistema (6.3.1) para $\alpha = 1/2$ y $H = 1.6, 2, 2.3, 2.5, 3$ y 3.5 .

Para el sistema (6.3.1) no funciona ninguno de los métodos *analíticos* discutidos anteriormente así que tenemos que resolverlo numéricamente. Para ello usaremos la orden `rk` que discutimos en la sección 5.2, página 5.2, pero ahora para un sistema de ecuaciones diferenciales cuya sintaxis es, en general,

```
rk([ED01, ..., ED0m], [y1, ..., ym], [vini1, ..., vinitm], [x, x0, xmax, paso])
```

La salida de este comando es una lista con tantas entradas como iteraciones y cada entrada de esa lista es a su vez una lista de $m + 1$ elementos donde el primero es el valor de la x el segundo es el valor de y_1 evaluada en dicho x , el tercero el de y_2 en x y así sucesivamente.

Así pues, tenemos

```
(%i6) load(dynamics)$
(%i7) kill(a,u,v,sol)$ a:0.5;
           sol:rk([u*(1-v),a*v*(u-1)], [u,v], [0.4,0.2], [t,0,40,0.02])$ 
(%o8) 0.5
```

A continuación construimos las listas que vamos a representar: la los valores de (x_k, u_k) y (x_k, v_k) , respectivamente. Como antes, usamos el comando `makelist`.

```
(%i10) u:makelist([sol[k][1],sol[k][2]],k,1,length(sol))$ 
           v:makelist([sol[k][1],sol[k][3]],k,1,length(sol))$ 
           ciclo:makelist([sol[k][2],sol[k][3]],k,1,length(sol))$
```

Finalmente, representamos las gráficas de $u(x)$, $v(x)$, por separado

```
(%i13) wxplot2d([discrete,u],[legend, "u"],
               [xlabel, "t"], [ylabel, "u"], [color,blue])$ 
           wxplot2d([discrete,v],[legend, "v"],
               [xlabel, "t"], [ylabel, "v"], [color,red])$ 
(%t13) << Graphics >>
(%t14) << Graphics >>
```

o en un único gráfico (ver Fig.6.2).

```
(%i15) wxplot2d([[discrete,u],[discrete,v]], [legend, "u", "v"],
    [xlabel, "t"], [ylabel, "u,v"] )$  

(%t15) << Graphics >>
```

Para terminar podemos dibujar las trayectorias $u(v)$

```
(%i16) wxplot2d([discrete,ciclo], [legend, "u vs v"],
    [xlabel, "u"], [ylabel, "v"], [color,magenta] )$  

(%t16) << Graphics >>
```

Las dos últimas gráficas están representadas en la figura 6.2.

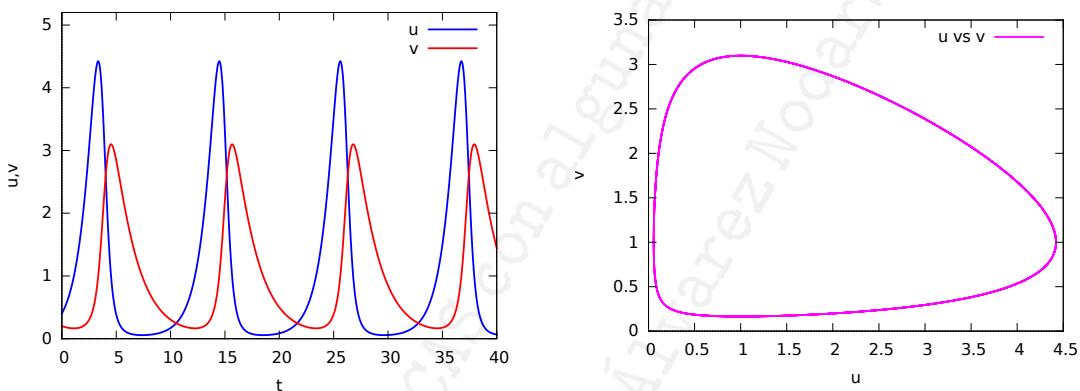


Figura 6.2: Las soluciones u y v del sistema (6.3.1) para $\alpha = 1/2$, $u(0) = 0,4$ y $v(0) = 0,2$.

6.3.1. Representando campos de direcciones para EDOs.

Vamos a discutir en este apartado una herramienta muy útil para el estudio de las EDOs: los denominados campos de direcciones. El campo de direcciones es esencialmente una gráfica donde se representan las soluciones de nuestra ecuación y donde se incluyen, mediante flechas, las pendientes de las tangentes a los distintos puntos de curva que define la solución (trayectoria).

Uno de los comandos más versátiles es `plotdf` lo que requiere tener instalado el XMAXIMA (no basta con MAXIMA o el Wxmaxima). La orden tiene un sinnúmero de opciones y la ventana que se abre es interactiva. Pinchando en distintas zonas aparecen las trayectorias correspondientes. Como ejemplo veamos el campo de direcciones del sistema (6.3.1). La orden que viene a continuación solo pinta la trayectoria cerrada que está en el primer cuadrante, las restantes se obtienen al pinchar sobre la ventana emergente. Así, representamos el campo de direcciones de nuestro sistema (6.3.1)

```
(%i17) kill(all)$  

(%i1) load("plotdf")$  

(%i2) plotdf([x*(1-y),a*y*(x-1)], [parameters,"a=.5"],  

    [trajectory_at,0.4,0.2], [xradius,5], [yradius,5])$
```

Finalmente, si incluimos la opción `sliders` nos aparece en la ventana una paleta para variar el parámetro a del sistema.

```
(%i3) plotdf([x*(1-y),a*y*(x-1)], [parameters,"a=.5"],
[sliders,"a=0.2:0.8"], [trajectory_at,0.6,3.1],
[x,-2,8], [yradius,5]);
```

En la figura 6.3 vemos las salidas de los dos comandos anteriores,

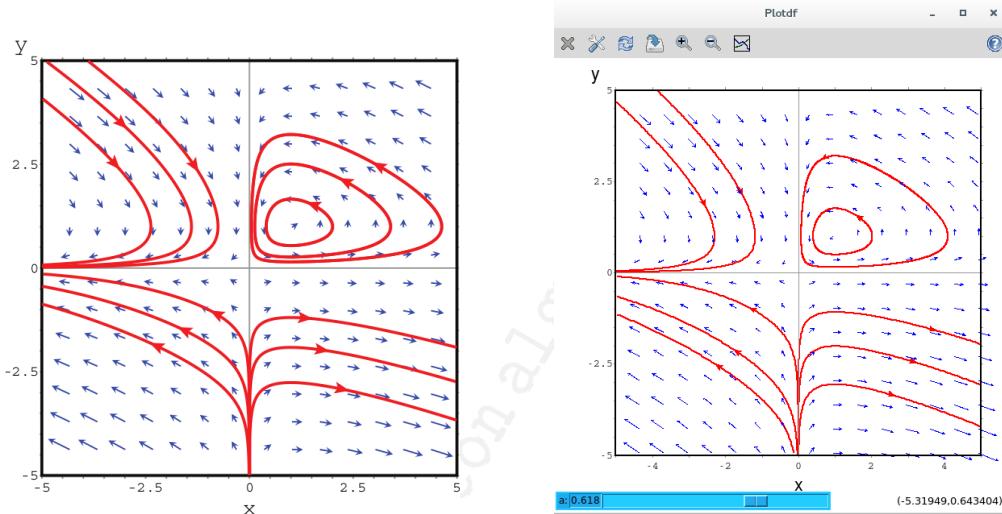


Figura 6.3: Campo de direcciones para el modelo para el sistema (6.3.1). A la derecha vemos una captura de la pantalla del `plotdf` de openmath (contenido en XMAXIMA)

MAXIMA también cuenta con el paquete `drawdf` para representar campos de direcciones. Aparte de las distintas opciones propias de `drawdf` también se pueden usar las opciones heredadas del paquete `draw`. Veamos como ejemplo dos gráficas del campo de direcciones del sistema (6.3.1) que ya dibujamos con `plotdf`. La primera secuencia tiene pocas opciones mientras que en la segunda agregamos una serie de opciones extra. Dada la gran cantidad de las mismas es recomendable revisar la documentación general del paquete `drawdf`. Las salidas las podemos ver en la figura 6.4.

```
(%i4) kill(all)$
(%i1) load("drawdf")$
(%i2) drawdf([x*(1-y),.5*y*(x-1)], [x,-5,5], [y,-5,5], line_width = 3,
solns_at([0.6,3.1],[0.6,2.1],[-0.6,3.1],[-0.6,2.1],[0.6,-3.1],
[0.6,-2.1],[-0.6,-3.1],[-0.6,-2.1]), points_joined = false,
point_type = filled_circle, point_size = 3,color=blue,duration = 30,
points([[0.6,3.1],[-0.6,3.1],[0.6,-3.1],[-0.6,-3.1],
[0.6,2.1],[-0.6,2.1],[0.6,-2.1],[-0.6,-2.1]]))$ 
(%i3) drawdf([x*(1-y),.5*y*(x-1)], [x,-5,5], [y,-5,5],
line_width = 3, soln_arrows = false, field_degree=1,
solns_at([0.6,3.1],[0.6,2.1],[-0.6,3.1],[-0.6,2.1],[0.6,-3.1],
[0.6,-2.1],[-0.6,-3.1],[-0.6,-2.1]),field_grid = [25,23],
points_joined = false, point_type = filled_circle,
point_size = 3,color=blue,duration = 30,
```

```
points([[0.6,3.1],[-0.6,3.1],[0.6,-3.1],[-0.6,-3.1],
[0.6,2.1],[-0.6,2.1],[0.6,-2.1],[-0.6,-2.1]]),
xlabel = "x", ylabel = "y", background_color = light_gray,
title = "Campo de direcciones del sistema Lotka-Volterra")$
```

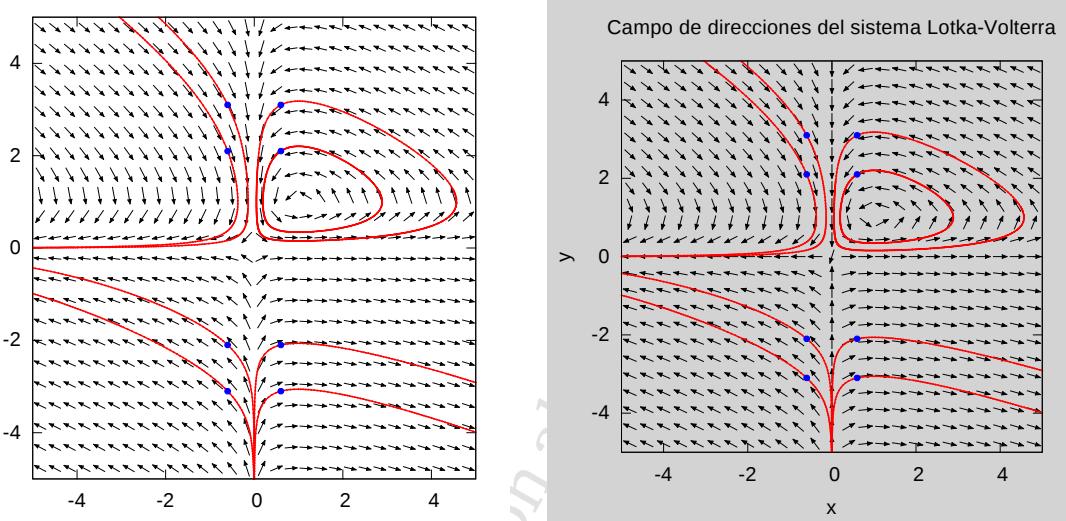


Figura 6.4: Campo de direcciones para el modelo para el sistema (6.3.1) usando `drawdf`. A la izquierda sin opciones y a la derecha con varias opciones extra.

Por último mostramos como dibujar con diferentes colores varias trayectorias a la vez e incluyendo el nombre de cada una de ellas de forma automática:

```
(%i4) colors : ['red,'blue,'purple,'orange,'green]$  
wxdrawdf([x*(1-y),.5*y*(x-1)], [x,-0.5,5], [y,-0.5,5],  
line_width = 3, soln_arrows = false, field_degree=1,  
makelist([key = concat("sol. en (2," ,k,"/6)"),  
color = colors[k], soln_at(2, k/6)], k,1,5));  
(%t5) << Graphics >>
```

La clave es usar el comando `concat`. El resultado lo vemos en la figura 6.5, donde a la derecha además se usa la opción `show_field=false` que elimina las líneas del campo de direcciones.

Entre las opciones que tiene `drawdf` conviene destacar las siguientes: `soln_at(x0,y0)` que dibuja la solución particular (trayectoria) que pasa por (x_0, y_0) , (si se quiere pintar varias soluciones también se puede usar la opción `solns_at([x0,y0], [x1,y1], etc)`), `soln_arrows=true` para que dibuje las flechas en las trayectorias (soluciones), `field_arrows=true` para que dibuje las flechas del campo de direcciones, `field_color=color` para controlar el color del campo de direcciones, `duration=num` para decidir la longitud del intervalo de la integración numérica cuyo valor por defecto es `num=10` y `direction=opción`, que tiene dos posibilidades, `forward` o `reverse`, es decir si integra hacia delante o hacia atrás en el tiempo, respectivamente (esta última opción no está explicada en el manual [1]).

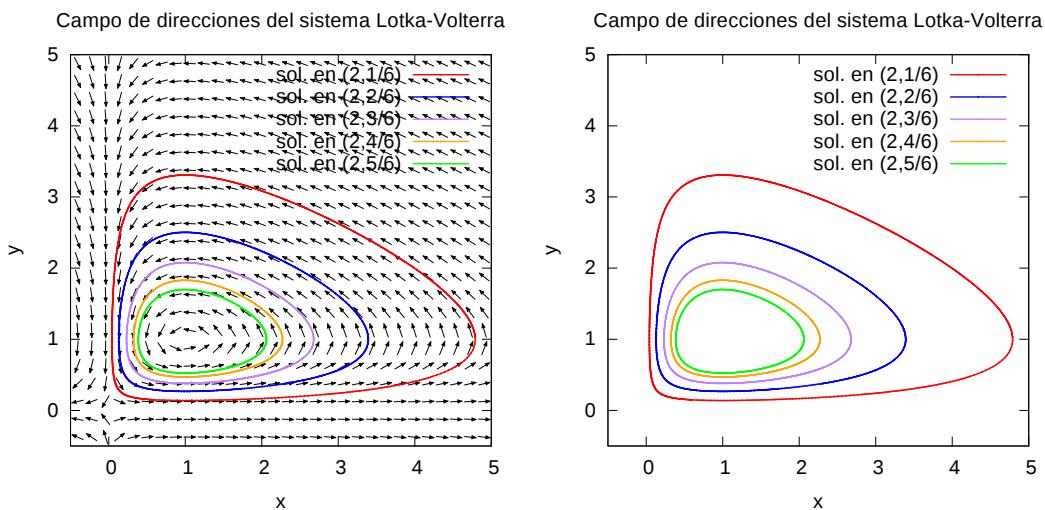


Figura 6.5: Campo de direcciones para el modelo para el sistema (6.3.1) con 5 trayectorias en el primer cuadrante. A la derecha se representa únicamente las trayectorias.

6.4. Resolviendo EDOs lineales de orden n .

Consideremos ahora otro grupo muy importante de EDOs: las EDOs lineales con coeficientes constantes de orden mayor que 1 definidas por la ecuación:

$$y^{(n)}(x) + a_{n-1}(x)y^{(n-1)}(x) + \cdots + a_1(x)y'(x) + a_0(x)y(x) = f(x). \quad (6.4.1)$$

Cuando $f(x) \equiv 0$, diremos que la EDO (6.4.1) es homogénea, en caso contrario es no homogénea. Si además imponemos que la solución y sea tal que

$$y(x_0) = y_0, \quad y'(x_0) = y'_0, \quad \dots \quad y^{(n-1)}(x_0) = y_0^{(n-1)},$$

entonces diremos que y es la solución del correspondiente problema de valores iniciales.

Para estudiar las propiedades de las EDOs lineales de orden n vamos a introducir unas nuevas funciones $z_1(x), \dots, z_n(x)$ de forma que

$$\left. \begin{array}{l} z_1(x) = y(x) \\ z_2(x) = y'(x) \\ z_3(x) = y''(x) \\ \vdots \\ z_{n-1}(x) = y^{(n-2)}(x) \\ z_n(x) = y^{(n-1)}(x) \end{array} \right\} \Rightarrow \begin{array}{l} \frac{dz_n}{dx} = -a_{n-1}z_n - a_{n-2}z_{n-1} - \cdots - a_0z_1 + f, \\ \frac{dz_{n-1}}{dx} = z_n, \\ \vdots \\ \frac{dz_1}{dx} = z_2. \end{array}$$

Es decir, una EDO del tipo (6.4.1) es equivalente al siguiente sistema lineal de ecuaciones:

$$\frac{d}{dx} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_{n-1} \\ z_n \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{n-2} & -a_{n-1} \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_{n-1} \\ z_n \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ f(x) \end{pmatrix},$$

o en forma matricial $Z'(x) = AZ(x) + F(x)$. Nótese que cuando $f(x) \equiv 0$, el sistema anterior se transforma en un sistema homogéneo.

Lo anterior tiene una implicación evidente: toda la teoría de las ecuaciones lineales (6.4.1) se puede construir a partir de la teoría de los sistemas lineales y por tanto el estudio con MAXIMA del apartado anterior se puede usar para resolver este tipo de ecuaciones. No obstante, para el caso $N = 2$, caso de especial relevancia por sus múltiples aplicaciones prácticas, MAXIMA cuenta con el comando `ode2`, que ya usamos antes, y que combinado con el comando `ic2` resuelve ecuaciones lineales de orden 2. La orden sintaxis de la orden `ic2` es

```
ic2(solución, valor de x, valor de y, valor de y')
```

donde `solución` es la solución general (salida) que da el comando `ode2` y `valor de x`, `valor de y` y `valor de y'`, son los valores que toma la y y la y' cuando $x = x_0$, i.e., los valores iniciales.

Veamos algunos ejemplos sencillos. Comencemos resolviendo el PVI $y'' + 2y' + y = x^2 - 1$, $y(0) = 0$, $y'(0) = 2$.

```
(%i1) eq1:'diff(y,x,2)-2*'diff(y,x,1)+2*y=x^2-1;
(%o1) 'diff(y,x,2)-2*('diff(y,x,1))+2*y=x^2-1
(%i2) ode2(eq1,y,x);
(%o2) y=%e^x*(%k1*sin(x)+%k2*cos(x))+(x^2+2*x)/2
(%i3) ic2(%,,x=0,y=0,'diff(y,x)=2);
(%o3) y=%e^x*sin(x)+(x^2+2*x)/2
```

También podemos usar el comando `desolve`, aunque en este caso tenemos que escribir explícitamente tanto la variable dependiente como la independiente. En conjunto con la orden `atvalue` también nos da la correspondiente solución:

```
(%i4) eq2:'diff(y(x),x,2)-2*'diff(y(x),x,1)+2*y(x)=x^2-1;
(%o4) 'diff(y(x),x,2)-2*('diff(y(x),x,1))+2*y(x)=x^2-1
(%i5) atvalue(y(x),x=0,0)$
atvalue('diff(y(x),x),x=0,2)$
desolve(eq2,y(x));
(%o7) y(x)=%e^x*sin(x)+x^2/2+x
```

Intentemos lo mismo con la EDO $x(y')^2 - (1 - xy)y' + y = 0$. Si usamos el comando `ode2` el sistema reconoce que no es una EDO lineal y da error.

```
(%i8) eq3:x*'diff(y,x)^2-(1+x*y)*'diff(y,x)+y=0;
(%o8) x*('diff(y,x,1))^2-(x*y+1)*('diff(y,x,1))+y=0
(%i9) ode2(eq3,y,x);
(%t9) x*('diff(y,x,1))^2-(x*y+1)*('diff(y,x,1))+y=0
"first order equation not linear in y"
(%o9) false
```

La orden `desolve` devuelve su típico `ilt(...)` cuando no es capaz de resolverlo. En este

caso podemos recurrir al paquete `contrib_ode`⁶ y la orden homónima que nos encuentran dos posibles soluciones:

```
(%i10) load('contrib_ode)$
define: warning: redefining the built-in function lcm
(%i11) contrib_ode(eq3,y,x);
(%t11) x*(’diff(y,x,1))~2-(x*y+1)*(’diff(y,x,1))+y=0
      "first order equation not linear in y"
(%o11) [y=log(x)+%c,y=%c*e^x]
```

Como ejercicio comprobar que efectivamente las funciones obtenidas son solución de la EDO correspondiente.

Otra opción es resolver numéricamente las correspondientes ecuaciones diferenciales, como es el caso de la ecuación $y'' - 2xy' + 2y = 0$, que es imposible resolver usando los tres comandos anteriores.

```
(%i12) kill(all)$
(%i1) eq4:'diff(y(x),x,2)-2*x*'diff(y(x),x,1)+2*y(x)=0;
(%o1) 'diff(y(x),x,2)-2*x*(’diff(y(x),x,1))+2*y(x)=0
(%i2) ode2(eq4,y,x);
(%t2) 'diff(y(x),x,2)-2*x*(’diff(y(x),x,1))+2*y(x)=0
      "not a proper differential equation"
(%o2) false
(%i5) atvalue(y(x),x=0,1)$ atvalue('diff(y(x),x),x=0,0)$
      desolve(eq4,y(x));
(%o5) y(x)=ilt( ... )
(%i6) load('contrib_ode)$
(%i7) contrib_ode(eq4,y,x);
(%o7) false
```

En este caso, al igual que en todos, podemos recurrir al comando `rk` del paquete `dynamics` (ver página 6.3) pero antes hay que convertir nuestra EDO de orden 2 en un sistema lineal. Para ello hacemos el cambio $z = y'$ de donde se tiene

$$y'' - 2xy' + 2y = 0 \Leftrightarrow \begin{cases} y' = z, \\ z' = 2xz - y. \end{cases}$$

Así pues tenemos

```
(%i8) load(dynamics)$
(%i9) sol:rk([z,2*x*y-y],[y,z],[1,0],[x,0,2,0.02])$ 
      valy:makelist([sol[k][1],sol[k][2]],k,1,length(sol))$ 
      wxplot2d([discrete,valy],[legend, "y"], 
              [xlabel, "x"], [ylabel, "x"], [color,blue])$ 
(%t11) << Graphics >>
```

donde la solución está representada en la gráfica 6.6 (izquierda).

⁶Este paquete contiene algoritmos que generalizan los que incluye la orden `ode2` para resolver ecuaciones lineales y no lineales de orden 2. Se prevé que en futuras ediciones de MAXIMA pase a ser parte del núcleo del programa.

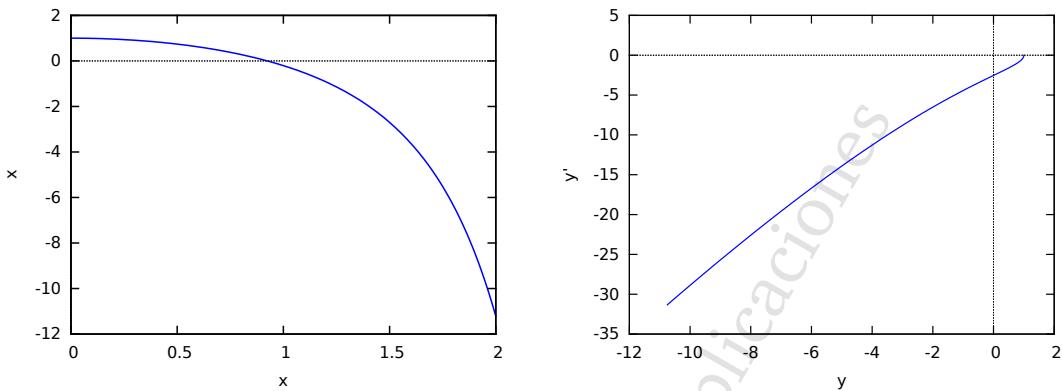


Figura 6.6: La solución (izquierda) del PVI $y'' - 2xy' + 2y = 0$ con $y(0) = 1$ e $y'(0) = 0$ en el intervalo $[0, 2]$ y su correspondiente diagrama (y, y') (derecha).

En el caso concreto de ecuaciones diferenciales de orden 2 del tipo

$$y'' = f(x, y, y'), \quad y(x_0) = y_0, \quad y'(x_0) = y'_0, \quad (6.4.2)$$

MAXIMA cuenta además con la orden `runge2`, cuya sintaxis es

```
runge2(f, x0, xf, h, y0, dy0)
```

donde `f` es la función $f(x, y, y')$ de la derecha en (6.4.2), `x0` y `xf` son los extremos del intervalo $[x_0, x_f]$ donde vamos a calcular la solución, `h` es el paso de integración numérica y los números `y0` y `dy0` son los valores iniciales de la función y la primera derivada en el punto inicial `tt`. Para usarlo hay que usar el paquete `diffeq` que contiene a la orden `runge1` que usamos en el apartado 5.2.

Como ejemplo resolveremos la ecuación $y'' - 2xy' + 2y = 0$, que resolvimos antes usando el comando `rk`. Además representaremos también la gráfica de y , y' y el diagrama (y, y') (este último se representa en la gráfica 6.6(derecha)):

```
(%i12) load(diffeq)$
(%i13) f(t,y,dy) := 2*x*dy - 2*y $ 
(%i14) res: runge2(f,0,2,0.01,1,0)$
      wxplot2d([discrete,res[1],res[2]], [legend, "y"],
                [xlabel, "x"], [ylabel, "x"], [color, blue])$ 
(%i16) wxplot2d([discrete,res[1],res[3]], [legend, "y"],
                [xlabel, "x"], [ylabel, "y'"], [color, blue])$ 
(%i17) wxplot2d([discrete,res[2],res[3]], [legend, "y"],
                [xlabel, "y"], [ylabel, "y'"], [color, blue])$
```

6.4.1. Las funciones especiales de la física-matemática.

Una ecuación muy común en la modelización de fenómenos naturales es la siguiente:

$$p(x)y'' + q(x)y' + r(x)y = f(x), \quad (6.4.3)$$

donde $p(x)$, $q(x)$, $r(x)$ y $f(x)$ son funciones lo suficientemente buenas.

Dentro de las ecuaciones del tipo anterior tenemos las *funciones especiales de la física matemática* que son solución de la ecuación

$$\sigma(x)y'' + \tau(x)y' + \lambda y = 0,$$

donde σ y τ son polinomios de grado 2, 1, respectivamente, y λ es una constante [5]. Las soluciones de la ecuación anterior se suelen denominar funciones de tipo hipergeométrico y juegan un papel muy importante en un sinnúmero de aplicaciones en ciencias e ingeniería.

Es conveniente saber que MAXIMA contiene una colección muy completa de funciones especiales que incluye no sólo a los polinomios ortogonales clásicos de Jacobi, Laguerre, Hermite, Legendre, Gegenbauer —o ultraesféricos— (incluidos en el paquete *orthopoly*) sino a muchas otras. Entre ellas están las funciones Gamma, Beta, de Bessel, Hankel, Airy, de Legendre, hipergeométrica, entre otras. La nomenclatura y normalización de las mismas corresponden a las del manual clásico de M. Abramowitz y I. A. Stegun [2]. Para más detalle consultar los apartados de Funciones especiales y del paquete *orthopoly* del manual [1].

Veamos un ejemplo de como se puede trabajar con funciones especiales con MAXIMA. Para ello vamos a intentar resolver la EDO siguiente:

$$x(1-x)y''(x) + \left(\frac{3}{2} - 2x\right)y'(x) + 2y = 0. \quad (6.4.4)$$

En este caso los comandos usuales no funcionan.

```
(%i1) eqh:x*(1-x)*'diff(y,x,2)+(3/2-2*x)*'diff(y,x)+2*y=0;
(%o1) (1-x)*x*('diff(y,x,2))+(3/2-2*x)*('diff(y,x,1))+2*y=0
(%i2) ode2(eqh,y,x);
(%o2) false
```

Así que usaremos el paquete *contrib_ode*.

```
(%i3) load('contrib_ode)$
(%i4) odelin(eqh,y,x);
(%o4) {gauss[a](-1,2,3/2,x),gauss[b](-1,2,3/2,x)}
(%i5) contrib_ode(eqh,y,x);
(%o5) [y=gauss[b](-1,2,3/2,x)*%k2+gauss[a](-1,2,3/2,x)*%k1]
```

Aquí la orden *odelin* genera las soluciones linealmente independientes de la ecuación diferencial, mientras que la salida de *contrib_ode* es la solución general de la misma.

En nuestro ejemplo vemos que el comando *contrib_ode* da como salida dos funciones *gauss_a* y *gauss_b*, que no son más que las funciones y_1 e y_2 definidas por

$$y_1(x) = {}_2F_1\left(\begin{array}{c} \alpha, \beta \\ \gamma \end{array} \middle| x\right) = \sum_{k=0}^{\infty} \frac{(\alpha)_k (\beta)_k}{(\gamma)_k} \frac{x^k}{k!}, \quad y_2(x) = x^{1-\gamma} {}_2F_1\left(\begin{array}{c} \alpha - \gamma + 1, \beta - \gamma + 1 \\ 2 - \gamma \end{array} \middle| x\right),$$

respectivamente, donde por $(a)_n$ se denota el producto

$$(a)_0 = 1, \quad (a)_n = a(a+1)\cdots(a+n-1), \quad n \geq 1.$$

y que son las soluciones (alrededor del cero) de la ecuación hipergeométrica de Gauss

$$x(1-x)y''(x) + (\gamma - (\alpha + \beta + 1)x)y'(x) - \alpha\beta y = 0.$$

Hay que destacar que MAXIMA no tiene definidas las funciones `gauss_a` y `gauss_b`, simplemente las reconoce. Si queremos trabajar con ellas tenemos que definir las correspondientes series de potencias o usar la orden `hypergeometric ([<a1>, ..., <ap>], [<b1>, ..., <bq>], x)` que define la función hipergeométrica

$${}_pF_q \left(\begin{matrix} a_1, a_2, \dots, a_p \\ b_1, b_2, \dots, b_q \end{matrix} \middle| x \right) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k \cdots (a_p)_k}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{x^k}{k!}, \quad (6.4.5)$$

así como el paquete `hypergeometric` para manipularlas.

Vamos a mostrar como trabajar con ellas. Comenzaremos definiendo una expresión para la ecuación diferencial

```
(%i16) ecuh(expr):= x*(1-x)*diff(expr,x,2)+(3/2-2*x)*diff(expr,x)+2*expr $
```

que usaremos luego para comprobar si son correctas las soluciones encontradas por MAXIMA. Definamos ahora las dos soluciones linealmente independientes de la ecuación (6.4.4). Comenzamos con la primera

```
(%i17) kill(a,b,c)$ a:-1$ b:2$ c:3/2$
      hypergeometric([a,b],[c],x);
(%o11) hypergeometric([-1,2],[3/2],x)
```

Como se ve la salida es la misma que la entrada. No obstante podemos usar la variable `expand_hypergeometric` a la que le asignaremos el valor `true` que desarrolla la función hipergeométrica en caso de que uno de los argumentos a_1, \dots, a_p es un entero negativo obteniéndose un polinomio.

```
(%i12) expand_hypergeometric : true$
(%i13) hypergeometric([a,b],[c],x);
(%o13) 1-(4*x)/3
(%i14) define(y1(x),%);
(%o14) y1(x):=1-(4*x)/3
(%i15) sol2:(x)^(1-c)*hypergeometric([a-c+1,b-c+1],[2-c],x);
(%o15) hypergeometric([-3/2,3/2],[1/2],x)/sqrt(x)
```

Para la segunda solución los parámetros a_1 y a_2 no son enteros negativos así que procedemos de otro modo. Para ello cargamos el paquete `hypergeometric` y usamos el comando `hypergeometric_simp` que intenta escribir la función hipergeométrica en términos de otras funciones de MAXIMA.

```
(%i16) load (hypergeometric)
$(%i17) hypergeometric_simp(sol2), radcan;
(%o17) -(sqrt(1-x)*(4*x-1))/sqrt(x)
(%i18) define(y2(x),%);
(%o18) y2(x):=-(sqrt(1-x)*(4*x-1))/sqrt(x)
```

Ahora comprobamos las funciones obtenidas satisfacen la ecuación (6.4.4)

```
(%i19) ecuh(y1(x)), ratsimp;
(%o19) 0
(%i20) ecuh(y2(x)), ratsimp;
(%o20) 0
```

y las dibujamos en el intervalo $(0, 1]$ (ver gráfica de la izquierda de la figura 6.7)

```
(%i21) wxplot2d([y1(x),y2(x)], [x,0.01,.95], [legend,"y1","y2"]);
(%t21) (Graphics)
```

Veamos ahora el problema de valores iniciales (PVI) para la ecuación (6.4.4) con las condiciones $y(1/2) = 0$, $y'(1/2) = 1$ (nótese que en el punto $x = 0$ la segunda solución tiende a infinito). Para ello definimos las derivadas de las soluciones

```
(%i22) define(dy1(x),diff(y1(x),x));
      define(dy2(x),radcan(diff(y2(x),x)));
(%o22) dy1(x):=-4/3
(%o23) dy2(x):=(8*x^2-4*x-1)/(2*sqrt(1-x)*x^(3/2))
```

y la solución general así como su derivada

```
(%i25) define(sol(x),alpha*y1(x)+beta*y2(x))$
      define(dsol(x),diff(alpha*y1(x)+beta*y2(x),x))$
```

Usando las funciones anteriores podemos resolver el PVI y definir la solución general

```
(%i26) eq1: sol(1/2)=0;
      eq2:dsol(1/2)=1;
(%o26) alpha/3-beta=0
(%o27) -2*beta-(4*alpha)/3=1
(%i28) solve([eq1,eq2],[alpha,beta]);
      ev(alpha*y1(x)+beta*y2(x),%[1])$
      define(soly(x),radcan(%));
(%o28) [[alpha=-1/2,beta=-1/6]]
(%o30) soly(x):=(sqrt(1-x)*(4*x-1)+sqrt(x)*(4*x-3))/(6*sqrt(x))
```

A continuación comprobamos que la solución cumple con todas las condiciones requeridas, i.e., las condiciones iniciales y la propia ecuación diferencial:

```
(%i33) soly(1/2);
      diff(soly(x),x)$ ev(% ,x=1/2);
(%o31) 0
(%o33) 1
(%i34) ecuh(soly(x)), ratsimp;
(%o34) 0
```

y la representamos

```
(%i35) wxplot2d(soly(x), [x,0.5,.95], [legend,"solución y(x)"]);
(%t35) (Graphics)
```

Vamos ahora a resolver el mismo PVI numéricamente usando el comando `rk` (ver página 130)

```
(%i36) sol: rk([u,-((3/2-2*x)/(x*(1-x)))*u-(2/(x*(1-x)))*y],  
           [y,u],[0.0,1.0],[x,0.5,.95,0.01])$  
xx:makelist(sol[k][1],k,1,length(sol))$  
yy:makelist(sol[k][2],k,1,length(sol))$  
wxplot2d([discrete,xx,yy], [style, [points,2,2]] );  
(%t41) (Graphics)
```

Finalmente dibujamos ambas soluciones, numérica y analítica

```
(%i42) wxplot2d([soly(x), [discrete,xx,yy]], [x,0.5,.95],  
              [legend,"solución y(x)","solución numérica"],  
              [y,0,.3],[style, [lines,2,1],[points,2,2]]);  
(%t42) (Graphics)
```

que podemos ver en la gráfica de la derecha de la figura 6.7.

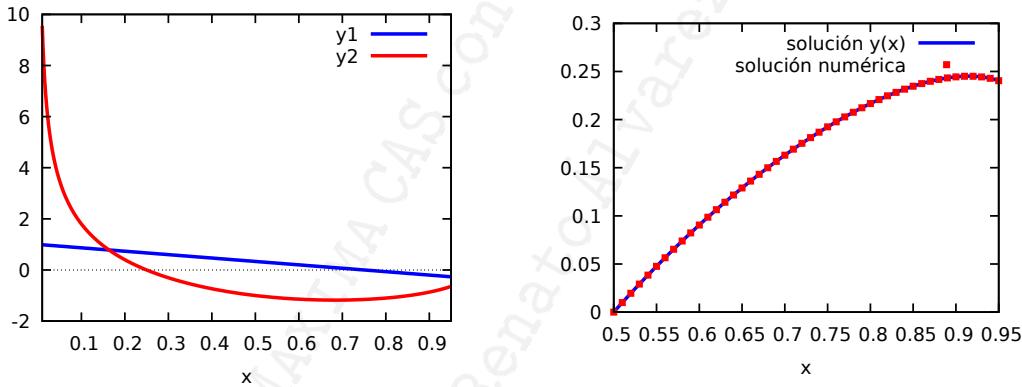


Figura 6.7: Las dos soluciones linealmente independientes de la EDO (6.4.4) (izquierda) y comparación de las soluciones numéricas y analíticas del PVI.

Veamos ahora la ecuación

$$x(1-x)y''(x) + \left(\frac{1}{2} - 2x\right)y'(x) + \frac{1}{2}y = 0. \quad (6.4.6)$$

```
(%i1) eq5:x*(1-x)*'diff(y,x,2)+(1/2-2*x)*'diff(y,x)+1/2*y=0;  
(%o1) (1-x)*x*('diff(y,x,2))+((1/2-2*x)*('diff(y,x,1))+y)/2=0  
(%i3) load(hypergeometric)$  
load('contrib_ode)$  
(%i4) sol5:odelin(eq5,y,x);  
(%o4) {gauss[a](sqrt(3)/2,-sqrt(3)/2,1/2,x)/sqrt(x-1),  
gauss[b](sqrt(3)/2,-sqrt(3)/2,1/2,x)/sqrt(x-1)}
```

Si nos interesa trabajar en el intervalo $[0, 1]$ está claro que debemos reescribir la $\sqrt{x-1}$ como $\sqrt{1-x}$. Así, para la primera solución tenemos

```
(%i15) kill(a,b,c)$ a:sqrt(3)/2$ b:-sqrt(3)/2$ c:1/2$
      assume(x>0)$
      hypergeometric([a,b],[c],x)/sqrt(1-x)$
      hypergeometric_simp(%), radcan$ 
      sol1:%;
(%o12) cos(sqrt(3)*asin(sqrt(x)))/sqrt(1-x)
```

y para la segunda

```
(%i13) (x)^(1-c)*hypergeometric([a-c+1,b-c+1],[2-c],x)/sqrt(1-x)$
      hypergeometric_simp(%), radcan$ 
      sol2:%;
(%o15) sin(sqrt(3)*asin(sqrt(x)))/(sqrt(3)*sqrt(1-x))
```

Como ejercicio comprobar que ambas son soluciones de la ecuación (6.4.6) y representarlas gráficamente.

Para terminar este apartado vamos a representar gráficamente los cinco primeros polinomios de Jacobi $P_n^{(1/5,0)}(x)$. Usaremos el comando draw

```
(%i30) load(draw)$
(%i31) draw2d( font="Arial",font_size=27,line_width=5,line_type=dots,
      color=grey,explicit(0,x,-1.0,1.0), parametric(0,t,t,-1,1),
      color=grey,explicit(0,x,-1.0,1.0),line_type=solid,line_width=8,
      color=green, explicit( jacobi_p (1, 1/5, 0, x),x,-1.0,1.0),
      color=dark_violet, explicit( jacobi_p (2,1/5,0,x),x,-1.0,1.0),
      color=navy, explicit( jacobi_p (3, 1/5, 0, x),x,-1.0,1.0),
      color=blue, explicit( jacobi_p (4, 1/5, 0, x),x,-1.0,1.0),
      color=red, explicit( jacobi_p (5, 1/5, 0, x),x,-1.0,1.0) )$
```

A continuación usaremos la orden concat para nombrar los distintos polinomios e incluirlos en la gráfica. Para ello escribimos la secuencia

```
(%i32) colors:[green,dark_violet,navy,blue,red];
(%i33) polijac:makelist([key = concat("Pol. Jacobi de grado ",k),
      color=colors[k], explicit( jacobi_p (k, 1/5, 0, x),
      x,-1.0,1.0)],k,1,5)$
```

que nos genera una lista cuyos elementos son

```
[key="titulo",color="el color", explicit(polinomio,x,-1,1)]
```

que luego usamos para representarlos (ver figura 6.8 (derecha)):

```
(%i34) draw2d(font="Arial", font_size = 27,
      line_width = 5, line_type = dots, color = grey,
      explicit( 0 ,x,-1.0,1.0), parametric(0,t,t,-1,1.5),
      color=grey, explicit(0,x,-1.0,1.0),
      line_type=solid, line_width=5, polijac);
```

Para exportar en pdf esta última gráfica agregamos las opciones

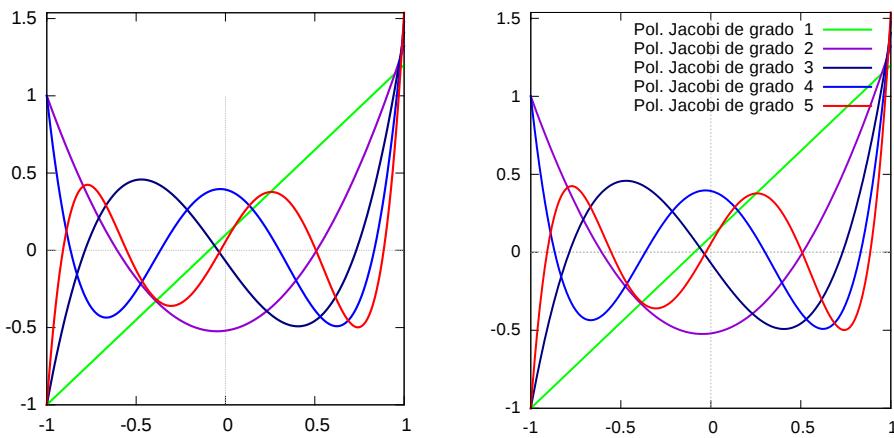


Figura 6.8: Los gráficos de los polinomios de Jacobi $P_n^{(1/5,0)}(x)$, $n = 1, 2, 3, 4, 5$.

```
file_name="/home/renato/poljac1", terminal=pdf,dimensions=[1000,1000]
```

En la figura 6.8 (izquierda) están representados los primeros cinco polinomios de Jacobi $P_n^{(1/5,0)}(x)$ incluyendo los dos ejes de coordenadas. En la figura de la derecha mostramos la salida de la última secuencia donde se muestra el código de colores usado para cada uno de los polinomios.

En el próximo apartado veremos como resolver las EDOs del tipo (6.4.3) usando series de potencias.

Introducción al MAXIMA CAS con algunas aplicaciones
Prof. Dr. Renato Álvarez Nodarse

Capítulo 7

Resolución de EDOs mediante series de potencias.

7.1. Un ejemplo ilustrativo.

Sea la EDO

$$y'' + p(x)y' + q(x)y = f(x). \quad (7.1.1)$$

El objetivo es resolver la EDO anterior cuando las funciones $p(x)$, $q(x)$ y $f(x)$ dependen de x . Para ello vamos a suponer que dichas funciones son “razonablemente” buenas en un entorno de cierto punto x_0 de la recta real y buscaremos la solución como una *serie de potencias*, es decir en la forma

$$y(x) = \sum_{k=0}^{\infty} a_k(x - x_0)^k, \quad a_k \in \mathbb{R}. \quad (7.1.2)$$

Las funciones que se pueden expresar mediante una serie de potencias convergentes en todo un entorno de $x = x_0$ como la anterior se denominan funciones analíticas en $x = x_0$. Por sencillez supondremos $x_0 = 0$, ya que en caso que $x_0 \neq 0$ siempre podemos realizar el cambio de variable $z = x - x_0$ de forma que en la nueva variable $z_0 = 0$.

La idea del método es sustituir la serie $y(x)$ (7.1.2) en (7.1.1) e igualar los coeficientes de las potencias. De esta forma obtenemos un sistema de ecuaciones para los coeficientes a_n en (7.1.2). Es importante que la serie converja en todo un entorno de x_0 , de esta forma tenemos asegurada la existencia de la solución al menos en cierta región.

Como ejemplo resolvamos la EDO $y'' - 2xy' - 2y = 0$. Para ello buscamos la solución $y(x) = \sum_{k=0}^{\infty} a_k x^k$, la sustituimos en la EDO y usamos que

$$\begin{aligned} y'(x) &= \frac{d}{dx} \sum_{k=0}^{\infty} a_k x^k = \sum_{k=0}^{\infty} (a_k x^k)' = \sum_{k=0}^{\infty} k a_k x^{k-1} = \sum_{n=0}^{\infty} (n+1) a_{n+1} x^n, \\ y''(x) &= \frac{d}{dx} y'(x) = \frac{d}{dx} \sum_{k=0}^{\infty} k a_k x^{k-1} = \sum_{k=0}^{\infty} k(k-1) a_k x^{k-2} = \sum_{n=0}^{\infty} (n+1)(n+2) a_{n+2} x^n \end{aligned}$$

lo que nos da

$$\sum_{k=0}^{\infty} k(k-1) a_k x^{k-1} - 2 \sum_{k=0}^{\infty} k a_k x^k - 2 \sum_{k=0}^{\infty} a_k x^k = 0,$$

que equivale a

$$\sum_{n=0}^{\infty} [(n+1)(n+2)a_{n+2} - (2n+2)a_n] x^n = 0.$$

Como $(x^n)_n$ es una sucesión de funciones linealmente independientes la igualdad a cero tiene lugar si y sólo si $(n+1)(n+2)a_{n+2} - (2n+2)a_n = 0$, de donde tenemos

$$a_{n+2} = \frac{2}{n+2}a_n, \quad n \geq 0. \quad (7.1.3)$$

La ecuación anterior define todos los valores de a_n en función de a_0 y a_1 . En efecto, si sabemos a_0 , la recurrencia anterior permite calcular los valores a_2, a_4, \dots, a_{2k} , $k \in \mathbb{N}$ y si conocemos a_1 entonces podemos calcular $a_3, a_5, \dots, a_{2k+1}$, $k \in \mathbb{N}$. Así, tenemos

$$a_{2n} = \frac{2}{2n} a_{2n-2} = \left(\frac{2}{2n}\right) \left(\frac{2}{2n-2}\right) a_{2n-4} = \dots = \frac{2^n}{(2n)(2n-2)\dots 2} a_0 = \frac{a_0}{n!},$$

$$a_{2n+1} = \frac{2}{2n+1} a_{2n-1} = \left(\frac{2}{2n+1}\right) \left(\frac{2}{2n-1}\right) a_{2n-3} = \dots = \frac{2^n}{(2n+1)(2n-1)\dots 3 \cdot 1} a_1,$$

es decir $2^n/(2n+1)!! a_1$, donde $(2n+1)!! = 1 \cdot 3 \cdot 5 \cdots (2n+1)$. De esta forma obtenemos dos soluciones linealmente independientes (una tiene solo potencias pares y la otra solo impares)

$$y(x) = a_0 \sum_{n=0}^{\infty} \frac{x^{2n}}{n!} + a_1 \sum_{n=0}^{\infty} \frac{2^n x^{2n+1}}{(2n+1)!!}.$$

La primera suma es fácilmente reconocible como la serie de potencias de la función e^{x^2} , no así la segunda que en general no se expresa como combinación de funciones elementales. De la expresión explícita de las dos sumas anteriores es fácil comprobar que el radio de convergencia es infinito.

Resolvámoslo ahora con MAXIMA. Para ello definimos la sucesión $bn[n]$ solución de (7.1.3) cuando $a_0 = 1$ y $a_1 = 0$.

```
(%i1) bn[0]:=1;bn[1]:=0;
bn[n]:=2/(n)*bn[n-2];
(%o1) 1
(%o2) 0
(%o3) bn[n]:=2/n*bn[n-2]
```

Luego definimos la solución aproximada hasta el orden que queramos:

```
(%i4) solaprox(x,p):=sum(bn[n]*x^n,n,0,p);
(%o4) solaprox(x,p):=sum(bn[n]*x^n,n,0,p)
(%i5) solaprox(x,10);
(%o5) x^10/120+x^8/24+x^6/6+x^4/2+x^2+1
```

Ahora repetimos el proceso para obtener la solución de (7.1.3) cuando $a_0 = 0$ y $a_1 = 1$

```
(%i6) bi[0]:=0;bi[1]:=1;
bi[n]:=2/(n)*bi[n-2];
(%o6) 0
(%o7) 1
(%o8) bi[n]:=2/n*bi[n-2]
```

que nos conduce a la segunda solución linealmente independiente:

```
(%i9) solaprox(x,p):=sum(bi[n]*x^n,n,0,p);
(%o9) solaprox(x,p):=sum(bi[n]*x^n,n,0,p)
(%i10) solaprox(x,10);
(%o10) (16*x^9)/945+(8*x^7)/105+(4*x^5)/15+(2*x^3)/3+x
```

Ahora podemos dibujar ambas soluciones

```
(%i11) wxplot2d([solaprox(x,10),solaprox(x,10)], [x,-2,2],
[legend,"y par","y impar"], [xlabel,"x"], [ylabel,"y"]);
(%t11) << Graphics >>
```

En el caso que nos ocupa el comando `ode2` es capaz de resolver la EDO. En particular resolveremos el PVI cuando $y(0) = 1$ e $y'(0) = 0$

```
(%o11)(%i12) ed:'diff(y,x,2)-2*x*'diff(y,x)-2*y=0;
(%o12) 'diff(y,x,2)-2*x*('diff(y,x,1))-2*y=0
(%i13) ode2(ed,y,x);
      ic2(%,x=0,y=1,diff(y,x)=0);
(%o13) y=(sqrt(%pi)*%k1*%e^x^2*erf(x))/2+%k2*%e^x^2
(%o14) y=%e^x^2
```

que nos conduce a la primera de las dos soluciones:

```
(%i15) define(soana(x),rhs(%));
(%o15) soana(x):=%e^x^2
```

Dibujamos las soluciones aproximada y exacta, respectivamente:

```
(%i16) wxplot2d([solaprox(x,10),soana(x)], [x,-2,2],
[legend,"y aproximada","y exacta"], [xlabel,"x"], [ylabel,"y"]);
(%t16) << Graphics >>
```

Finalmente, resolvemos el problema numéricamente con el comando `rk` que ya hemos usado varias veces

```
(%o16)(%i17) load(dynamics)$
(%i18) solnum: rk([z,2*x*z+2*y],[y,z],[1,0],[x,0,2,0.02])$ 
      soly:create_list([solnum[i][1],solnum[i][2]],i,1,length(solnum))$
```

y dibujamos las soluciones aproximada, exacta y numérica, respectivamente:

```
(%i20) wxplot2d([solaprox(x,10),soana(x),discrete,soly]),
      [x,0,2],[legend,"y aproximada","y exacta","y numerica"]);$ 
(%t20) << Graphics >>
```

El primer y tercer gráficos están representados en la figura 7.1.

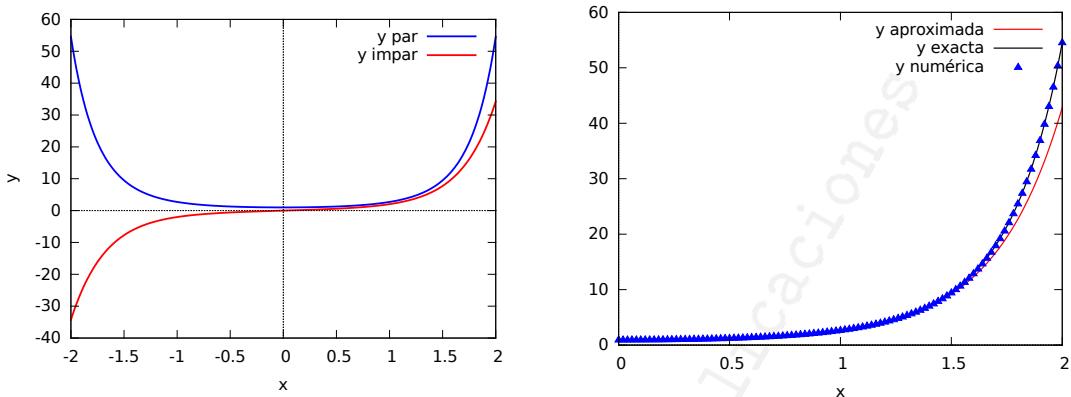


Figura 7.1: Las dos soluciones linealmente independientes de la EDO $y'' - 2xy' - 2y = 0$ (izquierda) y la comparación de las soluciones aproximada, exacta y numérica, respectivamente, del PVI cuando $y(0) = 1$ e $y'(0) = 0$.

7.2. Automatizando el método de las series de potencias.

Como colofón vamos a resolver la EDO inicial $y'' - 2xy' - 2y = 0$ mediante series “truncadas” hasta el orden que queramos usando MAXIMA. La idea es buscar la solución de la forma $y(x) = \sum_{k=0}^N a_k x^k$, con N prefijado.

Comenzaremos limpiando las variables a usar (aunque se recomienda reiniciar MAXIMA) y definiendo la serie truncada y el grado hasta el que queremos calcular la solución:

```
(%i1) kill(y,x,a,des,Aa,N,seriep)$  
seriep(x,p):=sum(a[n]*x^n,n,0,p)$  
N:12$
```

A continuación sustituimos nuestro polinomio `seriep` en la EDO y definimos la salida como una nueva función `des(x)`. Esta función es un polinomio de grado a lo más N . La idea es igualar coeficientes lo que nos dará un sistema lineal de $N+1$ ecuaciones con $N+1$ incógnitas. Para ello usamos el comando `coeff(expr, var, pot)` que nos devuelve el valor del coeficiente que tiene la variable `var` elevada a la potencia `pot`. Por ejemplo

```
coeff(b*x^4+a^2*x^2-2*sin(x)+3*sin(x)^3,x,4)
```

devolvería `b` mientras que

```
coeff(b*x^4+a^2*x^2-2*sin(x)+3*sin(x)^3,sin(x),3)
```

devolvería 3. Dado que fijaremos $a_0 = 1$ y $a_1 = 0$, tenemos que incluirlos en el sistema. Para ello hacemos dos listas con `makelist` y las juntamos con `append` (recordemos que para juntar dos listas dadas `l1` y `l2` se puede usar el comando `joint(l1,l2)`). Finalmente, construimos la lista de incógnitas `incog` y resolvemos el sistema

```
(%i4) define(des(x),expand(diff(seriep(x,N),x,2)-2*x*diff(seriep(x,N),x,1)  
-2*seriep(x,N)))$  
siseq1:append([a[0]-1=0,a[1]=0,des(0)=0],
```

```

makelist(coeff(des(x),x^k)=0 ,k,1,N-2))$  

incog:makelist(a[k],k,0,N)$  

coef1:solve(siseq1,incog);  

(%o7) [[a[0]=1,a[1]=0,a[2]=1,a[3]=0,a[4]=1/2,a[5]=0,  

a[6]=1/6,a[7]=0,a[8]=1/24,a[9]=0,a[10]=1/120,a[11]=0,a[12]=1/720]]

```

El siguiente paso consiste en construir el polinomio de forma que podamos trabajar con él. para ello construimos la lista de los coeficientes a partir de la salida de solve usamos el comando `apply ("+", lista)`¹ que suma los elementos de la lista lista

```

(%i8) listacoef1:makelist( rhs(coef1[1][k]),k,1,N+1)$  

listpot:makelist(x^k,k,0,N)$  

define(ser1(x), apply ("+", listacoef1*listpot) );  

(%o10) ser1(x):=x^12/720+x^10/120+x^8/24+x^6/6+x^4/2+x^2+1

```

Finalmente repetimos los cálculos tomando los valores $a_0 = 0$ y $a_1 = 1$

```

siseq2:append([a[0]=0,a[1]-1=0],  

makelist(coeff(des(x),x,k)=0 ,k,0,N-2))$  

incog:makelist(a[k],k,0,N)$  

coef2:solve(siseq2,incog);  

(%o23) [[a[0]=0,a[1]=1,a[2]=0,a[3]=2/3,a[4]=0,a[5]=4/15,a[6]=0,  

a[7]=8/105,a[8]=0,a[9]=16/945,a[10]=0,a[11]=32/10395,a[12]=0]]  

listacoef2:makelist( rhs(coef2[1][k]),k,1,N+1)$  

listpot:makelist(x^k,k,0,N)$  

define(ser2(x), apply ("+", listacoef2*listpot));  

(%o26) ser2(x):=(32*x^11)/10395+(16*x^9)/945+(8*x^7)/105+  

(4*x^5)/15+(2*x^3)/3+x

```

Como ejercicio, dibujar ambas funciones y compararlas con las obtenidas al principio del apartado 7.1.

¹Este comando ya lo comentamos en el apartado 4.5.

Introducción al MAXIMA CAS con algunas aplicaciones
Prof. Dr. Renato Álvarez Nodarse

Capítulo 8

Otros ejemplos y ejercicios para el lector.

El objetivo de esta sección es presentar algunos problemas sencillos que podemos resolver usando MAXIMA. En algunos casos se incluye la solución detallada mientras que otros se proponen como ejercicio.

8.1. Dos ejemplos simples del álgebra lineal.

Vamos a discutir un par de ejercicios sencillos relacionados con el álgebra lineal.

8.1.1. Ejemplo 1

El primero es, dada una matriz $N \times N$, $A = [a_{ij}]_{i,j=1}^N$ definir una función que calcule la p -norma definida por

$$n_p(A) = \sqrt[p]{\sum_{i=1}^N \sum_{j=1}^N |a_{ij}|^p}.$$

Definamos una matriz genérica 3×3 y definamos la p -norma

```
(%i1) MM:genmatrix(a, 3, 3);  
(%i2) n(M,p):=block([m],  
m:list_matrix_entries (M^p),  
sum(m[k],k,1,length(m)))$  
(%i3) n(MM,p);
```

Una pregunta interesante es saber la sucesión de las potencias de una matriz tiene algún límite o no. Una manera de saberlo es estudiando si la norma de las potencias de la matriz tiene límite o no. Definamos la función A^k , $k \in \mathbb{N}$, que calcula la potencia k -ésima de A (no confundir con la orden M^k)

```
(%i4) define(powerM(M,k), M^(k));  
(%o4) powerM(M,k):=M^k
```

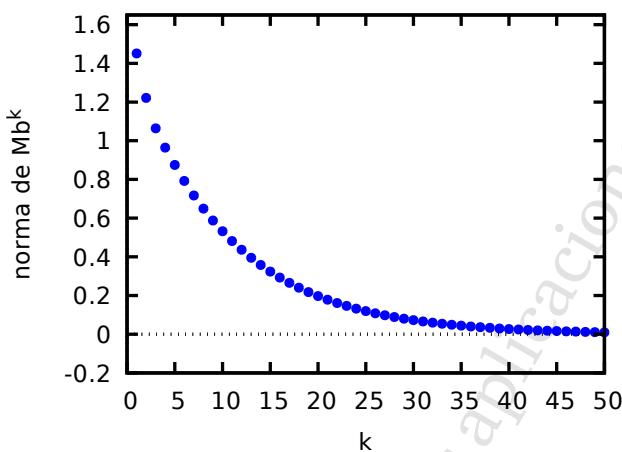


Figura 8.1: Comportamiento de la norma de la matriz Mb .

Como $n_p(A)$ es una norma¹ ello indica que si $n_p(A) \rightarrow 0$ entonces $A \rightarrow 0$. Veamos un ejemplo. Definimos una cierta matriz Mb y calculamos las normas de sus potencias para ver si la sucesión Mb^{n_k} tiene límite

```
(%i5) Mb: matrix( [0,0,1/3], [1/5,0,0], [0,7/10,9/10] );
(%i6) evo:makelist( n(powerM(Mb,k),2) ,k,1,50,1)$
wxplot2d([discrete,evo],[style,points],
          [xlabel,"k"],[ylabel,"norma de Mb^k"])$
(%t7) << Graphics >>
```

Como se en la figura 8.1 la norma de nuestra matriz se acerca a cero, lo que efectivamente indica que la propia matriz es cercana a matriz nula. Lo comprobamos calculando la norma de Mb^{200} y la potencia 200 de la matriz:

```
(%i8) float(n(powerM(Mb,200),2));
(%o8) 3.377091873620356*10^-9
(%i9) powerM(float(Mb),200);
(%o9) matrix(
[2.368893230664782*10^-6,1.127049380923565*10^-5,1.532047950973439*10^-5],
[4.979067672874238*10^-7,2.368893230664788*10^-6,3.220141088353051*10^-6],
[6.762296285541401*10^-6,3.217300697044229*10^-5,4.373418790694773*10^-5])
```

8.1.2. Ejemplo 2

Veamos ahora un problema de autovalores. Comenzaremos calculando los autovalores de la matriz Mb

```
(%i10) av:abs(float(eigenvalues(Mb)));
(%o10) [[0.2214571281540186,0.2214571281540186,0.9515408852827707],
           [1,1,1]]
```

¹Véase, por ejemplo, [7].

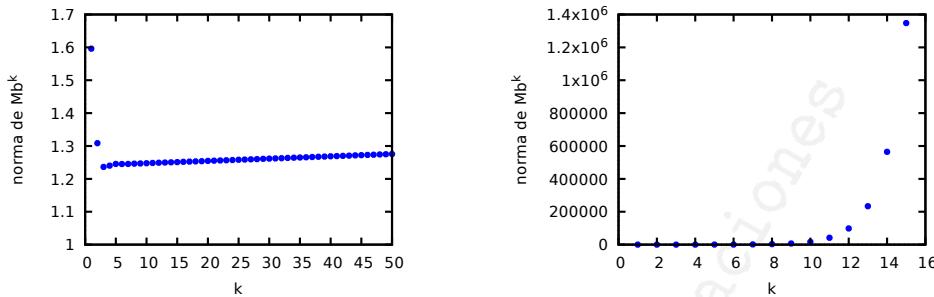


Figura 8.2: Comportamiento de la norma de las potencias de la matriz M_b cuando al menos un autovalor es mayor que 1 (izquierda) y cuando todos lo son (derecha).

Pasemos a nuestro segundo problema. Una pregunta que podemos hacernos es si variando los valores de la matriz podemos conseguir que los autovalores sean, en valor absoluto, mayores que 1. Esto se puede relacionar con la estabilidad de modelos asociados a la matriz M_b . Propongámonos resolver el siguiente problema: Mientras todos los autovalores de M_b sean, en valor absoluto, menores que 1, incrementar el valor $M_{b,1}$ de 0.1 en 0.1 hasta que al menos uno sea mayor. Para eso hay que hacer un bucle con `do` pero en vez de usar la orden `for` debemos usar la orden `while`.

```
(%o11) kill(Mb, av)$
Mb: matrix(
[0,0,1/3], [1/5,0,0], [0,7/10,9/10])$
av:abs(float(eigenvalues(Mb))); kk:0$ 
while av[1][1]<1.0 and av[1][2]<1.0 and av[1][3]<1.0
do (Mb[2,1]:Mb[2,1]+.01, kk:kk+1, av:abs(float(eigenvalues(Mb))))$ 
Mb; kk; abs(float(eigenvalues(Mb)));
(%o13) [[0.221457128154018,0.221457128154018,0.951540885282770],[1,1,1]]
(%o16) matrix([0,0,1/3],[0.4300000000000002,0,0],[0,7/10,9/10])
(%o17) 23
(%o18) [[0.316710410677950,0.316710410677950,1.00027764286021],[1,1,1]]
```

Como vemos en este ejemplo, en 23 pasos obtenemos una matriz que tiene al menos un autovalor mayor que uno. Aquí la orden `while` condiciones `do` orden lo que hace es ejecutar la orden `orden` mientras la salida del `while` sea `true`.

Hecho esto podemos dibujar la evolución de las normas de la sucesión M_1^{k+1} de potencias de la matriz resultante:

```
(%i21) M1:Mb$
evo:makelist( n(powerM(M1,k),2), k,1,50,1)$
wxplot2d([discrete,evo],[style,points])$
(%t23) << Graphics >>
```

que, como se ve (ver la gráfica de la izquierda en la figura 8.2), a partir de la tercera potencia es una sucesión creciente.

Cambiemos ligeramente el problema por el siguiente: hasta que todos los autovalores de M_b no sean, en valor absoluto, mayores que 1, incrementar el valor $M_{b,1}$ de 0.1 en 0.1.

Mientras al menos haya, en valor absoluto, un autovalor de M_b menor que uno, incrementar el valor $M_{b,1}$ en 0.01. Es decir incrementar el elemento $M_{b,1}$ hasta que todos los autovalores sean, en valor absoluto, mayores que uno.

```
(%i22) kill(Mb,av)$
      Mb: float(matrix( [0,0,1/3], [1/5,0,0], [0,7/10,9/10]) )$
      av:abs(float(eigenvalues(Mb))); kk:0$ 
      while av[1][1]<1.0 or av[1][2]<1.0 or av[1][3]<1.0 do
      ( Mb[2,1]:Mb[2,1]+.01, kk:kk+1,
      av:abs(float(eigenvalues(Mb))) )$
      Mb; av:abs(float(eigenvalues(Mb))); kk;
(%o24) [[0.221457128154018,0.221457128154018,0.951540885282770],[1,1,1]]
(%o27) matrix([0,0,0.33333333333333],[6.6299999999999,0,0],[0.0,0.7,0.9])
(%o28) [[1.00010345656338,1.00010345656338,1.5466799550598],[1.0,1.0,1.0]]
(%o29) 643
```

O sea, en 643 pasos se obtiene el resultado. Finalmente dibujamos la evolución de las normas potencias de la matriz resultante, que nos indica que al menos una entrada de la matriz se hacer muy grande (ver la gráfica de la derecha en la figura 8.2)

```
(%i30) M2:Mb$
      evo:makelist( n(powerM(M2,k),2),k,1,15,1)$
(%i32) wxplot2d([discrete,evo],[style,points],[ xlabel,"k"],
      [ ylabel,"norma de Mb^k"] )$
(%t32) << Graphics >>
(%i33) powerM(M2,100);
(%o33) matrix(
[1.983401672306752*10^18,4.626979803000077*10^17,1.022350987680987*10^18],
[8.502051794474104*10^18,1.983401672306753*10^18,4.382410870555725*10^18],
[9.203062828167024*10^18,2.146937074130075*10^18,4.743749339045423*10^18])
(%i34) n(powerM(M2,100),2);
(%o34) 2.124263271158474*10^38
```

Notemos que en el segundo caso hemos tenido que hacer muchas más iteraciones: 643 en concreto. Eso nos dice que podría ocurrir que nunca obtengamos todos los autovalores mayores que 1, por tanto es conveniente que el bucle terminara después de un número de pasos determinado. ¿Cómo se puede hacer esto? Esta pregunta la dejaremos como ejercicio al lector.

Como último ejercicio de este apartado realizar el mismo estudio cambiando el elemento $Mb_{3,1}$ en vez del elemento $Mb_{2,1}$.

8.2. Fórmulas de cuadratura para funciones explícitas.

Sea $f(x)$ una función continua definida en el intervalo $[a, b]$. El objetivo es encontrar fórmulas aproximadas para calcular la integral $\int_a^b f(x) dx$. En caso de conocer la primitiva $F(x)$ es evidente que podemos encontrar el valor exacto de la integral utilizando el Teorema fundamental del cálculo integral: $\int_a^b f(x) dx = F(b) - F(a)$. Sin embargo no siempre esto es posible. Por ejemplo, para la función $f(x) = e^{-x^2}$ no existe ninguna primitiva que podamos escribir utilizando funciones elementales. Existen una gran cantidad de métodos para resolver este problema.

8.2.1. Fórmula de los rectángulos.

Como ya vimos en el apartado 5.3, una aproximación para calcular la integral $\int_a^b f(x)dx$ consiste en aproximar el área bajo la curva $y = f(x)$ por un rectángulo de base $b - a$ y altura $f\left(\frac{a+b}{2}\right)$ (ver figura 8.3), entonces

$$\int_a^b f(x) dx = (b - a)f\left(\frac{a + b}{2}\right) + R(\xi), \quad \xi \in [a, b], \quad (8.2.1)$$

donde el error $R(\xi)$, si f tiene primera y segunda derivadas continuas en $[a, b]$, se expresa de la forma

$$R(\xi) = \frac{(b - a)^2}{24} f''(\xi), \quad \xi \in [a, b].$$

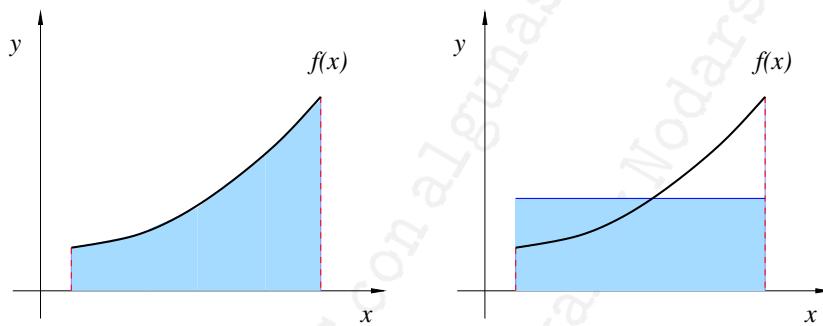


Figura 8.3: Aproximación de una integral por el método de los rectángulos. A la izquierda vemos el área bajo la curva que queremos calcular. A la derecha, la aproximación mediante el correspondiente rectángulo.

Si queremos aproximar la integral $\int_a^b f(x) dx$ con mejor exactitud podemos dividir el intervalo $[a, b]$ en n subintervalos, o sea, considerar la partición del intervalo

$$[a, b] = [a, x_1] \cup [x_1, x_2] \cup \dots \cup [x_{n-2}, x_{n-1}] \cup [x_{n-1}, b], \quad (8.2.2)$$

donde

$$x_k = a + \frac{b - a}{n}k, \quad n = 0, 1, 2, \dots, n, \quad x_0 = a, x_n = b. \quad (8.2.3)$$

Si ahora usamos la aditividad de la integral

$$\int_a^b f(x) dx = \int_a^{x_1} f(x) dx + \dots + \int_{x_k}^{x_{k+1}} f(x) dx + \dots + \int_{x_{n-1}}^b f(x) dx. \quad (8.2.4)$$

y aplicamos a cada sumando $\int_{x_k}^{x_{k+1}} f(x) dx$ la fórmula (8.2.1) obtenemos la ecuación

$$\int_a^b f(x) dx = \frac{b - a}{n} \sum_{k=0}^{n-1} f\left(\frac{x_k + x_{k+1}}{2}\right) + R(\xi), \quad (8.2.5)$$

donde

$$|R(\xi)| \leq M \frac{(b - a)^2}{24n^2}, \quad M = \max_{x \in [a, b]} |f''(x)|.$$

Veamos como implementar lo anterior con MAXIMA CAS. Definimos el intervalo $[a, b]$, el número de puntos en que vamos a dividir en intervalo y definimos la partición que usaremos:

```
(%i1) a:0; b:1; Nu:20;
      x[0]:=a; x[n]:=x[0]+n*(b-a)/Nu;
(%o1) 0
(%o2) 1
(%o3) 0
(%o4) 20
(%o5) x[n]:=x[0]+(n*(b-a))/Nu
```

Ya estamos en condiciones de definir la función a integrar e implementar la suma (8.2.5)

```
(%i6) define(f(x),x^2);
      rec:sum(f((x[k]+x[k+1])/2),k,0,Nu-1)*((b-a)/Nu);
      float(%);
(%o6) f(x):=x^2
(%o7) 533/1600
(%o8) 0.333125
```

En este caso, como la función tiene primitiva, podemos comparar el resultado numérico con el valor exacto

```
(%i9) exac:float(integrate(f(x),x,a,b));
      float(abs(rec-exac));
(%o9) 0.33333333333333
(%o10) 2.08333333333104*10^-4
```

8.2.2. Fórmula de los trapecios.

Otra aproximación de la integral $\int_a^b f(x) dx$ consiste en aproximar el área bajo la curva $y = f(x)$ no por un rectángulo sino por un trapecio de base $b - a$ (ver figura 8.4), entonces

$$\int_a^b f(x) dx = (b - a) \left(\frac{f(a) + f(b)}{2} \right) + R(\xi), \quad (8.2.6)$$

donde el error $R(\xi)$, si f tiene primera y segunda derivadas continuas en $[a, b]$ se expresa de la forma

$$R(\xi) = -\frac{(b - a)^2}{12} f''(\xi), \quad \xi \in [a, b].$$

Ahora podemos aproximar la integral $\int_a^b f(x) dx$ con mejor exactitud dividiendo, igual que antes, el intervalo $[a, b]$ en n puntos, o sea usando la partición (8.2.2) donde x_k son los valores (8.2.3). Nuevamente usamos la aditividad (8.2.4) y aplicamos a cada una de las integrales $\int_{x_k}^{x_{k+1}} f(x) dx$ la fórmula (8.2.1) que, en este caso, nos conduce a

$$\int_a^b f(x) dx = \frac{b - a}{2n} \left(f(a) + f(b) + 2 \sum_{k=1}^{n-1} f(x_k) \right) + R(\xi), \quad (8.2.7)$$

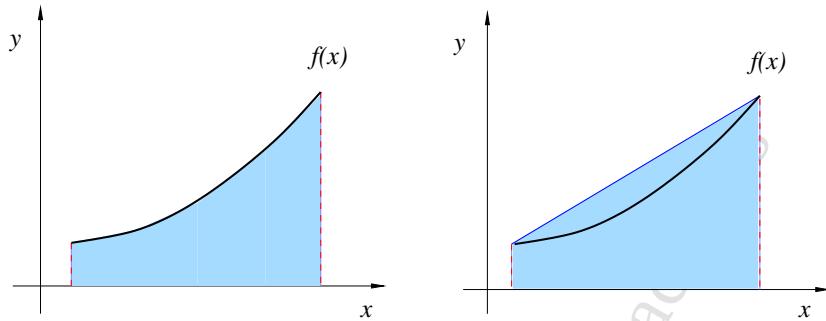


Figura 8.4: Aproximación de una integral por el método de los trapecios. A la izquierda vemos el área bajo la curva que queremos calcular. A la derecha, la aproximación mediante el correspondiente trapecio.

donde

$$|R(\xi)| \leq M \frac{(b-a)^2}{12n^2}, \quad M = \max_{x \in [a,b]} |f''(x)|.$$

Para implementar lo anterior con MAXIMA CAS volvemos a definir la partición:

```
(%i11) kill(all)$
      a:0;b:1;x[0]:=a;Nu:20; x[n]:=x[0]+n*(b-a)/Nu;
(%o1) 0
(%o2) 1
(%o3) 0
(%o4) 20
(%o5) x[n]:=x[0]+(n*(b-a))/Nu
```

A continuación definimos la función e implementamos la suma (8.2.7)

```
(%i6) define(f(x),x^2);
      tra: (f(a)+f(b)+ 2*sum(f(x[k]),k,1,Nu-1))*((b-a)/(2*Nu))$ 
      float(%);
(%o6) f(x):=x^2
(%o8) 0.33375
```

Finalmente, comparamos el resultado numérico con el valor exacto

```
(%i9) exac:float(integrate(f(x),x,a,b));
      float(abs(tra-exac));
(%o9) 0.33333333333333
(%o10) 4.16666666666763*10^-4
```

8.2.3. Método de Simpson.

El método de Simpson consiste en aproximar la integral $\int_a^b f(x)dx$ de la siguiente forma:

$$\int_a^b f(x) dx = A f(a) + B f\left(\frac{a+b}{2}\right) + C f(b) + R(\xi), \quad (8.2.8)$$

donde A, B, C son tales que $R(\xi)$ es igual a cero si $f(x) = 1$, $f(x) = x$ y $f(x) = x^2$, respectivamente. Es decir, si sustituimos en (8.2.8) la función f por cualquiera de las funciones $f(x) = 1$, $f(x) = x$ o $f(x) = x^2$, la fórmula es exacta, o sea $R(\xi) = 0$. Realizando dicha sustitución obtenemos un sistema de ecuaciones para las incógnitas A, B, C lo que nos conduce a la expresión

$$\int_a^b f(x) dx = \frac{b-a}{6} f(a) + \frac{4(b-a)}{6} f\left(\frac{a+b}{2}\right) + \frac{b-a}{6} f(b) + R(\xi). \quad (8.2.9)$$

Este método es equivalente a aproximar el área debajo de f por una parábola (ver figura 8.5).

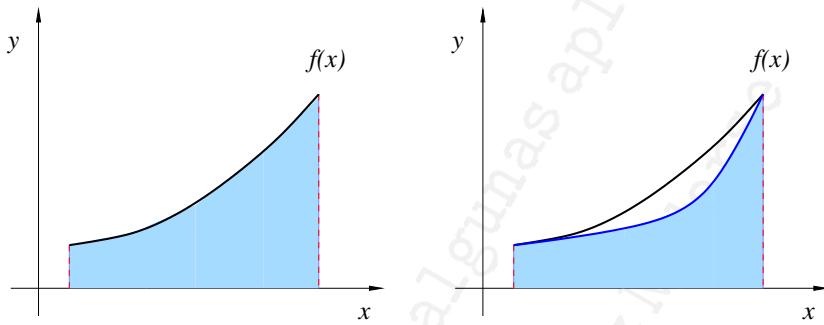


Figura 8.5: Aproximación de una integral por el método de Simpson. A la izquierda vemos el área bajo la curva que queremos calcular. A la derecha, la aproximación usando el método de los trapecios que equivale a encontrar el área bajo cierta parábola.

Al igual que en los casos anteriores vamos aproximar la integral $\int_a^b f(x) dx$ con mejor exactitud dividiendo el intervalo $[a, b]$, en este caso, en $2n$ subintervalos de la forma

$$[a, b] = [a, x_1] \cup [x_1, x_2] \cup \dots \cup [x_{2n-2}, x_{2n-1}] \cup [x_{2n-1}, b],$$

donde

$$x_k = a + \frac{b-a}{2n} k, \quad k = 0, 1, 2, \dots, 2n, \quad x_0 = a, x_{2n} = b.$$

Aplicaremos ahora la fórmula de Simpson (8.2.9) en cada subintervalo $[x_{2k}, x_{2k+2}]$, $k = 0, 1, \dots, n-1$, o sea, escribamos la integral original como la suma de las integrales

$$\int_a^b f(x) dx = \int_a^{x_2} f(x) dx + \dots + \int_{x_{2k}}^{x_{2k+2}} f(x) dx + \dots + \int_{x_{2n-2}}^b f(x) dx.$$

y apliquemos el método de Simpson a cada uno de los sumandos. Nótese que los intervalos siguen teniendo una longitud $x_{2k+2} - x_{2k} = \frac{b-a}{n}$ igual que antes. Esto nos conduce a la expresión

$$\int_a^b f(x) dx = \frac{b-a}{6n} \left(f(a) + f(b) + 4 \sum_{k=1}^n f(x_{2k-1}) + 2 \sum_{k=1}^{n-1} f(x_{2k}) \right) + R(\xi), \quad (8.2.10)$$

donde

$$|R(\xi)| \leq M \frac{(b-a)^5}{2880n^4}, \quad M = \max_{x \in [a,b]} |f^{(4)}(x)|.$$

Implementemos este método con MAXIMA. Primero definimos los puntos de la partición

```
(%i11) kill(all)$
(%i1) a:0;b:1;x[0]:a;Nu:10;
      x[n]:=x[0]+n*(b-a)/(2*Nu);
(%o1) 0
(%o2) 1
(%o3) 0
(%o4) 10
(%o5) x[n]:=x[0]+(n*(b-a))/(2*Nu)
```

para luego definir la función y la suma numérica (8.2.10)

```
(%i6) define(f(x),x^2);
      simp: (f(a)+f(b)+ 4*sum(f(x[2*k-1]),k,1,Nu) +
      2*sum(f(x[2*k]),k,1,Nu-1))*((b-a)/(6*Nu))$ 
      float(%);
(%o6) f(x):=x^2
(%o8) 0.3333333333333333
```

Finalmente, comparamos el resultado numérico con el valor exacto

```
(%i9) exac:float(integrate(f(x),x,a,b));
      float(abs(simp-exac));
(%o9) 0.333333333333333
(%o10) 0.0
```

que da cero tal y como predice la teoría.

8.2.4. Ejercicios: Comparación de los métodos de cuadratura.

Problema 8.2.4.1 Sea la función $f(x) = \cos x$. Calcular la integral

$$I = \int_0^{\frac{1}{2}} \cos x dx,$$

utilizando las fórmulas (8.2.1), (8.2.6), (8.2.9), respectivamente. Comparar los resultados con el resultado exacto

$$\int_0^{\frac{1}{2}} \cos x dx = \sin \frac{1}{2} = 0,4794255386\dots$$

Calcular una aproximación de la integral cambiando la función $f(x)$ por su polinomio de McLaurin de orden 5. Comparar los resultados con los del apartado anterior.

Problema 8.2.4.2 Calcular el orden del error cometido al calcular la integral

$$I = \int_0^1 f(x) dx \quad f(x) = \begin{cases} \frac{\sin x}{x}, & x \neq 0 \\ 1, & x = 0 \end{cases}$$

por los métodos de los rectángulos, los trapecios y de Simpson, respectivamente, utilizando en todos ellos una partición del intervalo $[0, 1]$ con $n = 4$ puntos. ¿Quién approxima mejor?

Problema 8.2.4.3 Vamos a considerar ahora el caso de la función $f(x) = x^{(1/2)}(1 - x)^{(3/2)}$. Para dicha función calcula la integral $\int_0^1 x^{(1/2)}(1 - x)^{(3/2)}dx$ usando el paquete numérico quad_qag, el método de Simpson y compáralo con el valor exacto de la misma usando la orden integrate.

Problema 8.2.4.4 Calcular la integral

$$I = \int_0^1 e^{-x^2} dx,$$

utilizando los métodos de los rectángulos, los trapecios y de Simpson cuando $n = 4$. Comparar los resultados con el resultado exacto con 10 cifras decimales $I = 0,7468241328\dots$ Utiliza la serie de McLaurin para calcular un valor aproximado y compáralo con los anteriores.

8.3. Fórmulas de cuadratura para una función definida por una lista.

Vamos ahora a considerar un problema distinto a los anteriores. Tanto el paquete quad_qag como los ejemplos de antes están concebidos para una función dada $f(x)$ definida explícitamente sobre cierto intervalo $[a, b]$ que se ha dividido en intervalos equidistantes. Está claro que esto no siempre es así, pues en determinados problemas podemos tener en vez de una función, una lista de sus valores numéricos evaluados en ciertos puntos del intervalo dado. La idea es implementar los tres métodos del apartado anterior cuando, en vez de tener una función explícita, tenemos dos listas de números: la lista de las abscisas y la de las correspondientes ordenadas. Antes de comenzar conviene hacer notar que, tanto en el caso de la fórmula de los rectángulos (8.2.1) como en la de Simpson (8.2.9), se toman valores de la función en puntos intermedios, es decir para aproximar la integral en el intervalo $[a, b]$ se necesitan tres puntos. Así que si queremos usar las expresiones (8.2.1) y (8.2.9) conviene que la lista tenga número $2n + 1$ impar de valores de forma que podemos dividirla en n intervalos que contengan tres puntos cada uno, tal y como hicimos para obtener la expresión (8.2.10).

Mostremos como proceder tomando como función test a $f(x) = 3x^2 + 2x + 1$ en el intervalo $[0, 1]$ cuya integral exacta es 3.

```
(%o11) kill(all)$
(%i1) f(x):=3*x^2+2*x+1;
      integrate(f(x),x,0,1);
(%o1) f(x):=3*x^2+2*x+1
(%o2) 3
```

Vamos ahora a crear una lista de los valores de las x y su correspondiente lista de sus imágenes. Para ello tomaremos el intervalo $[0, 1]$ y lo dividiremos en 100 subintervalos (luego tendremos 101 nodos) pues necesitamos aplicar tanto la fórmula de los rectángulos como la de Simpson a tres puntos consecutivos x_{n-1}, x_n, x_{n+1} :

```
(%i3) a:0$ b:1$ Nu:100$
      h:float((b-a)/(Nu))$
      x[0]:a$ x[n]:=x[0]+n*h;
```

```

lx:makelist( x[k],k,0,Nu)$ length(lx);
lf:makelist(float(f(x[k])),k,0,Nu)$ nn:length(lf);
(%o8) x[n]:=x[0]+n*h
(%o10) 101
(%o12) 101

```

A continuación aplicamos la fórmula de (8.2.5). Para ello hay que distinguir dos casos. El primero corresponde al caso cuando la función está dada explícitamente y que corresponde exactamente a la fórmula (8.2.5). Este caso es el calculado en la variable `rec`. Para el segundo caso conviene reescribir (8.2.5) de la siguiente forma

$$\int_a^b f(x) dx = \sum_{k=0}^{n-1} f\left(\frac{x_k + x_{k+1}}{2}\right) (x_{k+1} - x_k) + R(\xi), \quad (8.3.1)$$

lo que nos conduce a definir la variable `recl` que nos calcula el valor aproximado de la integral usando las listas de los valores de las x y $f(x)$. Aquí conviene recordar que cuando se trabaja con listas los elementos de las mismas se ordenan comenzando en la posición uno². Así tenemos

```

(%i13) rec:sum(f(x[2*k-1]),k,1,(nn-1)/2)*(2*h);
        recl:sum(lf[2*k]*(lx[2*k+1]-lx[2*k-1]),k,1,nn/2);
        recl-rec;
(%o13) 2.999899999999999
(%o14) 2.9999
(%o15) 8.881784197001253*10^-16

```

Como vemos la diferencia es del orden de precisión de MAXIMA (en este caso 10^{-16}) y la razón está en el uso de `float` para aproximar numéricamente los valores de la función f y de las x (recordemos que MAXIMA hace los cálculos simbólicos por defecto). Si eliminamos los dos comandos `float` el resultado es la fracción $29999/10000$.

Pasemos ahora al caso de la fórmula de los trapecios. Aquí podemos tomar cualquier partición pero por coherencia y para compararlo con los otros dos tomaremos la mismas listas que antes y modificaremos la expresión (8.2.7) de forma análoga a (8.3.1):

```

(%i16) tra:sum((f(x[2*k])+f(x[2*k+2]))/2, k, 0, (nn-2)/2)*(2*h);
        tral:sum((lf[2*k-1]+lf[2*k+1])/2*(lx[2*k+1]-lx[2*k-1]),k,1,nn/2);
        tral-tra;
(%o16) 3.0002
(%o17) 3.0002
(%o18) 0.0

```

Finalmente, implementamos en método de Simpson

```

(%i21) simpl:sum((1/6)*(lx[2*k+1]-lx[2*k-1])*(lf[2*k-1]+4*lf[2*k] +
        lf[2*k+1]), k,1,(nn)/2);
        simp:float((f(lx[1])+f(lx[nn])+2*sum(f(lx[2*k+1]),k,1,(nn-2)/2)
        +4*sum(f(lx[2*k]),k,1,(nn)/2))*(h/3));

```

²Por ejemplo si hacemos `lx[1]`; obtenemos el valor 0, pero si escribimos `lx[0]`; obtenemos el mensaje de

```

simpl-simp;
(%o19) 2.999999999999999
(%o20) 2.999999999999999
(%o21) 0.0
(%i22) simpl-3;
(%o22) -4.440892098500626*10^-16

```

Nótese que debido a que hemos usado float el resultado que obtenemos no es 3, pero la diferencia es del orden de precisión con que estamos trabajando. Si usamos valores exactos el resultado es 3 como cabría esperar:

```

(%i22) kill(all)$
(%i1) a:0$ b:1$ Nu:100$
      h:((b-a)/(Nu))$
      x[0]:=a; x[n]:=x[0]+n*h; f(x):=3*x^2+2*x+1;
      lx:makelist( x[k] ,k,0,Nu)$ length(lx);
      lf:makelist((f(x[k])),k,0,Nu)$ nn:length(lf);
      simpl:sum((1/6)*(lx[2*k+1]-lx[2*k-1])*(lf[2*k-1]+4*lf[2*k]
      +lf[2*k+1]), k,1,(nn)/2);
(%o5) 0
(%o6) x[n]:=x[0]+n*h
(%o7) f(x):=3*x^2+2*x+1
(%o9) 101
(%o11) 101
(%o12) 3

```

Como ejercicio proponemos al lector que construya una tabla de valores para una función algo más complicada, por ejemplo para $f(x) = 2e^{-x/5} \cos(2x + \pi/3)$ y repita los operaciones antes descritas.

8.4. Calculando los extremos de una función definida por una lista.

8.4.1. Definiendo el problema.

Como comentamos en el apartado anterior en muchos problemas prácticos en vez de una función tenemos una lista de los puntos $(x_n, f(x_n))$ de dicha función. En este apartado consideraremos el problema de encontrar todos los extremos relativos de una lista dada de puntos. Comencemos con un ejemplo muy sencillo. Sea la lista de enteros del 0 al 5:

```

(%i13) kill(all)$
(%i1) lis:[0,1,2,3,4,5,4,3,2,1,0,1,2,3,4,5,4,3,2,1,0,
      1,2,3,4,5,4,3,2,1,0,1,2,3,4,5,4,3,2,1,0]$ 
      length(lis);
      wxplot2d([discrete,lis])$ 
(%o2) 41

```

error: apply: no such "list.element: [0]" aunque el valor de x[0] está bien definido.

La idea es encontrar los máximos y mínimos locales de la lista. El máximo y mínimo global se pueden encontrar con las órdenes `smax` y `smin` del paquete `descriptive`

```
(%i4) load(descriptive)$  
(%i5) smax(lis); smin(lis);  
(%o5) 5  
(%o6) 0
```

Aquí nos interesan dos cosas: los extremos y su posición. La idea es buscar a lo largo de la lista el cambio de tendencia (creciente a decreciente y viceversa) asumiendo que si se repite un valor varias veces, tomaremos como extremo el último de ellos. Como reto dejamos al lector que modifique la secuencia de forma que nos informe si hay puntos consecutivos con el mismo valor.

```
(%i7) comp:length(lis)-2;  
     11:makelist( lis[i]>lis[i+1] or lis[i+1]<=lis[i+2] ,i,1,comp)$  
     12:makelist( lis[i]<lis[i+1] and lis[i+1]>=lis[i+2] ,i,1,comp)$  
(%i10) transpose(matrix(l1,l2));
```

La salida (%i10) es la matriz

true	false
false	true
true	false
false	true
true	false
⋮	⋮
true	false
true	false

Nótese que la secuencia

$$\text{lis}[i]>\text{lis}[i+1] \text{ or } \text{lis}[i+1]\leq\text{lis}[i+2]$$

tiene como salida `true` si una de las dos comparaciones es cierta y `false` únicamente si ambas son falsas. Por el contrario

$$\text{lis}[i]<\text{lis}[i+1] \text{ and } \text{lis}[i+1]\geq\text{lis}[i+2]$$

tiene como salida `true` únicamente cuando ambas son ciertas y `false` en cualquier otro caso. En general para los operadores lógicos `and`, `not`, `or` se tienen los resultados estándar

A	B	A and B	A or B	not A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Las listas 11 y 12 que hemos construido son complementarias y todos sus elementos son `true` y `false`, respectivamente, excepto en las posiciones donde está el máximo local. Lo anterior nos permite usar el comando `sublist_indices` que ya describimos en la página 16 combinado con la función indeterminada `lambda` para encontrar la posición en la lista de los máximos locales:

```
(%i11) sublist_indices (l1, lambda ([x], x='false));
(%o11) [5,15,25,35]
(%i12) sublist_indices (l2, lambda ([x], x='true));
(%o12) [5,15,25,35]
```

Para encontrar la posición en nuestra la lista original `lis` sólo tenemos que adicionar 1 a cada uno de los valores de cualquiera de las salidas anteriores:

```
(%i13) indi:sublist_indices (l2, lambda ([x], x='true))+1;
(%o13) [6,16,26,36]
```

y encontramos los valores que toman dichos máximos (en nuestro ejemplo evidentemente son los valores iguales a 5)

```
(%i14) makelist(lis[k],k,indi);
(%o14) [5,5,5,5]
```

Dado que este procedimiento lo usaremos varias veces a lo largo de estas notas vamos a definir una función que lo haga todo de una vez, lo cual haremos usando la orden `block` que ya vimos en la página 57:

```
(%i15) lismax(li):=block( [comp,mm], comp:length(li)-2,
                           mm:1+sublist_indices(
                           makelist(li[i]>li[i+1] or li[i+1]<=li[i+2],i,1,comp),
                           lambda ([x], x='false)))$
```

```
(%i16) lismax(lis);
(%o16) [6,16,26,36]
```

o, equivalentemente,

```
(%i17) lismaxeq(li):=block( [comp,mm], comp:length(li)-2,
                           mm:1+sublist_indices(
                           makelist(li[i]<li[i+1] and li[i+1]>=li[i+2],i,1,comp),
                           lambda ([x], x='true)))$
```

```
(%i18) lismaxeq(lis);
(%o18) [6,16,26,36]
```

De forma análoga podemos construir la función que encuentre los mínimos locales de una lista. Dejamos como ejercicio al lector que adapte las secuencias anteriores para el caso de los mínimos. La función en este caso es:

```
(%i19) lismin(li):=block( [comp,mm], comp:length(li)-2,
                           mm:1+sublist_indices(
                           makelist(li[i]<li[i+1] or li[i+1]>=li[i+2],i,1,comp),
                           lambda ([x], x='false)))$
```

cuya salida efectivamente nos da la posición de los mínimos locales con la cual podemos encontrar los valores de los mismos.

```
(%i20) indimin:lismin(lis);
(%o20) [11,21,31]
(%i21) makelist(lis[k],k,indimin);
(%o21) [0,0,0]
```

Como vamos usar las funciones `lismax` y `lismin` definidas anteriormente varias veces es conveniente crear nuestro propio paquete que las contenga y al que podemos agregar luego cuantas funciones necesitemos. Hay dos formas muy sencillas. La primera es simplemente crear un fichero `.mac` que contengan las dos definiciones y lo guardamos donde MAXIMA lo pueda encontrar (podemos usar la orden `file_search_maxima` que ya comentamos en la página 13). La otra forma es escribir en una nueva sesión de wxMAXIMA la secuencia

```
lismax(li):=block( [comp,mm], comp:length(li)-2,
mm:1+sublist_indices(
makelist( li[i]>li[i+1] or li[i+1]<=li[i+2] ,i,1,comp),
lambda ([x], x='false)))$  
lismin(li):=block( [comp,mm], comp:length(li)-2,
mm:1+sublist_indices(
makelist( li[i]<li[i+1] or li[i+1]>=li[i+2] ,i,1,comp),
lambda ([x], x='false)))$  
stringout ("proglst.mac", functions);
```

donde la orden `stringout` con la opción `functions` graba en el fichero `proglst.mac` todas las funciones que hayamos definido en nuestra sesión. Cuando tengamos listo el fichero `proglst.mac` podemos cargarlo con `load`.

8.4.2. Ejemplo de aplicación.

Vamos a usar nuestras dos funciones para el cálculo numérico de los extremos de una función de una variable. Como ejemplo tomaremos la función $f : [0, 2\pi] \mapsto \mathbb{R}$, $f(x) = \exp(-x/4) \sin(x) + \cos(3x)$ y crearemos las listas de las x y $f(x)$ correspondientes:

```
(%o22) kill(all)$
(%i1) load(draw)$
f(x):=exp(-x/4)*sin(x)+cos(3*x);
a:0$ b:6*pi$ Nu:1000; h:float((b-a)/(Nu))$
x[0]:=a; x[n]:=x[0]+n*h;
(%o2) f(x):=exp((-x)/4)*sin(x)+cos(3*x)
(%o8) x[n]:=x[0]+n*h
```

A continuación creamos la lista de las x con `makelist` y el de sus imágenes según $f(x)$ para lo cual usamos el comando `map` que ya describimos en la página 53

```
(%i9) xx:makelist(x[k],k,0,Nu)$
      l1:map(f,xx)$
      wxdraw2d(point_type=filled_circle, points_joined = true,
                point_size = .5, points(xx,l1))$
(%t11) (Graphics)
```

A continuación cargamos nuestro paquete que contiene las funciones `lismax` y `lismin` usando la orden `load`:

```
(%i12) load("proglist.mac");
(%o12) "proglist.mac"
```

y buscamos los valores donde se alcanzan los máximos así como los valores donde se alcanzan y cuando valen dichos valores:

```
(%i13) lma:lismax(l1);
(%o13) [7,110,223,336,445,556,668,779,890]
(%i14) xlma:makelist(xx[k],k,lma);
(%o14) [0.1130973355292325, 2.054601595447725, ..., 16.75725521424795]
(%i15) ylma:makelist(l1[k],k,lma);
(%o15) [1.052700670327255, 1.522523160830247, ..., 0.9868383461695104]
(%i16) pmax:makelist([xx[k],l1[k]],k,lma);
(%o16) [[0.1130973355292325,1.052700670327255],..., [16.75725521424795,0.9868383461695104]]
```

La lista `pmax` nos permite dibujar los extremos:

```
(%i17) wxdraw2d(point_type=filled_circle, points_joined = true,
                  point_size=0.5, points(xx,l1),
                  point_type=asterisk, points_joined = false, color=red,
                  point_size=3, points( pmax ) )$
(%t17) (Graphics)
```

La gráfica la podemos ver en la figura 8.6

Vamos a comprobar gráficamente que hemos calculado correctamente los máximos. Para ello crearemos listas de tres elementos alrededor de cada máximo que luego dibujaremos. Si no ha habido errores el punto central ha de ser el extremo (máximo en este caso). En este ejemplo tomamos el el primer máximo:

```
(%i18) for j:1 thru length(lma) do block(
      listax[j]:makelist(xx[k],k,lma[j]-1,lma[j]+1),
      listaf[j]:makelist(l1[k],k,lma[j]-1,lma[j]+1));
(%o18) done
(%i19) listax[1];listaf[1];
(%o19) [0.09424777960769379,0.1130973355292325,0.1319468914507713]
(%o20) [1.052210542969385,1.052700670327255,1.049968020465215]
(%i21) wxdraw2d(point_type=filled_circle, points_joined = true,
                  point_size = 3, points( listax[1],listaf[1]))$
(%t21) (Graphics)
```

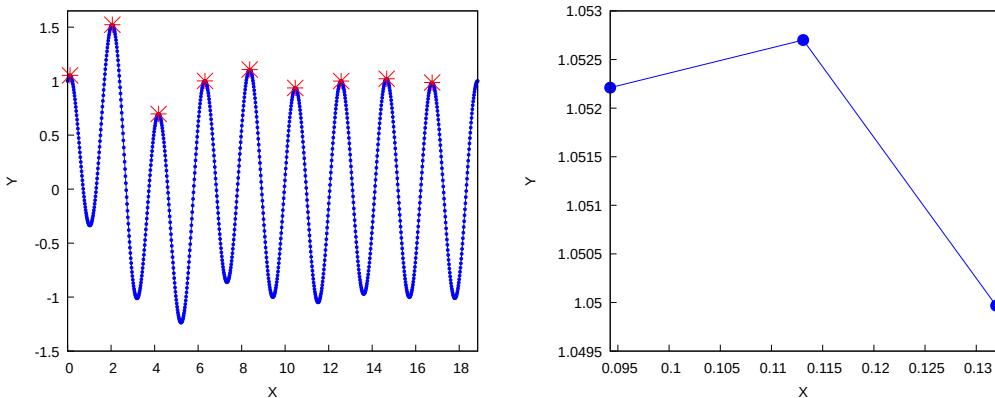


Figura 8.6: Los máximos de la función $f(x) = \exp(-x/4) \sin(x) + \cos(3x)$ calculados numéricamente (izquierda) y detalle del primer máximo (derecha).

Podemos también mostrar una tabla con las salidas alrededor de cada extremo

```
(%i22) transpose(matrix(makelist(
    makelist(l1[k], k, lma[i]-1, lma[i]+1), i, 1, length(lma))))$;
(%o22) matrix([[1.052210542969385, 1.052700670327255, 1.049968020465215]],
    [[1.521882481561259, 1.522523160830247, 1.519862713256897]],
    .
    .
    .
    [[1.021469901618179, 1.022013705837951, 1.019355791634181]],
    [[0.9856783052939371, 0.9868383461695104, 0.9848072691935124]])
```

O realizar una animación (ver el final del apartado 3, página 57)

```
(%i24) retraso:200000$ /* Depende de la potencia del ordenador */
for k:1 thru length(lma) step 1 do
  block([tiempo:1], while tiempo < retraso do tiempo: tiempo+1,
    draw2d(point_type=filled_circle, points_joined = true,
    point_size = 3, points( listax[k], listaf[k])))$
```

Como ejercicio proponemos al lector que adapte el ejemplo anterior para encontrar los mínimos de la función y luego hacer las correspondientes comprobaciones. Para ello se recomienda usar la función lismin definida en la página 165.

8.4.3. Ejemplo de extremos e integración numérica.

Vamos a considerar ahora el siguiente problema. Supongamos que tenemos una lista de datos que representa cierta medida experimental (por ejemplo la velocidad de cierta partícula) y queremos encontrar: sus extremos locales y la variación media de dicha medida.

Para generar la correspondiente lista usaremos una función relativamente sencilla que nos permita obtener los resultados analíticos “exactos” de forma que podamos comprobar el algoritmo. Como función test tomaremos la función $f(x) = x/100 + \sin(x/2)$ definida en el intervalo $[0, 100]$. Lo primero que haremos es generar una lista de valores de f :

```
(%i25) kill(all)$
(%i2) load(draw)$ load("proglist.mac");
(%o2) "proglist.mac"
(%i3) define(f(x),x/100+sin(1/2*x));
(%o3) f(x):=x/100+sin(x/2)
(%i4) a:0$ b:100$ Nu:1000$ h:(b-a)/(Nu)$
      x[0]:a$ x[n]:=x[0]+n*h;
(%o9) x[n]:=x[0]+n*h
(%i12) xx:makelist(x[n],n,0,Nu)$
      ff:makelist(f(n),n,xx)$
      draw2d(point_type=filled_circle, points_joined = true,
              point_size = 1, points(xx,ff), yrange=[-1,2],
              xaxis =true, xaxis_type=solid, grid = true)$
(%t12) (Graphics)
```

La gráfica la vemos en la figura 8.7 (izquierda). A continuación usamos las funciones lismax y lismin, que definimos antes y que se encuentran en el fichero proglist.mac (ver apartado 8.4) que previamente hemos cargado, y que nos permiten generar la lista de los máximos y mínimos locales, respectivamente, así como donde se alcanzan:

```
(%i16) indimax:lismax(ff);
      listamax:makelist(xx[k],k,%)$ float(%);
      listama:f(listamax)$
(%o13) [33,158,284,410,535,661,787,912]
(%o15) [3.2,15.7,28.3,40.9,53.4,66.0,78.6,91.1]

(%i20) indimin:lismin(ff);
      listamix:makelist(xx[k],k,%)$ float(%);
      listami:f(%)$
(%o17) [95,221,346,472,598,723,849,974]
(%o19) [9.4,22.0,34.5,47.1,59.7,72.2,84.8,97.3]
```

y los dibujamos para comprobar que el cálculo ha sido correcto (ver la gráfica derecha de la figura 8.7):

```
(%i21) wxdraw2d(point_type=filled_circle, points_joined = true,
                  point_size=1, points(xx,ff), color=red, points_joined=false,
                  point_type=asterisk, point_size=3, points(listamix,listami),
                  color=black, points(listamax,listama), yrange=[-1,2],
                  xaxis =true, xaxis_type=solid, grid = true)$
(%t21) (Graphics)
```

Calculemos ahora los extremos de forma exacta usando las técnicas habituales del cálculo diferencial. Primero calculamos la primera derivada de f y la representamos

```
(%i22) define(df(x),diff(f(x),x));
(%o22) df(x):=cos(x/2)/2+1/100
(%i23) wxdraw2d(explicit(df(x),x,0,100), yrange=[-.6,.6],
                  xaxis =true, xaxis_type=solid, grid=true)$
(%t23) (Graphics)
```

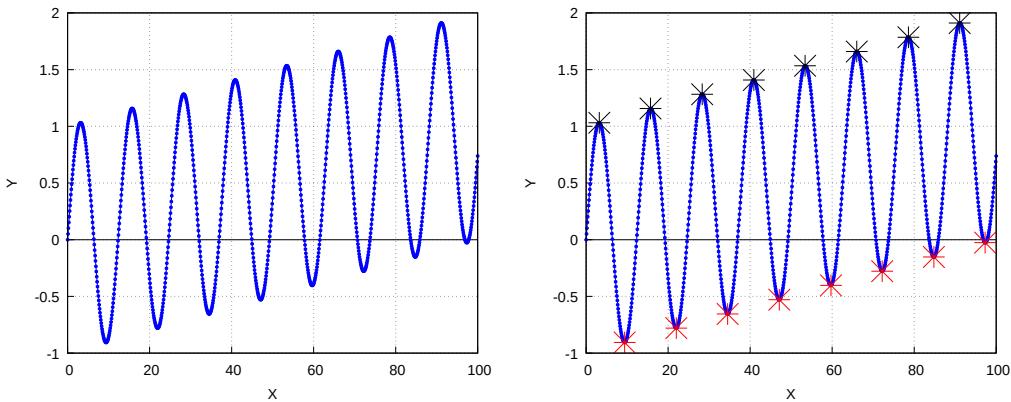


Figura 8.7: La función $f(x) = x/100 + \sin(x/2)$ (izquierda) y sus extremos (derecha) calculados a partir de la lista de sus valores.

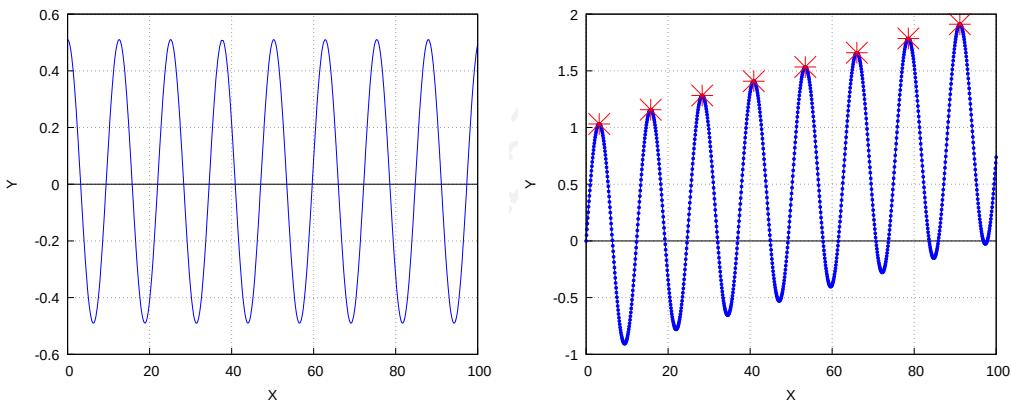


Figura 8.8: La función $f'(x) = 1/100 + 1/2 \cos(x/2)$ (izquierda) y los máximos locales exactos (derecha) de $f(x) = x/100 + \sin(x/2)$.

Para calcular los ceros usamos el comando `find_root` (dejamos como ejercicio que el lector construya la lista de los valores exactos usando el comando `solve` y las propiedades de las funciones trigonométricas). Para ello necesitamos saber entre qué valores aproximadamente se encuentran los ceros de $f'(x)$.

Un vistazo a la gráfica 8.8 (izquierda) nos indica que los ceros son cercanos a los de la función³ $\cos x/2$ por lo que crearemos una lista con dichos valores y luego en el entorno de cada uno de ellos buscamos los ceros que guardaremos en la lista `extremos`

```
(%i25) cc:makelist(float((2*k+1)*%pi),k,0,15)$
      nex:length(%);
(%o25) 16
(%i26) for k:1 thru nex do
           extre[k]:find_root(df(x)=0,x,cc[k]-0.2,cc[k]+.2)$
(%i27) extremos:makelist(extre[k],k,1,nex);
(%o27) [3.181595320736574, 9.3847752936226, ..., 97.3493695941368]
```

Comprobamos que efectivamente son extremos locales evaluando los valores de la segunda derivada:

³Por ejemplo, ejecútese la orden `plot2d([\cos(x/2)/2+1/100,\cos(x/2)], [x,0,20])`.

```
(%i28) define(d2f(x),diff(f(x),x,2));
(%o28) d2f(x):=-sin(x/2)/4
(%i30) val2d:d2f(extremos); length(%);
(%o29) [-0.2499499949989997,0.2499499949989997, ...,0.2499499949989997]
(%o30) 16
```

Así tenemos máximo, mínimo, máximo, etc. Vamos ahora a generar la lista de los máximos y compararla con la que obtuvimos antes. Para ello usaremos el comando `is` cuya sintaxis es `is(predicado)` que lo que hace es intentar evaluar el predicado dando como salida `true`, `false` o `unknown`. Lo que haremos pues es determinar si la segunda derivada es negativa, lo cual ocurre si la salida es `true`, e implica que en dicho punto se alcanza un máximo relativo. En caso contrario, si la salida es `false`, habrá un mínimo⁴

```
(%i32) sig:makelist(is(val2d[i]<0),i,1,nex);length(%);
(%o31) [true,false,true,false,true,false,true,false,
         true,false,true,false,true,false]
(%o32) 16
(%i33) posmax:sublist_indices (sig, lambda ([x], x='true));
(%o33) [1,3,5,7,9,11,13,15]
```

Las posiciones de los máximos en la lista son `[1,3,5,7,9,11,13,15]`. Ahora calculamos donde se alcanzan y el valor de los mismos:

```
(%i35) maxex:makelist(extremos[k],k,posmax); f(%);
(%o34) [3.181595320736574,15.74796593509574, ...,91.14618962125079]
(%o35) [1.031615933203364,1.157279639346956, ...,1.911261876208506]
```

y los dibujamos (ver figura 8.8 (derecha))

```
(%i36) draw2d(point_type=filled_circle, points_joined = true,
               point_size=1, points(xx,ff), color=red, points_joined=false,
               point_type=asterisk, point_size = 3,
               points(maxex,f(maxex)),yrange=[-1,2],
               xaxis =true, xaxis_type=solid, grid = true)$
```

y lo comparamos con los que calculamos antes:

```
(%i37) maxex-listamax;
(%o37) [-0.01840467926342581,0.04796593509574798, ...,0.04618962125078951]
```

cuyo posible error es menor que 0.1, siendo este el paso que usamos para crear la lista original.

Como ejercicio dejamos al lector que realice los cálculos para los mínimos locales.

Pasemos ahora a calcular la media de nuestra magnitud en cierto intervalo $[a, b]$. Para ellos usaremos la fórmula⁵

$$\bar{f} := \int_a^b f(x)dx. \quad (8.4.1)$$

⁴Aquí, en general, conviene tener cuidado con el posible valor cero, donde sabemos que el criterio de la segunda derivada no es concluyente.

⁵También podríamos tomar el valor $\bar{f} := \frac{1}{b-a} \int_a^b f(x)dx$.

Dado que estamos suponiendo que nuestros datos son una lista con los valores de x_n y $f(x_n)$ vamos a usar el método de Simpson para calcular la integral (ver el apartado 8.3, página 160). Ahora bien, como ya vimos allí nuestra lista ha de tener un número impar de elementos por lo que lo que primero haremos es definir una función que dada una lista nos devuelva la misma lista si el número de elementos es impar, y si es par que elimine, digamos el último. Para ello usaremos el comando `evenp(num)` cuya salida es `true` si el número `num` es par y `false` en otro caso. Ello lo combinaremos con el comando `rest(lista, -1)` que ya comentamos en la sección 2.2 que elimina el último elemento de la lista. Para no repetir la operación con la lista de las x y las $f(x)$ usaremos la orden `block`. Así definiremos la función `limpar` de la siguiente forma:

```
(%i38) limpar(tie,pob):=block([lon,tn,pn], lon:length(tie),
                                if evenp(lon) then block(tn:rest(tie,-1), pn:rest(pob,-1))
                                else block(tn:tie, pn:pob),[tn,pn])$
```

Como ejercicio definir una función que genere, a partir de una lista cualquiera dada, una lista con un número impar eliminando el último elemento en caso de que la lista original tenga un número par de elementos (la respuesta es la orden `limparuna` del fichero `proglst.mac`).

Vamos ahora a construir una función que calcule la media. Para ello usaremos el método de Simpson (ver página 161) para aproximar la fórmula (8.4.1). La entrada estará constituida por las listas `t` y `p` que corresponden a los valores de las $x \in [a, b]$ y sus correspondientes imágenes de $f(x)$. Es conveniente usar nuevamente el comando `block`

```
(%i39) cmedio(t,p):=block([lx,lf,nn], lx:t, lf:p, nn:length(lx),
                           sum((1/6)*(lx[2*k+1]-lx[2*k-1])*(lf[2*k-1]
                           +4*lf[2*k]+lf[2*k+1]), k,1,(nn)/2)/(lx[nn]-lx[1]))$
```

Nótese que las listas `t` y `p` deben tener el mismo número de puntos que ha de ser impar.

Un ejercicio interesante para el lector es adaptar el comando anterior para que siempre use la fórmula de Simpson con listas de tamaño impar⁶ y que además genere un aviso cuando cambie la lista. Conviene incluir el comando resultante de este ejercicio en nuestro paquete `proglst.mac` con el nombre `cmedia` ya que lo usaremos más adelante.⁷

Como vamos a utilizar estas funciones en otras sesiones conviene agregarlas a nuestro paquete `proglst.mac`. La pregunta que surge a continuación es qué intervalo tomar para calcular la media. Es evidente que nuestra función no es periódica pero tiene un comportamiento oscilante así que vamos a elegir como intervalo $[a, b]$ el intervalo entre los dos primeros máximos.

```
(%i40) pp1:1$ pp2:2$ /* 1er y 2do máximo */
      xxx:makelist(xx[k],k,indimax[pp1],indimax[pp2])$
      fff:makelist(fff[k],k,indimax[pp1],indimax[pp2])$
      lon:length(xxx);
(%o44) 126
(%i45) wxdraw2d(point_type=filled_circle,points_joined=true,point_size=1,
                  points(xxx,fff),xaxis=true,xaxis_type=solid,grid=true)$
(%t45) (Graphics)
```

⁶Basta, por ejemplo, definir las variables internas de la forma `lx:limpar(t,p)[1]`, `lf:limpar(t,p)[2]`.

⁷El lector puede encontrar la respuesta a este ejercicio en el fichero `proglst.mac`.

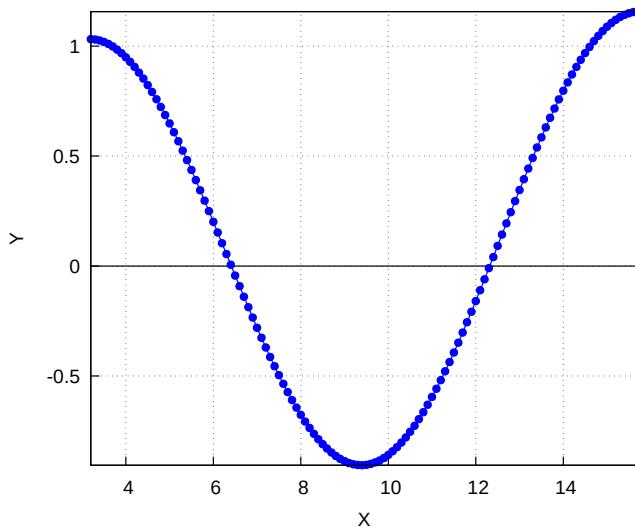


Figura 8.9: La función $f(x) = x/100 + \sin(x/2)$ representada entre sus dos primeros máximos.

Como las listas tienen un número par de elementos hay que eliminar uno de los elementos. Así que usamos la orden `limpar` que definimos antes

```
(%i46) limpar(xxx,fff)$
      xn:limpar(xxx,fff)[1]$ length(xn);
      fn:limpar(xxx,fff)[2]$ length(fn);
(%o48) 125
(%o50) 125
```

Finalmente calculamos los valores de la media usando los valores numéricos con la orden `cmedio` y lo comparamos con el valor exacto integrando nuestra función

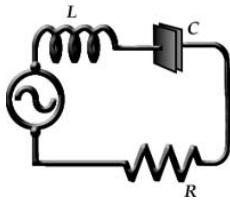
```
(%i51) medsimp:float(cmedio(xn,fn));
(%o51) 0.08058791197546415
(%i53) integrate(f(x),x,xxx[1],xxx[length(xxx)])$ 
      medex:float(%)/(xxx[length(xxx)]-xxx[1]);
(%o53) 0.08919101667914105
(%i54) abs(medsimp-medex);
(%o54) 0.008603104703676895
```

Como ejercicio dejamos al lector que calcule la media cuando tomamos como intervalos los puntos correspondientes al primer y cuarto máximos, respectivamente. ¿Qué ocurre si tomamos todo el intervalo, o distintos intervalos de longitud arbitraria? ¿Tiene sentido hacer esto último?

8.5. Algunos modelos simples descritos por EDOs.

En todos los casos resolver analíticamente cuando se pueda y además numéricamente. Dibujar la solución en cada uno de los casos.

Ejemplo 1: Se sabe que la intensidad i de circuito está gobernada por la EDO



$$L \frac{di}{dt} + Ri = U(t),$$

donde L es la impedancia, R la resistencia y $U(t)$ el voltaje, que en general depende del tiempo. Supongamos que el voltaje $U = U_0$ es constante y que $i(0) = i_0$. Encontrar la dependencia de i respecto al tiempo t . Realizar el mismo estudio si $U = U_0 \sin(\omega t)$. En ambos casos resolver la EDO y dibujar la solución en función del tiempo (eligiendo ciertos valores iniciales).

En el primer caso tenemos

```
(%i55) kill(all)$
(%i1) lru:L*'diff(i,t)+R*i-U=0;
(%o1) -U+i*R+('diff(i,t,1))*L=0
(%i2) ode2(lru,i,t);
(%o2) i=%e^(-(t*R)/L)*((%e^((t*R)/L)*U)/R+c)
(%i3) ic1(%,t=0,i=0);
(%o3) i=(%e^(-(t*R)/L)*(%e^((t*R)/L)-1)*U)/R
(%i4) define(i(L,R,U,t),second(%));
(%o4) i(L,R,U,t):=(%e^(-(t*R)/L)*(%e^((t*R)/L)-1)*U)/R
(%i5) wxplot2d( i(1,1,3,t), [t,0,10], [xlabel,"t"], [ylabel,"i(t)"])$
```

El segundo caso se deja como ejercicio, así como exportar ambos gráficos en formato pdf y jpg.

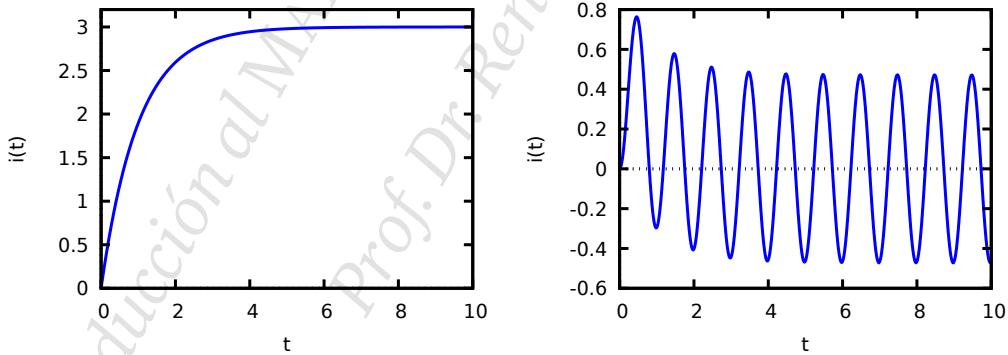
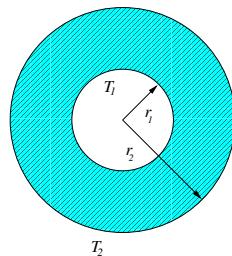
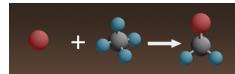


Figura 8.10: La intensidad $i(t)$ para EDO $L \frac{di}{dt} + Ri = U(t)$, con $U = U_0$ (izquierda) y $U_0 \sin(\omega t)$ (derecha).

Problema 8.5.0.1 Sea una esfera hueca homogénea de radio interior r_1 y radio exterior r_2 . Supongamos que la temperatura de la cara interior es T_1 y la exterior es T_2 . Encontrar la temperatura en la esfera en función del radio si T satisface la EDO $Q = -\kappa r^2 \frac{dT}{dr}$, $\kappa > 0$ y dibujar $T(r)$.



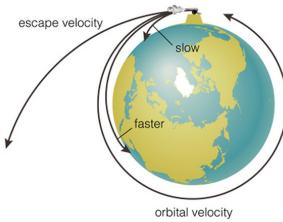
Problema 8.5.0.2 Supongamos que tenemos una reacción química $A + B \rightarrow C$ y que en $t = 0$ la concentración de A es a y la de B es b . Se sabe que la velocidad la velocidad de formación de C es proporcional a la concentración de A y B . Lo anterior nos conduce a la EDO



$$x' = \kappa(a - x)(b - x), \quad x(0) = 0.$$

Asumiendo que $a \neq b$. ¿Cómo varía x con el tiempo?

Problema 8.5.0.3 Encontrar la velocidad de escape v_E al espacio exterior de un cuerpo que se encuentre en la superficie de la Tierra si se sabe que la EDO que modeliza el movimiento es $v \frac{dv}{dr} = -\frac{gR^2}{r^2}$.



Ejemplo 2: La velocidad $v(t)$ de caída de un cuerpo en un medio viscoso se puede modelizar mediante la ecuación

$$v' = g - \kappa v^r, \quad v(0) = v_0, \quad (8.5.1)$$

donde g y κ son ciertas constantes (la gravedad y la viscosidad). Encontrar cómo varía v con el tiempo. Usar $r = 2, 3$ y $1,7$.



Este ejemplo es conveniente resolverlo numéricamente pues para el caso $r = 1,7$ no es posible analíticamente.

```
(%i1) kill(all)$
assume(vf*w>1,g>0,k>0,w>0);
(%o1) [vf*w>1,g>0,k>0,w>0]
(%i2) vel:'diff(v,t) = g*(1- w^2*v^2) ;
(%o2) 'diff(v,t,1)=g*(1-v^2*w^2)
(%i3) sol:ode2(vel,v,t);
(%o3) (log(v*w+1)-log(v*w-1))/(2*g*w)=t+%
```

Antes de resolverla hay que simplificar un poco:

```
(%i4) logcontract(%);
      ic1(%,t=0,v=v0);
      solve(%,v);
(%o4) -log((v*w-1)/(v*w+1))/(2*g*w)=t+%
(%o5) -log((v*w-1)/(v*w+1))/(2*g*w)=-(log((v0*w-1)/(v0*w+1))-2*g*t*w)/(2*g*w)
(%o6) [v=((v0*w+1)*%e^(2*g*t*w)+v0*w-1)/((v0*w^2+w)*%e^(2*g*t*w)-v0*w^2+w)]
```

Finalmente, definimos una función que usaremos más adelante y calculamos el límite asintótico cuando $t \rightarrow \infty$

```
(%i7) define(solcaso2(v0,w,g,t),rhs(%[1]))$ 
(%i8) limit( solcaso2(v0,w,g,t),t,inf);
(%o8) 1/w
```

Resolvamos el problema numéricamente usando el método de Runge-Kutta que ya vimos antes:

```
(%i9) load(diffeq);
(%o9) "/usr/share/maxima/5.36.1/share/numeric/diffeq.mac"
(%i10) kill(k,g,w)$
k:1$ g:9.8$ w:sqrt(k/g)$ x0:0$ x1:3$
f(x,y):=g*(1-w^2*y^2); h:1/20$
(%o16) f(x,y):=g*(1-w^2*y^2)
(%i18) solnum:runge1(f,0,3,h,1)$
```

Ahora representamos los valores obtenidos por los dos métodos:

```
(%i19) wxplot2d([[discrete,solnum[1],solnum[2]],solcaso2(1,w,g,x),1/w],
[x,0,3], [xlabel,"t"], [ylabel,"v(t)"], [y,1,3.75],
[legend,"solución numérica","Solución exacta","solución asintótica"]);
(%t20) (Graphics)
```

La gráfica la vemos en la figura 8.11 (izquierda)

Problema 8.5.0.4 Resolver los otros dos casos analíticamente y numéricamente y comparar los resultados.

8.6. Modelos sencillos de dinámica de poblaciones.

Imaginemos que tenemos una población de cierta especie (consideraremos que tenemos un número bastante alto de individuos) y sea $p(t)$ el número de individuos de dicha especie en el momento t (evidentemente $p(t) \in \mathbb{N}$). Sea $r(t, p)$ la diferencia entre el índice de natalidad y mortalidad de la población. Supongamos que la población está aislada (o sea, no hay emigración ni inmigración). Entonces la variación $p(t+h) - p(t)$ es proporcional a $p(t)h$ y el coeficiente de proporcionalidad es $r(t, p)$. Luego, tomando el límite $h \rightarrow 0$, obtenemos la EDO

$$p(t+h) - p(t) = r(t, p)p(t)h, \quad h \rightarrow 0, \quad \Rightarrow \quad p'(t) = r(t, p)p(t).$$

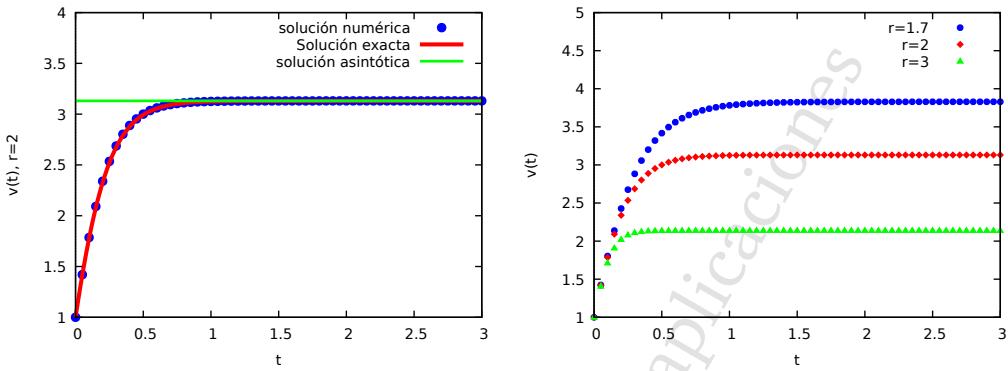


Figura 8.11: Caída de un cuerpo viscoso descrito por la EDO (8.5.1) con $r = 2$ (izquierda) y comparación de las soluciones para $r = 1, 2, 3$ (derecha).

8.6.1. El modelo malthusiano.

La ecuación más sencilla posible se obtiene si consideramos $r(t, p) = r$, constante usualmente denominada *tasa de crecimiento*. Así, la población de individuos de la especie puede ser modelizada mediante el PVI

$$p'(t) = r p(t), \quad p(t_0) = p_0, \quad r > 0. \quad (8.6.1)$$

El modelo anterior se conoce como modelo de Malthus o modelo malthusiano pues fue propuesto por el economista inglés Thomas R. Malthus (1766–1834). La solución de (8.6.1) es una exponencial

$$p_m(t) = p_0 e^{r(t-t_0)}. \quad (8.6.2)$$

Está claro que si $r < 0$ la especie está condenada a la extinción y si $r > 0$ ésta crece en proporción geométrica.

Vamos a analizar este modelo con ayuda de MAXIMA. Para ello resolvemos la ecuación diferencial

```
(%i10) kill(all)$
(%i11) kill(p,r)$
      edo:'diff(p,t,1) = r*p;
      sol:ode2(edo,p,t);
      solmal:expand(ic1(sol,t=t0,p=p0));
(%o2) 'diff(p,t,1)=p*r
(%o3) p=%c*e^(r*t)
(%o4) p=p0*e^(r*t-r*t0)
(%i5) define(maltus(t,t0,p0,r),second(solmal));
```

Veamos un ejemplo. Según datos recopilados por la Oficina del Censo de los Estados Unidos (<http://www.census.gov/population/international/data/>) la población mundial en 1950 era de $2,558 \cdot 10^9$ personas. Vamos a usar los datos de dicha organización para,

asumiendo un crecimiento exponencial, estimar el valor de r en la fórmula de Malthus. El fichero `poblacion.dat` contiene cuatro columnas de las que nos interesarán la primera y la segunda que corresponden al año y la estimación del número total de personas en el mundo para ese año según los datos recopilados en los distintos censos. Para tratar los datos necesitaremos usar los paquetes `descriptive` y `lsquares` que ya hemos visto en la sección 4.3.

```
(%i6) load (descriptive)$
(%i7) mm:read_matrix (file_search ("poblacion.dat"))$
xx:col(mm,1)$ yy:col(mm,2)$
(%i10) xx1:makelist(xx[k][1],k,1,length(xx))$
yy1:makelist(float(yy[k][1]),k,1,length(yy))$
datosfit:submatrix(mm,3,4)$
pp0:yy1[1];
(%o13) 2.557628654*10^9
```

Vamos ahora a hacer una regresión no lineal para estimar el valor de r en la fórmula de Maltus

```
(%i14) load (lsquares)$
(%i15) kill(t,p,r)$
mse : lsquares_mse (datosfit, [t, p], p=pp0*exp(r*t-r*1950));
sol:lsquares_estimates(datosfit,[t,p],p=pp0*exp(r*t-r*1950),[r]);
*****
N=      1   NUMBER OF CORRECTIONS=25
INITIAL VALUES
F=  4.524885217990591D+72   GNORM=  5.777688591667302D+74
*****
I  NFN      FUNC          GNORM          STEPLENGTH
IFLAG= -1
LINE SEARCH FAILED. SEE DOCUMENTATION OF ROUTINE MCSRCH
ERROR RETURN OF LINE SEARCH: INFO= 4
POSSIBLE CAUSES: FUNCTION OR GRADIENT ARE INCORRECT
OR INCORRECT TOLERANCES
Maxima encountered a Lisp error:
Condition in / [or a callee]: INTERNAL-SIMPLE-ERROR: Zero divisor.
Automatically continuing.
To enable the Lisp debugger set *debugger-hook* to nil.
```

Como vemos nos da un error. El ajuste no converge. Para intentar resolver el problemas echarémos mano de un pequeño truco: cambiar los valores de p por sus logaritmos, de forma que la fórmula a ajustar es ahora $\log(p) = \log(p_0) + r(t - t_0)$.

```
(%i18) xx1:makelist(xx[k][1],k,1,length(xx))$
yylog1:makelist(float(log(yy[k][1])),k,1,length(yy))$
datalog:transpose(float(matrix(xx1,yylog1)))$ 
logpp0:yylog1[1];
(%o21) 21.66234635911915
```

y ahora repetimos el ajuste

```
(%i22) kill(t,p,r)$
mse : lsquares_mse (datalog, [t, logp], logp=logpp0+r*(t-1950))$
sol:lsquares_estimates_approximate(mse, [r]);
rr:second(sol[1][1]);
*****
N=      1   NUMBER OF CORRECTIONS=25
INITIAL VALUES
F=  1.328767598368111D+03   GNORM=  2.704354242998442D+03
*****
I  NFN      FUNC          GNORM          STEPLENGTH
1    2      4.133553696694144D-01  4.764575700155741D+01  3.697740422095198D-04
2    3      9.065760190767720D-04  9.401026965441327D-15  1.000000000000000D+00
THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
(%o24) [[r=0.01731313844533347]]
(%o25) 0.01731313844533347
```

que esta vez funciona sin problemas. A continuación definimos la función de ajuste y la dibujamos (ver gráfica de la izquierda de la figura 8.12)

```
(%i26) funfit(t):=logpp0+rr*(t-1950);
(%o26) funfit(t):=logpp0+rr*(t-1950)
(%i27) plot2d([[discrete,xx1,yylog1],funfit(t)], [t,1950,2015], [y,21.6,23],
[style,points,[lines,3]],
[legend,"log(Nº personas)","Estimación maltusiana"]);
(%t27) << Graphics >>
```

Como se puede comprobar el ajuste es bastante bueno. Calculemos el coeficiente de correlación C_{corr} . Para ello tenemos que calcular la suma de los errores cuadrados SSE y la suma de los cuadrados totales SST definidos en general como sigue: Supongamos que tenemos los puntos $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ correspondientes a ciertas mediciones (en nuestro ejemplo las x_i son los años y las y_i el número de personas estimadas en cada año). Sea $f_{\text{fit}}(x)$ la función de ajuste y \bar{y} la media aritmética de los valores y_i , $\bar{y} = (\sum_{i=1}^N y_k)/N$. Entonces

$$SSE = \sum_{i=1}^N (y_k - f_{\text{fit}}(x_k))^2, \quad SST = \sum_{i=1}^N (y_k - \bar{y})^2, \quad C_{\text{corr}} = \sqrt{1 - \frac{SSE}{SST}}.$$

Si este coeficiente está próximo a uno entonces tendremos una ajuste bueno, de hecho mientras más cercano a uno sea C_{corr} mejor será el ajuste.

```
(%i28) SSE:sum( float((yylog1[k]-funfit((xx1[k])))^2) , k,1,length(xx1))$ 
SST:sum( (yylog1[k]-mean(yylog1))^2 , k,1,length(yy1))$ 
corre:sqrt(1-SSE/SST);
(%o30) 0.9953861231059471
```

Finalmente recuperamos los valores reales y dibujamos los resultados (ver gráfica de la derecha de la figura 8.12)

```
(%i31) yy2:makelist(float((yy[k][1])),k,1,length(yy))$ 
(%i32) maltus(t,1950,pp0,rr);
(%o32) 2.557628654*10^9*e^(0.01731313844533347*t-33.76061996840027)
```

```
(%i33) SSE:sum(float((yy2[k]-maltus((xx1[k]),1950,pp0,rr))^2),
           k,1,length(xx1))$  

      SST:sum( (yy2[k]-mean(yy2))^2 , k,1,length(yy1))$  

      corre:sqrt(1-SSE/SST);  

(%o35) 0.9920141011306452  

(%i36) wxplot2d([[discrete,xx1,yy2],maltus(t,1950,pp0,rr)],[t,1950,2015],
               [style,points,[lines,3]],
               [legend,"Nº personas","Estimación maltusiana"]);  

(%t36) << Graphics >>
```

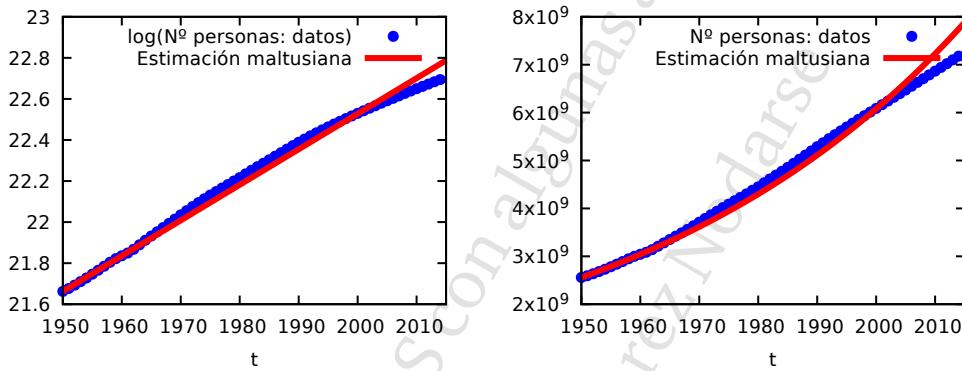


Figura 8.12: Ajuste de la población según el modelo de Malthus. Tomando logaritmo al número de individuos (izquierda) y según el número de individuos (derecha).

De la gráfica se ve claramente que para $t > 2000$ empieza a existir una divergencia clara entre la predicción maltusiana y la real. Podemos calcular algunos valores

```
(%i37) maltus(2000,1950,pp0,rr);  

(%o37) 6.078375327752732*10^9  

(%i38) maltus(2010,1950,pp0,rr);  

(%o38) 7.22732389676767*10^9  

(%i39) maltus(2025,1950,pp0,rr);  

(%o39) 9.370494836416982*10^9  

(%i40) maltus(2050,1950,pp0,rr);  

(%o40) 1.4445664958926*10^10
```

Como se puede ver el valor del año 2000 es bastante bueno en comparación con la estimación real de $6,088571383 \cdot 10^9$, pero ya la predicción del año 2010 empieza a alejarse en relación a la estimada $6,866332358 \cdot 10^9$. Nótese que la fórmula de Maltus $p(t) = p_0 e^{r(t-t_0)}$ predice que la población se duplica cada

$$2N = Ne^{r\Delta t}, \quad \Delta t = \log 2/r = 40 \text{ años.}$$

¿Es realista este modelo? Esta claro que no. Por ejemplo, si asumimos que el crecimiento es según la fórmula de Malthus con los valores que hemos obtenido tendremos que en el 2500 habrán unas $3,5 \cdot 10^{13}$ personas. Si tenemos en cuenta que la superficie de la Tierra

es de aproximadamente $5,1 \cdot 10^{14} m^2$ de las cuales sólo el 30% es tierra, nos queda que en ese año la densidad de población será de un habitante por cada $4,2m^2$ o sea !un cuadrado de $2m$ de lado!⁸

8.6.2. El modelo logístico de Verhulst.

Aunque hemos visto que el modelo funciona razonablemente bien para poblaciones grandes, hay que hacer varias correcciones pues si $p(t)$ empieza a crecer demasiado habrá muchos otros factores como la falta de espacio o de alimentos que frenará el crecimiento. Así que unos años después, en 1837, el matemático y biólogo holandés P. F. Verhulst propuso un modelo algo más realista conocido como el modelo logístico. Verhulst razonó que como estadísticamente el encuentro de dos individuos es proporcional a p^2 (¿por qué?) entonces tendremos que sustraerle al término rp un término cp^2 , de forma que la EDO que modeliza una población será

$$p'(t) = r p(t) - cp^2(t), \quad p(t_0) = p_0, \quad r, c > 0. \quad (8.6.3)$$

En general c ha de ser mucho más pequeño que r ya que si r no es muy grande la EDO (8.6.1) es una aproximación bastante buena, pero si p comienza a crecer demasiado entonces el término $-cp^2$ no se puede obviar y termina frenando el crecimiento exponencial. La EDO (8.6.3) se puede resolver fácilmente pues es una ecuación separable. Para hacerlo con MAXIMA se requiere un pequeño truco:

```
(%i46) kill(p,r,c)$
      edo:'diff(p,t,1) = p*(r-c*p);
      sol:ode2(edo,p,t);
(%o47) 'diff(p,t,1)=p*(r-c*p)
(%o48) -(log(c*p-r)-log(p))/r=t+%
(%i49) expand(ic1(sol,t=0,p=p0));
      logcontract(%);
      solexp:solve([], [p]);
(%o49) log(p)/r-log(c*p-r)/r=-t0+t-log(c*p0-r)/r+log(p0)/r
(%o50) log(-p/(r-c*p))/r=- (r*t0-r*t+log(-(r-c*p0)/p0))/r
(%o51) [p=(p0*r*%e^(r*t-r*t0))/(c*p0*(%e^(r*t-r*t0)-1)+r)]
```

Así, su solución es

$$p_l(t) = \frac{rp_0 e^{r(t-t_0)}}{r - cp_0 + cp_0 e^{r(t-t_0)}} = \frac{rp_0}{cp_0 + (r - cp_0)e^{-r(t-t_0)}}. \quad (8.6.4)$$

Nótese que $\lim_{t \rightarrow \infty} p(t) = r/c$, independientemente de p_0 . En el caso cuando $0 < p_0 < r/c$ la evolución de la población está representada en la gráfica de la izquierda de la figura 8.13.

Ya estamos en condiciones de definir la función $p_l(t)$ (8.6.4) con MAXIMA

⁸Malthus abusando de su ecuación predijo un colapso socioeconómico pues proclamó que la capacidad procreadora de los humanos es tan grande que siempre nacen más niños que los que pueden sobrevivir, lo cual se agrava por el hecho de que la cantidad de alimentos no crece exponencialmente —Malthus creía que este crecimiento era aritmético—. Así, Malthus concluyó que un gran número de humanos está predestinado a sucumbir en la lucha por la existencia. Aunque hoy día no se consideran ciertas las teorías malthusianas, en su momento tuvieron gran influencia, de hecho fue la lectura del ensayo “Acerca de los fundamentos de la

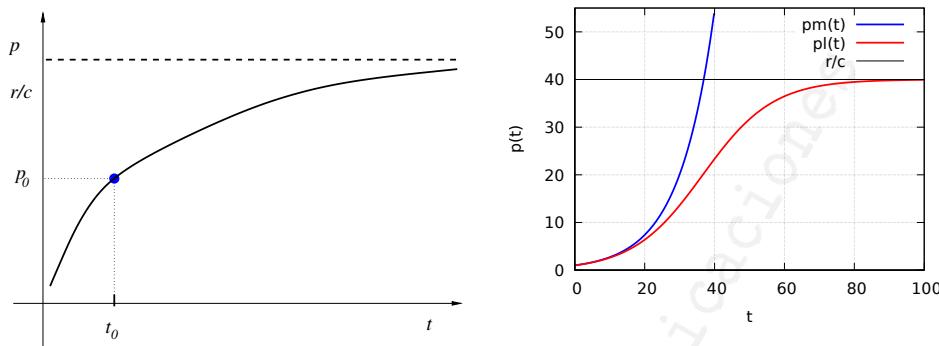


Figura 8.13: Evolución de la población según el modelo logístico $p_l(t)$ (8.6.3) (izquierda) y su comparación con el modelo malthusiano $p_m(t)$ (derecha).

```
(%i52) define(pob(t,t0,p0,r,c),second(solexp[1]));
(%o52) pob(t,t0,p0,r,c):=(p0*r*%e^(r*t-r*t0))/(c*p0*(%e^(r*t-r*t0)-1)+r)
```

En la gráfica de la derecha de la figura 8.13 comparamos la evolución según el modelo malthusiano y el logístico, para ello usamos la orden (ver figura 8.13, derecha).

```
(%o53) wxplot2d([maltus(t,0,1,0.1),pob(t,0,1,0.1,0.0025),.1/0.0025],
[t,0,100],[y,0,55],[ylabel,"p(t)"],grid2d,
[legend,"pm(t)","pl(t)","r/c"],[style,[lines,3]],
[color,blue,red,black],[box, false],[axes, solid]);
```

Vamos a aplicar el modelo logístico a la población mundial. El primer problema es como estimar los parámetros. Para ello asumimos que el valor de r es el mismo que en el modelo anterior. De la ecuación diferencial (8.6.3) se sigue, que

$$p'(t) \approx \frac{p(t+1) - p(t)}{p(t)} = r - cp(t).$$

Vamos a calcular los valores $\frac{p(t+1) - p(t)}{p(t)}$ con los datos que tenemos y luego, para cada uno de dichos valores calculamos un valor de c . Como estimación del valor c usaremos la media los valores obtenidos. Así, obtenemos la función $p_l(t)$

```
(%i54) ratprint: false$ 
(%i55) for k:1 thru length(yy1)-1 do block(
  cc[k]:second(float(solve((yy1[k+1]-yy1[k])/yy1[k]=rr-c*yy1[k],c)[1])));
(%i56) coeffc:makelist(cc[k],k,1,length(yy1)-1)$
  cc:mean(coeffc);
  var(coeffc);
(%o57) 8.361178966147587*10^-14
(%o58) 5.013464530632254*10^-25
(%i59) pob(t,1950,pp0,rr,cc);
```

y la comparamos con los datos reales y con el modelo malthusiano

"población" la que indujo a Darwin el mecanismo de selección natural universalmente aceptado hoy día.

```
(%i60) plot2d([[discrete,xx1,yy1],pob(t,1950,pp0,rr,cc)], [t,1950,2015],
[style,points,[lines,3]],
[legend,"log(datos)","Estimación logística"]);
(%t60) << Graphics >>
(%i61) plot2d([[discrete,xx1,yy1],pob(t,1950,pp0,rr,cc),
maltus(t,1950,pp0,rr)], [t,1950,2015],
[y,2.5*10^9,8.5*10^9],[style,points,[lines,3],[lines,3]],
[legend,"log(Nº personas: datos)","Estimación logística"]);
(%t61) << Graphics >>
```

Según el modelo malthusiano el equilibrio $p(t_{\text{equil}}) = r/c = 2,07 \cdot 10^{11}$ se alcanzará en el 2203

```
(%i62) ee:rr/cc;
(%o62) 2.070657561024615*10^11
(%i63) tfin:1950+1/rr*log(ee/pp0);
(%o63) 2203.793146566312
(%i64) maltus(tfin,1950,pp0,rr);
(%o64) 2.070657561024626*10^11
```

Una gráfica ilustra muy bien lo que ocurre (véase la gráfica de la derecha en la figura 8.14).

```
(%i65) plot2d([pob(t,1950,pp0,rr,cc),maltus(t,1950,pp0,rr),ee ],
[t,1950,2300],[style,[lines,3]], [legend,"Estimación logística",
"Estimación malthusiana","valor estable"]);
(%t65) << Graphics >>
```

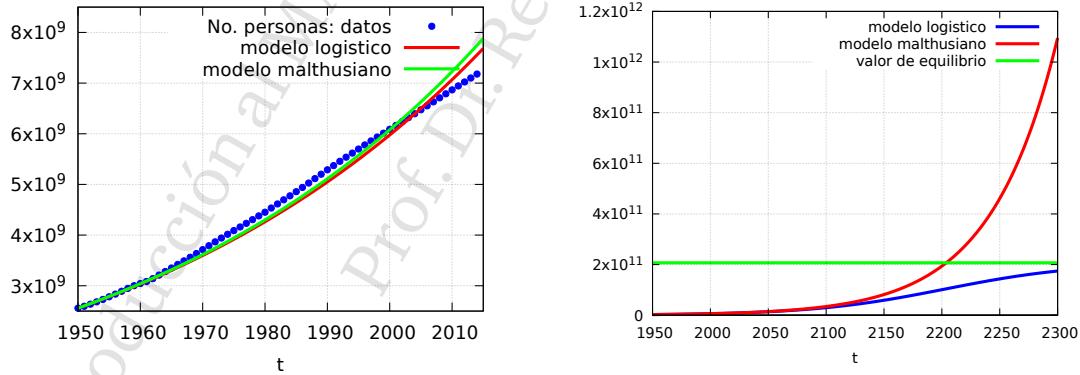


Figura 8.14: Comparación de los modelos logístico $p_l(t)$ (8.6.4) y malthusiano $p_m(t)$ (8.6.2) (izquierda) y predicción de ambos modelos (derecha).

Obviamente el modelo logístico sigue siendo muy simple ya que no tiene en cuenta ni las guerras (habituales desde hace cientos de años), ni las epidemias, hambrunas, etc.

8.6.3. Un modelo de poblaciones fluctuantes.

En muchas ocasiones la ecuación (8.6.3) se suele reescribir de la forma:

$$p'(t) = r p(t) \left(1 - \frac{p(t)}{K} \right), \quad p(t_0) = p_0, \quad r, K > 0.$$

donde K es la *capacidad de carga* del medio, que no es más que el tamaño máximo de población que el medio puede soportar indefinidamente en un periodo determinado (se puede comprobar que $\lim_{t \rightarrow \infty} p(t) = K$). Obviamente el caso más sencillo es cuando la capacidad de carga K es constante, no obstante puede darse el caso que dicha capacidad varíe cíclicamente, de forma que tengamos la ecuación

$$p'(t) = r p(t) \left(1 - \frac{p(t)}{K_0 + K_1 \cos(\omega t + \phi)} \right), \quad p(t_0) = p_0, \quad r, K_0, K_1 > 0.$$

Este modelo fue propuesto y estudiado por Robert May en 1976.⁹

Vamos a estudiar que ocurre con este modelo de poblaciones fluctuantes asumiendo que $K_0 > K_1$. Por simplicidad trabajaremos con la ecuación equivalente

$$p'(t) = p(t) \left(1 - \frac{p(t)}{1 + a \cos(\omega t + \phi)} \right), \quad a \in (0, 1). \quad (8.6.5)$$

La ecuación anterior no es resoluble analíticamente así que la resolveremos numéricamente usando el método de Runge-Kutta con la orden rk. Antes de empezar nuestro estudio debemos cargar los paquetes necesarios:

```
(%i1) load(dynamics)$ load(draw)$
(%i3) define(g(t,p),p*(1-p/(1+a*cos(w*t+phi))))$
```

```
(%o3) g(t,p):=p*(1-p/(a*cos(t*w+phi)+1))
```

Comenzamos analizando el caso del modelo logístico *puro* que ya discutimos en el apartado anterior, para lo que fijamos $a = 0$ (ver la ecuación (8.6.3))

```
(%i4) a:0$ p0:0.5$ h:0.1$ g(t,p);
      solnum0:rk(g(t,p),p,p0,[t,0.0,50,h])$
```

```
(%o7) (1-p)*p
```

A partir de la salida de rk creamos las listas de las abscisas y las ordenadas y las dibujamos:

```
(%i8) pp0:makelist(solnum0[k][2],k,1,length(solnum0))$
```

```
tt0:makelist(solnum0[k][1],k,1,length(solnum0))$
```

```
wxdraw2d(point_type=filled_circle, points_joined = true,
```

```
 xlabel = "t", ylabel = "p(t)", yrange=[0.45,1.1],
 point_size = 0.5, points(tt0,pp0))$
```

```
(%t11) (Graphics)
```

Como nos interesará lo que pasa para tiempos largos vamos a quedarnos con la segunda mitad de la lista —la linea roja en la gráfica 8.15 (izquierda)—

⁹Ver el artículo *Models for single populations*. Theoretical Ecology: Principles and Applications, Blackwell Scientific Publications, 1976, Robert M. May (Eds.), pp. 5-29.

```
(%i12) cola0:floor(length(solnum0)*.5);
        colapp0:rest(pp0,cola0)$
        colatt0:rest(tt0,cola0)$
        wxdraw2d(point_type=filled_circle, points_joined=true,
                  xlabel = "t", ylabel = "p(t)", yrange=[0.45,1.1],
                  point_size = 1, points(tt0,pp0),
                  color=red,points(colatt0,colapp0))$%
(%o12) 250
(%t15) (Graphics)
```

Pasemos ahora al caso fluctuante. Fijamos, por ejemplo,

```
(%i16) a:0.4$ w:1.0$ p0:0.5$ h:0.1$ phi:0$
```

y encontramos la solución numérica

```
(%i17) solnum:rk(g(t,p),p,p0,[t,0.0,50,h])$
```

que nos permite representar la solución del modelo fluctuante (azul) y compararla a la del modelo logístico *puro* (rojo) —ver gráfica 8.15 (derecha)— para todo el intervalo —en este caso $[0, 50]$ —

```
(%i21) pp:makelist(solnum[k][2],k,1,length(solnum))$%
        tt:makelist(solnum[k][1],k,1,length(solnum))$%
        wxdraw2d(point_type=filled_circle, points_joined = true,
                  xlabel = "t", ylabel = "p(t)", yrange=[0.45,1.3],
                  point_size = 1, points( tt,pp), color=red,
                  points( tt0,pp0))$%
(%t24) (Graphics)
```

o, si así lo preferimos, para tiempos *largos*, por ejemplo, a partir de la mitad del tiempo total escogido —en este caso de $[25, 50]$ — y cuya gráfica la podemos ver en la figura 8.16 (izquierda)

```
(%i25) cola:floor(length(solnum)*.5);
        colapp:rest(pp,cola)$
        colatt:rest(tt,cola)$
        wxdraw2d(point_type=filled_circle, points_joined=true,
                  point_size = 1, points(colatt,colapp), color=red,
                  points(colatt0,colapp0), xlabel="t",ylabel="p(t)")$%
(%t28) (Graphics)
```

Está claro que la solución oscila alrededor del $y = 1$. Vamos ahora a encontrar donde tiene sus máximos. Dado que es una lista usaremos los comandos que construimos en el apartado 8.4. Lo primero que tenemos que hacer es cargar el paquete¹⁰ `proglist.mac`

```
(%i29) file_search_maxima : cons(sconcat(
```

¹⁰Recuérdese que MAXIMA debe saber donde buscar por lo que quizás haya que especificarle el camino (*path*) usando, como ya vimos en la página 13, la opción (*Add to path*) en la pestaña “Maxima” del menú de wxMAXIMA.

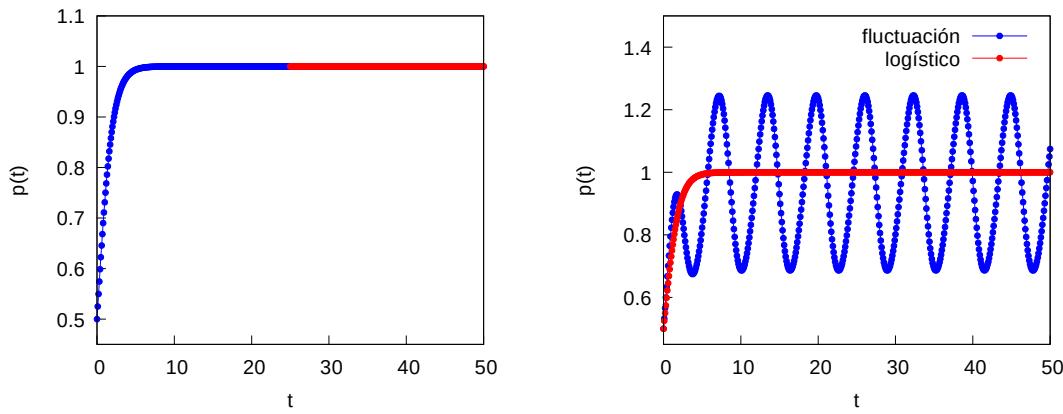


Figura 8.15: Gráfica de la solución del modelo logístico descrito por la ecuación (8.6.5) cuando $a = 0$ (izquierda) y modelo de poblaciones fluctuantes de May descrito por (8.6.2).

```
"/home/renato/###.{lisp,mac,mc}"), file_search_maxima)$
(%i30) load("proglist");
(%o30) "/home/renato/proglist.mac"
```

Ahora encontramos los máximos locales así como sus valores

```
(%i31) indimax:lismax(colapp);
      mt:makelist(colatt[k],k,indimax);
      mp:makelist(colapp[k],k,indimax);
(%o31) [11,74,137,200]
(%o32) [26.0,32.3,38.6,44.9]
(%o33) [1.245676086805322,1.245849815973797,
      1.245935016172712,1.245931228967183]
```

y los representamos —ver gráfica de la derecha en la figura 8.16—

```
(%i34) wxdraw2d(point_type=filled_circle, points_joined = true,
      point_size = 1, points( colatt,colapp), color=red,
      points(colatt0,colapp0), point_type=asterisk,color=black,
      point_size=3, points_joined = false, points(mt,mp),
      yrange=[0.65,1.35], xlabel = "t", ylabel = "p(t)")$
```

(%t34) (Graphics)

Veamos ahora el crecimiento medio de la población en un periodo. Primero definimos el índice de los máximos que usaremos (en este caso el primero y el segundo) y construimos la lista de los tiempos y de la función $p(t)$

```
(%i35) pp1:1$ pp2:2$
      tie:makelist(colatt[k],k,indimax[pp1],indimax[pp2])$
```

pob:makelist(colapp[k],k,indimax[pp1],indimax[pp2])\$

lon:length(tie);

(%o39) 64

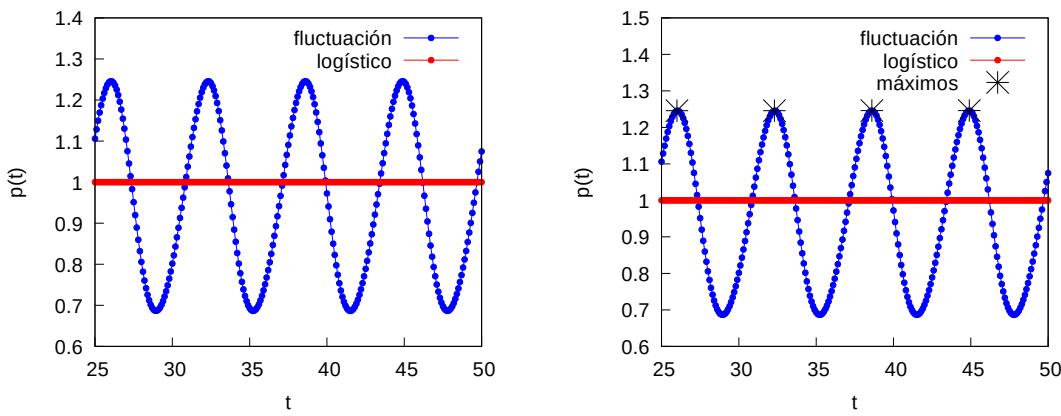


Figura 8.16: Gráfica de la solución asintótica (tiempos grandes) del modelo de poblaciones fluctuantes de May descrito por (8.6.2) (derecha) y posición de los máximos (derecha).

Finalmente con la orden `cmedia` que definimos en el ejercicio de la página 171 encontramos el promedio de la solución en un periodo:

```
(%i40) m1:cmedia(tie,pob);
(%t40) "Longitud de la lista par, eliminando el último elemento"
(%o40) 0.9538472231637653
```

También se pueden usar las órdenes `limparuna` o `limpar` que comentamos en la página 171.

```
(%i41) tn:limparuna(tie)$ pn:limparuna(pob)$
      m0:cmedio(tn,pn);
(%o43) 0.9538472231637653
(%i44) tn:limpar(tie,pob)[1]$ length(tn)$
      pn:limpar(tie,pob)[2]$ length(pn)$
      m1:cmedio(tn,pn);
(%o46) 0.9538472231637653
```

Como ejercicio final repetir el estudio para otros parámetros, por ejemplo, $a = 0,4$, $y \phi = \pi/6$ y compararlos con el caso aquí considerado.

8.6.4. El modelo depredador-presa de Lotka-Volterra.

Antes de terminar este apartado conviene mencionar uno de los modelos más simples de crecimiento de poblaciones cuando intervienen dos especies, una presa y un depredador, conocido como modelo de Lotka-Volterra. Si denotamos por $N(t)$ el número de individuos (presas) y por $P(t)$ en número de depredadores, el modelo de Lotka-Volterra está definido por el siguiente sistema no lineal acoplado

$$\frac{dN}{dt} = N(a - bP), \quad \frac{dP}{dt} = P(cN - d), \quad a, b, c, d > 0, \quad N(t_0) = N_0, \quad P(t_0) = P_0.$$

Este sistema es, esencialmente, el mismo que el sistema (6.3.1) discutido en el apartado 6.2 así que remitimos al lector a la discusión allí expuesta.

8.7. Encontrando los ceros de una función definida por una lista.

Vamos a considerar ahora un problema parecido al que discutimos en el apartado §8.4. Vamos a suponer que no tenemos la expresión explícita de una función, sino que tenemos una lista de los puntos $(x_n, f(x_n))$ de dicha función y nos interesa encontrar los ceros de la misma.

La mayoría de los programas que calculan numéricamente los ceros lo que hacen es, en primer lugar, comprobar si en los extremos del intervalo donde se buscan los ceros hay un cambio de signo pues, si la función es continua, entonces el Teorema de Bolzano asegura que hay al menos un cero, y luego aplicar algunos de los algoritmos estándar. Dado que nosotros queremos calcular el valor de los ceros de una función que viene dada por un conjunto de valores (por ejemplo, la solución de una ecuación diferencial como por Ejemplo 2 las de la sección 5 de la página 174, o las del apartado §8.6.3) y no por su expresión explícita, este método no nos sirve. No obstante, en este caso es relativamente simple pues dibujando la gráfica de los valores de la función podemos ver si la misma tiene ceros y proceder a encontrarlos de forma aproximada.

Vamos a mostrar un ejemplo sencillo de como proceder. Comenzaremos con una función cuyos ceros sepamos y que nos sirva de test. Para que no sea demasiado sencilla elegiremos la función

$$f(x) = (x - 2\gamma)(x - e)(x - \pi) \quad (8.7.1)$$

donde $\gamma = \lim_{n \rightarrow \infty} [\sum_{k=1}^n \frac{1}{k} - \ln(n)] = 0,57721\dots$, $e = 2,71828\dots$ es la base de los logaritmos naturales y $\pi = 3,141592\dots$

Lo que haremos es definir la función (nos aseguramos que son todo valores numéricos como se pretende usando el comando `float`) y hacemos una lista de los valores de la misma que nos servirán como los valores numéricos de la función cuyos ceros queremos encontrar:

```
(%i1) f(x):=(x-float(%e))*(x-float(%pi))*(x-2.0*float(%gamma));
(%o1) f(x):=(x-float(%e))*(x-float(%pi))*(x-2.0*float(%gamma))
(%i4) xx:makelist(float(k/200.0),k,0,800)$
      yy:f(xx)$
      lon:length(yy);
(%o1) 801
```

Si la representamos vemos que tiene efectivamente tres ceros entre 0 y 4.

```
(%i5) wxplot2d([[discrete,xx,yy],f(x)], [x,0,4],
              [xlabel,"t"], [ylabel,"y(t)"], [style,points,[lines,3]],
              [point_type,diamond], [legend,"lista","funcion"]);
(%t5) (Graphics)
```

Lo que vamos a hacer para encontrar la posición del cero y así poder calcularlo es generar una lista con los signos de los valores de la lista `yy` y la dibujamos

```
(%i7) lis:signum(yy)$
      wxplot2d([[discrete,lis]], [y,-1.1,1.1]);
(%t7) (Graphics)
```

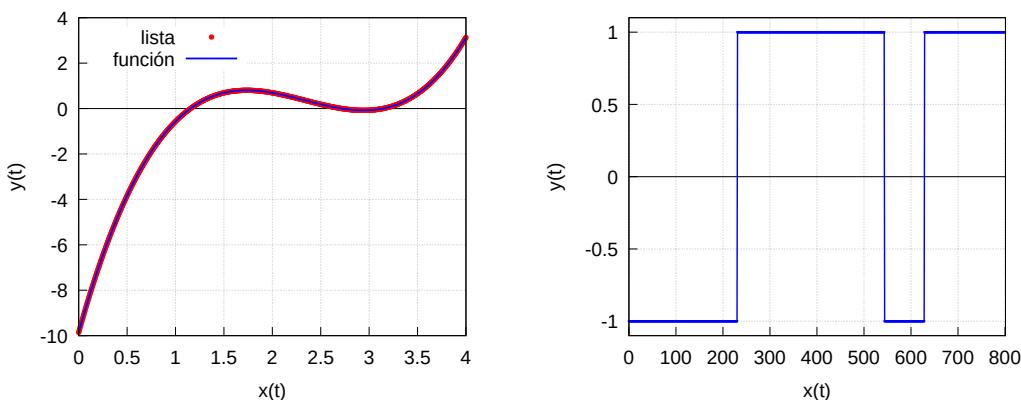


Figura 8.17: A la izquierda la función $f(x)$ definida en (8.7.1) y a la derecha la función $\text{signo}(f(x))$.

En la segunda gráfica se ve claramente los tres ceros así como su posición en la lista yy .

Lo que haremos es buscar donde ocurre el primer cambio de signo en la lista yy y eso nos dará la posición del cero. Conocida la posición podemos encontrar el correspondiente valor en la lista xx . Para ello usaremos el comando `sublist_indices` que discutimos en la página 16 y que usamos en combinación con las funciones indeterminadas `lambda` para encontrar los extremos de una función en la sección 8.4.

Por ejemplo si escribimos

```
(%i8) block([ll,lsig],ll:lis,lsig:sublist_indices(ll,lambda([x], x=1.0)));
(%o8) [232,233,234,235, ..., 797,798,799,800,801]
```

obtenemos que el primer cambio de signo de $-$ a $+$ ocurre en la posición 232. No obstante la lista podría empezar por signos positivos por lo que la orden anterior nos daría 1:

```
(%i9) block([ll,lsig],ll:-lis,lsig:sublist_indices(ll,lambda([x],x=1.0));
(%o9) [1,2,3,4,5,6,7 ..., 626,627,628,629]
```

En nuestro ejemplo además sabemos que la posición de cambio es la misma pues en la orden anterior simplemente hemos cambiado la lista yy por $-yy$. Por tanto, si la lista empieza con números positivos la orden correcta ha de ser

```
(%i10) block([ll,lsig],ll:-lis,lsig:sublist_indices(ll,lambda([x],x=-1.0));
(%o10) [232,233,234,235, ..., 799,800,801]
```

Conviene recordar que por el momento nos interesa encontrar únicamente el primer cero de nuestra función, así que definiremos dos funciones, una para encontrar el primer cero cuando la función empieza tomando valores positivos y otra para cuando empieza con valores negativos

```
(%i11) posmenosmas(lista):=block([ll,lsig],
      lsig:sublist_indices(ll,lambda([x],x=1.0)),first(lsig));
(%i13) posmenosmas(lis); posmenosmas(-lis);
(%o12) 232
(%o13) 1
```

```
(%i14) posmasmenos(lista):=block([ll,lsig], ll:signum(lista),
      lsig:sublist_indices(ll,lambda([x],x=-1.0)),first(lsig));
(%i16) posmasmenos(lis); posmasmenos(-lis);
(%o15) 1
(%o16) 232
```

Es importante destacar que las órdenes anteriores están pensadas para listas numéricas en coma flotante, así que en caso de que no lo sean hay que usar el comando `float`.

Teniendo en cuenta que

```
(%i17) lis[1]<0 or lis[1]<0;
(%o17) true
(%i18) -lis[1]<0 or -lis[1]<0;
(%o18) false
```

podemos combinarla para crear un único comando que nos dé la posición del primer cambio de signo de cualquier lista de números

```
(%i19) poscero(lista):=block ([ll],ll:lista, if ll[1]<0 or ll[1]<0 then
      posmenosmas(ll) else posmasmenos(ll) );
```

Para comprobarlo hacemos

```
(%i20) pc:poscero(yy);
(pc) 232
(%i21) poscero(-yy);
(%o21) 232
(%i24) yy[pc-1]; yy[pc]; yy[pc+1];
(%o22) -0.01384072053604195
(%o23) 0.001766064549686498
(%o24) 0.01719538434482167
```

que nos dice que, efectivamente, el primer cambio de signo de $f(x)$ ocurre entre la posición $pc-1=231$ y $pc=232$ de la lista yy de sus valores. Conocida dicha posición podemos calcular los valores de la x

```
(%i27) xx[pc-1]; xx[pc]; xx[pc+1];
(%o25) 1.15
(%o26) 1.155
(%o27) 1.16
```

y a partir de ellas encontramos una aproximación del primer cero por ejemplo así

```
(%i28) cero(lista,m):=block([ll],ll:lista,(ll[m-1]+ll[m]+ll[m+1])/3);
```

que podemos comprobar

```
(%i29) z1:cero(xx,poscero(yy));
(z1) 1.155
```

Pasemos a describir como encontrar el segundo cero. Para ello lo que haremos es crear una nueva lista, que llamaremos `yy2`, que obtendremos tras quitarle a la lista original `yy` los elementos que tienen el signo constante. Ello lo haremos con el comando `rest` que ya hemos usado varias veces a lo largo de este libro (ver página 15)

```
(%i31) yy2:rest(yy,pc)$ xx2:rest(xx,pc)$
```

y para la nueva lista encontramos la posición del primer cambio de signo y su cero con las órdenes `poscero` y `cero`, respectivamente

```
(%i32) pc2:poscero(yy2);
(pc2) 313
(%i33) z2:cero(xx2,pc2);
(z2) 2.72
```

Finalmente repetimos estas operaciones hasta cubrir todos los ceros que hemos visto en las gráficas del principio de esta sección

```
(%i35) yy3:rest(yy2,pc2)$ xx3:rest(xx2,pc2)$
(%i36) pc3:poscero(yy3);
(pc3) 85
(%i37) z3:cero(xx3,pc3);
(z3) 3.145
```

Finalmente comparamos los ceros numéricos con los reales de nuestro ejemplo

```
(%i39) z1; float(2*%gamma);
(%o38) 1.155
(%o39) 1.154431329803065
(%i41) z2; float(%e);
(%o40) 2.72
(%o41) 2.718281828459045
(%i43) z3; float(%pi);
(%o42) 3.145
(%o43) 3.141592653589793
```

Conviene hacer notar que hay muchas formas de definir la orden `cero` que usamos aquí. Por ejemplo, encontrados los dos puntos donde cambia el signo de nuestra lista podemos unir dichos puntos por una recta y ver donde dicha recta corta el eje de las x . Dicha función es la siguiente:

```
(%i38) newcero(listax,listay,m):=block([lx,ly,pc],pc:m,lx:listax,ly:listay,
    lx[pc-1]-ly[pc-1]*(lx[pc]-lx[pc-1])/(ly[pc]-ly[pc-1]));
```

Es importante asegurarse que efectivamente hay un cambio de signo pues de lo contrario podría ocurrir que la función anterior no estuviese definido (¿por qué?). Si comparamos con los ceros reales vemos que la aproximación es mucho mejor:

```
(%i41) z1; newcero(xx,yy,pc); float(2*%gamma);
(%o39) 1.155
```

```
(%o40) 1.154434199759916
(%o41) 1.154431329803065
(%i44) z2; newcero(xx2,yy2,pc2); float(%e);
(%o42) 2.72
(%o43) 2.718291504088847
(%o44) 2.718281828459045
(%i47) z3; newcero(xx3,yy3,pc3); float(%pi);
(%o45) 3.145
(%o46) 3.14157717194661
(%o47) 3.141592653589793
```

Conviene incluir nuestras funciones en el paquete `proglist.mac` que creamos al final de la sección 8.4.1 de forma que podamos cargarlos en cualquier sesión de MAXIMA con la orden `load`.

8.7.1. Ejemplo interesante: El tiro parabólico con resistencia del aire.

En el Ejemplo 2 de la sección 5 (página 174) vimos la caída de un cuerpo en un medio viscoso. Vamos a considerar ahora un problema algo más complejo: el tiro parabólico. Imaginemos que tenemos una bola de cierto material de diámetro d que es lanzada horizontalmente con cierta velocidad inicial v_0 tal y como se muestra en el siguiente esquema: Las leyes de Newton establecen que nuestra bola satisface las ecuaciones siguientes:

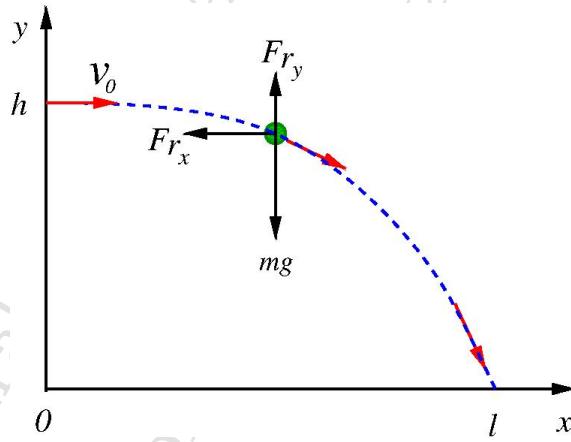


Figura 8.18: Esquema del movimiento de una esfera en un medio viscoso.

$$m \frac{d^2}{dt^2}y(t) = -mg + F_{r_x}(v), \quad m \frac{d^2}{dt^2}x(t) = F_{r_y}, \quad (8.7.2)$$

con las siguientes las condiciones iniciales

$$y(0) = h, \quad v_y(0) = \frac{d}{dt}y(0) = 0, \quad x(0) = 0, \quad v_x = \frac{d}{dt}x(0) = v_0. \quad (8.7.3)$$

En (8.7.2) F_{r_x} y F_{r_y} son las componentes horizontal y vertical de la fuerza de rozamiento del medio, respectivamente, m es la masa de la bola, g la aceleración de la gravedad.

Existen varias expresiones para el valor de dicha fuerza de rozamiento. Una posibilidad es que se cumpla la Ley de Stokes que establece que dicha fuerza es proporcional a la

velocidad y actúa en sentido contrario a la misma, i.e.,

$$F_{r_x} = -3\pi\eta d v_x, \quad F_{r_y} = -3\pi\eta d v_y, \quad v_x = \frac{dx}{dt}, \quad v_y = \frac{dy}{dt},$$

donde η es el coeficiente de viscosidad del medio. Usando que $m = \pi\rho d^3/6$, donde ρ es la densidad de la esfera, obtenemos el siguiente sistema de ecuaciones desacopladas (es decir cada ecuación es independiente de la otra):

$$\frac{d^2y}{dt^2} = -g - \alpha \frac{dy}{dt}, \quad \frac{d^2x}{dt^2} = -\alpha d \frac{dx}{dt}, \quad \alpha = \frac{18\eta}{\rho d^2}.$$

En este caso, el problema es relativamente sencillo de resolver pues tenemos dos ecuaciones diferenciales que podemos resolver independientemente una de la otra y de forma exacta. La solución es:

$$y(t) = h - \frac{g (e^{-\alpha t} + \alpha t - 1)}{\alpha^2}, \quad (8.7.4)$$

$$x(t) = \frac{v_0 (1 - e^{-\alpha t})}{\alpha}, \quad (8.7.5)$$

con las condiciones iniciales (8.7.3). La primera ecuación (8.7.4) nos permite encontrar el tiempo que tarda una gota en caer y , conociendo dicho tiempo, podemos encontrar con la segunda ecuación (8.7.5) la distancia x_c a la que cae. Para encontrar el tiempo de caída hemos de resolver la ecuación

$$F(t) = e^{-\alpha t} - \frac{h\alpha^2}{g} - 1 + \alpha t = 0, \quad (8.7.6)$$

que equivale a encontrar los ceros de la función $F(t)$ que está dada de forma implícita y por tanto podemos usar los comandos estándar como por ejemplo `find_root`. Por ello consideraremos un problema más complicado.

En general es bien sabido que la fuerza de rozamiento sigue la *Ley de Stoke* si el número de Reynolds definido por

$$\text{Re} = \frac{\rho_a v d}{\eta},$$

donde ρ_a es la densidad del medio y $v = \sqrt{v_x^2 + v_y^2}$, es mucho más pequeño que la unidad. En general, la fuerza de rozamiento es igual a

$$\vec{F}_r = - \left(\frac{\pi d^2 \rho_a}{8} \right) C_d(v) v \vec{v}, \quad (8.7.7)$$

donde el *coeficiente de arrastre* C_d se puede expresar como

$$C_d = \frac{24}{\text{Re}} \left(1 + \frac{\sqrt{\text{Re}}}{\delta_0} \right)^2,$$

siendo la constante $\delta_0 = 9,06$. Considerando las componentes x e y de la fuerza (8.7.7), las ecuaciones que describen el movimiento del tiro parabólico son:

$$m \frac{dv_x}{dt} = - \left(\frac{\pi d^2 \rho_a}{8} \right) C_d(v) v v_x, \quad m \frac{dv_y}{dt} = -mg - \left(\frac{\pi d^2 \rho_a}{8} \right) C_d(v) v v_y, \quad (8.7.8)$$

donde $v_x = dx/dt$, $v_y = dy/dt$ y $v = \sqrt{v_x^2 + v_y^2}$. Si ahora usamos que la masa de la esfera se puede calcular por la fórmula $m = \pi \rho d^3/6$, entonces las ecuaciones anteriores (8.7.8) se transforman en el sistema

$$\begin{aligned}\frac{dy}{dt} &= v_y, & \frac{dv_y}{dt} &= -g - \hat{\alpha}(v)v_y, \\ \frac{dx}{dt} &= v_x, & \frac{dv_x}{dt} &= -\hat{\alpha}(v)v_x,\end{aligned}\tag{8.7.9}$$

con las condiciones iniciales (8.7.3), donde

$$\hat{\alpha}(v) = \frac{18\eta}{\rho d^2} \left(1 + \frac{1}{\delta_0} \sqrt{\frac{d\rho_a v}{\eta}}\right)^2, \quad v = \sqrt{v_x^2 + v_y^2},$$

y ρ_a es la densidad del medio.

Las ecuaciones (8.7.9) son no lineales y no se pueden resolver analíticamente por lo que hay que hacerlo numéricamente. Para encontrar la distancia a la que cae nuestra esfera tenemos que calcular el tiempo de caída a y, conocido dicho tiempo, encontrar el valor de la x . Para calcular el tiempo de vuelo de nuestra partícula tenemos que encontrar para qué valor de t se anula $y(t)$, o sea los ceros de una función definida a partir de la lista de sus valores.

Como ejemplo vamos a tomar una esfera de acero de 1 milímetro de diámetro que cae en el aire desde una altura de 20 metros con una velocidad inicial $v_0 = 10 \text{ m/s}$. Para el aire tomaremos el valor del coeficiente de viscosidad $\eta = 1,85 \cdot 10^{-5} \text{ N}\cdot\text{s}/\text{m}^2$ (correspondiente a una temperatura de 25°C) y su densidad $\rho_a = 1,29 \text{ kg/m}^3$. El valor de la aceleración de la gravedad es $g = 9,8 \text{ m/s}^2$. Vamos a resolver nuestro sistema y comparar lo que ocurre en caso de que no hubiese resistencia del aire.

Movimiento sin resistencia del aire.

Estudiemos primero el caso del tiro parabólico sin resistencia del aire. En este caso las ecuaciones son (8.7.2) cuando $F_r = 0$. Aunque en este caso el problema se puede resolver analíticamente lo haremos de forma numérica para luego compararlo con el caso cuando hay resistencia del aire.

Comenzaremos cargando el paquete `proglisit.mac` que hemos creado para poder usar las órdenes del apartado anterior.

```
(%i1) load("proglisit.mac");
(%i5) g:9.8$ h:20$ v0y:0$ v0x:10$ 
(%i7) define(fy(t,y,dy,x,dx) , -g);
      define(fx(t,y,dy,x,dx) , 0);
(%o6) fy(t,y,dy,x,dx):=-9.8
(%o7) fx(t,y,dy,x,dx):=0
```

Para resolver el sistema (8.7.9) usamos la orden `rk` que ejecuta un Runge-Kutta (ver la página 111)

```
(%i12) sol:rk( [dy,fy(t,y,dy,x,dx),dx, fx(t,y,dy,x,dx)],
```

```
[y,dy,x,dx],[h,v0y,0,v0x],[t,0,3,0.01])$  
length(sol);  
til:makelist(sol[k][1],k,1,length(sol))$  
yy1:makelist(sol[k][2],k,1,length(sol))$  
xx1:makelist(sol[k][4],k,1,length(sol))$  
(%o9) 301
```

Y, como ya hicimos, representamos la función $y(t)$ a partir de la lista de valores

```
(%i13) wxplot2d([discrete,til,yy1],[xlabel,"t"], [ylabel,"y(t)])$
```

de donde vemos que tiene un cero. Calculamos la posición del mismo en la lista así como su valor con los comandos `poscero` y `newcero`, respectivamente

```
(%i14) pc:poscero(yy1);  
(pc) 204  
(%i15) tc:newcero(til,yy1,pc);  
(tc) 2.020304358780547
```

Creamos las nuevas listas de las x e y

```
(%i18) nx1:rest(xx1,pc-length(yy1))$  
ny1:rest(yy1,pc-length(yy1))$  
lx1:last(nx1);  
(lx1) 20.30000000000002
```

Lo anterior nos dice que la esfera cae al suelo en 2,02 segundos y alcanza los 20,30 metros. Dibujemos la trayectoria (ver figura 8.19, izquierda)

```
(%i19) wxdraw2d(grid=true,line_type=solid,line_width=5,nticks=100,  
xaxis =true,xaxis_type=solid,yaxis =true,yaxis_type=solid,  
xlabel="x(t)",ylabel="y(t)", point_type=filled_circle,  
point_size=1, color=red, title="Trayectoria",  
yrange=[0,h+.2], xrange=[0,1x1+.2], points(nx1,ny1));
```

Movimiento con resistencia del aire.

Vamos ahora a resolver el caso cuando la esfera cae con resistencia del aire, es decir cuando las ecuaciones de movimiento vienen dadas por el sistema (8.7.9).

Comenzamos fijando los parámetros de las ecuaciones

```
(%i25) g:9.8; eta:1.85*10^(-5);  
d:1000*10^(-6); rho:7800.0;  
rhoa:1.29; delta0:9.06;
```

definiendo el correspondiente sistema (8.7.9)

```
(%i27) define(alphanew(dx,dy),(18*eta*((dy^2+dx^2)^(1/4)*  
sqrt((d*rhoa)/eta))/delta0+1)^2)/(d^2*rho));  
define(fy(t,y,dy,x,dx), -g-alphanew(dx,dy)*dy);  
define(fx(t,y,dy,x,dx), -alphanew(dx,dy)*dx);
```

y las condiciones iniciales

```
(%i32) h:20$ v0y:0$ v0x:10$
```

A continuación resolvemos el sistema

```
(%i38) sol:rk( [dy,fy(t,y,dy,x,dx),dx, fx(t,y,dy,x,dx)], [y,dy,x,dx] ,
               [h,v0y,0,v0x],[t,0,5,0.01])$  

      length(sol);  

      ti:makelist(sol[k][1],k,1,length(sol))$  

      yy:makelist(sol[k][2],k,1,length(sol))$  

      xx:makelist(sol[k][4],k,1,length(sol))$  

      last(yy);  

(%o34) 501  

(%o38) -29.87585724524115
```

Representamos la solución de la $y(t)$

```
(%i39) wxplot2d([discrete,ti,yy],[xlabel,"t"], [ylabel,"y(t)])$
```

de donde se ve (ver también la entrada (%o38)) que necesariamente hay un cambio de signo (por tanto un cero). A continuación encontramos la posición y el valor del mismo

```
(%i41) pc1:poscero(yy);  

      (pc1) 259  

      tc1:newcero(ti,yy,pc1);  

      (tc1) 2.576044214168774
```

y definimos la solución numérica de nuestro problema

```
(%i43) nx:rest(xx,pc1-length(yy))$  

      ny:rest(yy,pc1-length(yy))$  

      lx:last(nx);  

      (lx) 12.87648490726287
```

De lo anterior se sigue que el tiempo de caída es 2,56 segundos y alcanza 12,88 metros, es decir, tarda más en caer pero tiene un menor alcance. Con los datos podemos dibujar la trayectoria de nuestra esfera dibujando los puntos $(x(t), y(t))$ (ver figura 8.19, derecha)

```
(%i44) wxdraw2d(grid=true,line_type=solid,line_width=5,nticks=100,
                 xaxis=true,xaxis_type=solid,yaxis=true,
                 yaxis_type=solid,xlabel="x(t)",ylabel="y(t)",
                 point_type=filled_circle,point_size =1,
                 color=blue,title="Trayectoria",
                 yrange=[0,h+.2],xrange=[0,lx+.2],points(nx,ny));
```

y lo podemos comparar con los resultados de la caída libre sin rozamiento (ver figura 8.20 (izquierda))

```
(%i45) wxdraw2d(grid=true,line_type=solid,line_width=5,nticks=100,
```

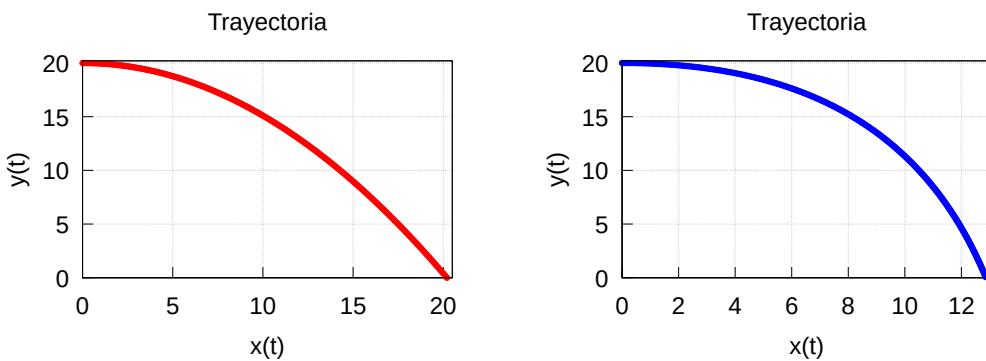


Figura 8.19: Trayectoria de la esfera sin (izquierda) y con (derecha) rozamiento del aire.

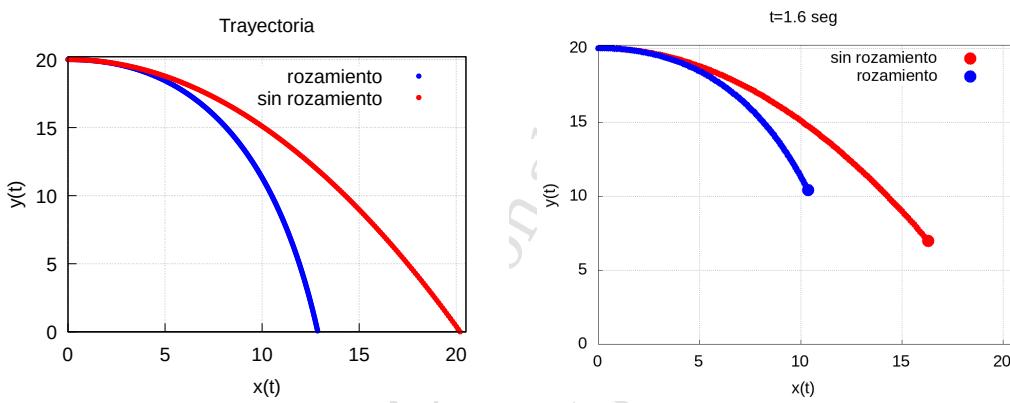


Figura 8.20: Comparación de la caída de la esfera con y sin rozamiento (izquierda) y fotograma de la película con la animación de la caída (derecha).

```
xaxis =true, xaxis_type=solid,yaxis=true,
xlabel="x(t)",ylabel="y(t)",point_type=filled_circle,
title="Trayectoria",yrange=[0,h+.2],xrange=[0,lx1+.2],
point_size=1,color=blue,key="rozamiento",points(nx,ny),
point_size=1,color=red,key="caída libre",points(nxl,nyl));
```

Finalmente, haremos una película de la caída de la gota con y sin rozamiento como una imagen gif animada (un fotograma se puede ver en la figura 8.20 (derecha))

```
(%i46) pt:float(floor(ti*10.0)/10)$
(%i47) draw(terminal=animated_gif,delay=1,file_name="peli-caida-comp",
makelist(gr2d( line_width = 3,dimensions = [1000,800],
xaxis=true,xaxis_type=solid,yaxis=true,xlabel="x(t)",ylabel="y(t)",
grid=true,key_pos=top_left,font ="Arial",font_size=24,
point_type=filled_circle,yrange=[0,h+.2],xrange=[0,lx1+.1],
point_size=2,color=red, points(makelist([xxl[ii],yyl[ii]],ii,1,kk)),
point_size=2,color=blue,points(makelist([nx[ii],ny[ii]],ii,1,kk)),
point_size = 4, color = red, key_pos = top_right,
key="sin rozamiento", points([[xxl[kk],yyl[kk]]]),
point_size=4,color=blue,key="rozamiento",points([[nx[kk],ny[kk]]]),
title=concat("t=",pt[kk]," seg")),kk,pc1))$
```

Hemos usado la secuencia `floor(ti*10.0)/10` para redondear hasta una cifra decimal el valor de la variable `ti`. Si queremos dos cifras habrá que multiplicar por 100 y dividir por 100, etc. Hay situaciones en las que MAXIMA imprime en pantalla en vez del número redondeado hasta la cantidad de cifras pero seguido de muchos ceros y terminado en uno. Por ejemplo, puede ocurrir que tengamos el número 552,5527666856816 y que si usamos la secuencia anterior para aproximarla con 2 cifras obtengamos el valor 552,560000000000001. Eso ocurre por diversas causas y tiene una solución muy sencilla que consiste en controlar la precisión de las salidas (`output`). Eso lo hace la variable `fpprintprec` que por defecto es 16 (recordemos que MAXIMA trabaja en doble precisión). En este caso basta definirla en 15, e.g. `fpprintprec:15` y se resuelve el problema.

8.8. Un ejemplo de cambio variables para funciones de dos variables.

Supongamos que tenemos una expresión del tipo¹¹

$$\Phi(x, y, z, z_x, z_y, z_{xx}, z_{xy}, z_{yy}, \dots) = 0$$

donde x e y son variables independientes y z es una función $z : \mathbb{R}^2 \mapsto \mathbb{R}$, $z = z(x, y)$ y queremos escribirlas en las nuevas variables u , v y $w = w(u, v)$ asumiendo que las variables nuevas y viejas se relacionan mediante el sistema

$$g_i(x, y, z, u, v, w) = 0, \quad i = 1, 2, 3,$$

que denominaremos expresiones del cambio de variables, donde las funciones g_i , $i = 1, 2, 3$ se asumen diferenciables tantas veces como haga falta.

Nos centraremos en el caso cuando se tiene la expresión explícita de las variables viejas en función de las nuevas

$$x = f_1(u, v, w), \quad y = f_2(u, v, w), \quad z = f_3(u, v, w). \quad (8.8.1)$$

Diferenciando (8.8.1) tenemos

$$\begin{aligned} dx &= D_u f_1 du + D_v f_1 dv + D_w f_1 dw, \\ dy &= D_u f_2 du + D_v f_2 dv + D_w f_2 dw, \\ dz &= D_u f_3 du + D_v f_3 dv + D_w f_3 dw, \end{aligned} \quad (8.8.2)$$

donde D_u , D_v y D_w son las correspondientes derivadas parciales respecto a las variables u , v y w , respectivamente. Si usamos que $dw = D_u w du + D_v w dv$ tenemos

$$\begin{aligned} dx &= \mathcal{D}_u f_1 du + \mathcal{D}_v f_1 dv, \\ dy &= \mathcal{D}_u f_2 du + \mathcal{D}_v f_2 dv, \\ dz &= \mathcal{D}_u f_3 du + \mathcal{D}_v f_3 dv \end{aligned} \quad (8.8.3)$$

donde

$$\mathcal{D}_u = \frac{\partial}{\partial u} + \frac{\partial w}{\partial u} \frac{\partial}{\partial w} = \frac{\partial}{\partial u} + w_u \frac{\partial}{\partial w}, \quad \mathcal{D}_v = \frac{\partial}{\partial v} + \frac{\partial w}{\partial v} \frac{\partial}{\partial w} = \frac{\partial}{\partial v} + w_v \frac{\partial}{\partial w}.$$

¹¹Por simplicidad describiremos el caso de dos variables independientes. El caso general es análogo.

Si el determinante

$$\Delta = \begin{vmatrix} \mathcal{D}_u f_1 & \mathcal{D}_v f_1 \\ \mathcal{D}_u f_2 & \mathcal{D}_v f_2 \end{vmatrix} \neq 0,$$

entonces las dos primeras ecuaciones de (8.8.3) se pueden resolver expresándose las diferenciales du y dv en función de las dx y dy

$$\begin{aligned} du &= \frac{1}{\Delta} \left(\mathcal{D}_v f_2 dx - \mathcal{D}_v f_1 dy \right), \\ dv &= \frac{1}{\Delta} \left(-\mathcal{D}_u f_2 dx + \mathcal{D}_u f_1 dy \right), \end{aligned} \quad (8.8.4)$$

que sustituimos en la tercera expresión de (8.8.3) obteniendo

$$dz = \frac{1}{\Delta} \left(\mathcal{D}_u f_3 D_v f_2 - D_v f_3 D_u f_2 \right) dx + \frac{1}{\Delta} \left(-\mathcal{D}_u f_3 D_v f_1 + D_v f_3 D_u f_1 \right) dy,$$

de donde deducimos

$$\begin{aligned} z_x &= \frac{1}{\Delta} \left(\mathcal{D}_u f_3 D_v f_2 - D_v f_3 D_u f_2 \right) = F_1(u, v, w, w_u, w_v), \\ z_y &= \frac{1}{\Delta} \left(-\mathcal{D}_u f_3 D_v f_1 + D_v f_3 D_u f_1 \right) = F_2(u, v, w, w_u, w_v). \end{aligned} \quad (8.8.5)$$

Si queremos obtener las expresiones de las segundas derivadas prodecemos como sigue:
Calculamos

$$d(z_x) = z_{xx} dx + z_{xy} dy = D_u F_1 du + D_v F_1 dv + D_w F_1 dw + D_{w_u} F_1 dw_u + D_{w_v} F_1 dw_v.$$

A continuación sustituimos en la parte derecha los valores de las diferenciales nuevas

$$dw = w_u du + w_v dv, \quad dw_u = w_{uu} du + w_{vu} dv, \quad dw_v = w_{uv} du + w_{vv} dv$$

y en la expresión resultante sustituimos los valores de las diferenciales du y dv obtenidos en (8.8.4). Esto nos da una expresión de $d(z_x)$ en función de las diferenciales antiguas. Igualando las expresiones delante de las diferenciales dx y dy obtenemos los valores z_{xx} y z_{xy} respectivamente. Para obtener z_{yy} se procede de forma análoga pero partiendo de la segunda ecuación de (8.8.5).

La idea es realizar los cálculos aquí descritos con MAXIMA. Por comodidad usaremos para las funciones las variables z y w , pero hemos de decirle a MAXIMA que ambas son funciones de x, y y u, v , respectivamente. Eso se hace, como vimos en la página 36, con el comando `depends`. Otro comando que nos interesará es el de sustitución que ya hemos usado anteriormente y que usaremos en la forma `subst(var=varnew, expr)` que sustituye la variable `var` por la variable nueva `varnew` en la expresión `expr`.

Consideremos el siguiente ejemplo: Escribir el laplaciano de orden dos $z_{xx} + z_{yy} = 0$ en las nuevas variables $x = r \cos(\phi)$, $y = r \sin(\phi)$, $z = w$.

Así, definimos nuestras nuevas variables y calculamos los diferenciales (8.8.2)

```
(%i1) depends(w,[r,phi]); depends(z,[x,y]);
(%o1) [w(r,phi)]
(%o2) [z(x,y)]
(%i3) x=r*cos(phi);eq1:diff(%);
```

```
(%o3) x=cos(phi)*r
(%o4) del(x)=cos(phi)*del(r)-sin(phi)*r*del(phi)
(%i5) y=r*sin(phi);eq2:diff(%);
(%o5) y=sin(phi)*r
(%o6) del(y)=sin(phi)*del(r)+cos(phi)*r*del(phi)
(%i7) z=w;eq3:diff(%);
(%o7) z=w
(%o8) ('diff(z,y,1))*del(y)+('diff(z,x,1))*del(x)=
('diff(w,r,1))*del(r)+('diff(w,phi,1))*del(phi)
```

Las ecuaciones correspondientes a (8.8.4) en este caso son

```
(%i9) linsolve([eq1,eq2],[del(r),del(phi)])$  
sol1:trigsimp(%);  
(%o10) [del(r)=sin(phi)*del(y)+cos(phi)*del(x),  
del(phi)=cos(phi)*del(y)-sin(phi)*del(x))/r]
```

que sustituimos en la expresión para el diferencial de z

```
(%i11) subst(sol1,eq3)$ expand(%)$ a:second(%);  
(%o13) sin(phi)*('diff(w,r,1))*del(y)+(cos(phi)*('diff(w,phi,1))*del(y))/r  
+cos(phi)*('diff(w,r,1))*del(x)-(sin(phi)*('diff(w,phi,1))*del(x))/r
```

A continuación seleccionamos los coeficientes delante de ambos diferenciales que nos dan los valores de las derivadas z_x y z_y , respectivamente

```
(%i14) zx:coeff(a,del(x),1);  
(%o14) cos(phi)*('diff(w,r,1))-(sin(phi)*('diff(w,phi,1)))/r  
(%i15) zy:coeff(a,del(y),1);  
(%o15) sin(phi)*('diff(w,r,1))+cos(phi)*('diff(w,phi,1))/r
```

Para calcular las segundas derivadas calculamos el diferencial de $dz_x = z_{xx}dx + z_{xy}dy$

```
(%i16) eqdzx:diff(zx);
```

y en el resultado sustituimos los diferenciales dr y $d\phi$ por los antiguos dx y dy

```
(%i17) subst(sol1,eqdzx)$  
b:expand(%);
```

Finalmente, identificamos el coeficiente delante del diferencial dx que corresponde a la derivada z_{xx}

```
(%i19) zxx:coeff(b,del(x),1);  
(%o19) cos(phi)^2*('diff(w,r,2))+(sin(phi)^2*('diff(w,r,1)))/r+  
      (sin(phi)^2*('diff(w,phi,2)))/r^2  
      -(2*cos(phi)*sin(phi)*('diff(w,phi,1,r,1)))/r  
      +(2*cos(phi)*sin(phi)*('diff(w,phi,1)))/r^2
```

Repetimos el proceso con la expresión de z_y

```
(%i20) eqdzy:diff(zy)$
(%i21) subst(sol1,eqdzy)$ c:expand(%);
(%i23) zyy:coeff(c,del(y),1);
(%o23) sin(phi)^2*(diff(w,r,2))+(cos(phi)^2*(diff(w,r,1)))/r+
(cos(phi)^2*(diff(w,phi,2)))/r^2
+(2*cos(phi)*sin(phi)*(diff(w,phi,1,r,1)))/r
-(2*cos(phi)*sin(phi)*(diff(w,phi,1)))/r^2
```

Para terminar calculamos $z_{xx} + z_{yy}$

```
(%i24) zxx+zyy$
(%i25) trigsimp(%); expand(%);
(%o26) 'diff(w,r,2)+'diff(w,r,1)/r+'diff(w,phi,2)/r^2
(%i27) subst(w=z,%);
(%o27) 'diff(z,r,2)+'diff(z,r,1)/r+'diff(z,phi,2)/r^2
```

que nos da

$$\frac{d^2z}{dr^2} + \frac{1}{r} \frac{dz}{dr} + \frac{1}{r^2} \frac{d^2z}{d\varphi^2} = 0.$$

Como ejercicio se propone encontrar las expresiones en las nuevas variables de las siguientes ecuaciones aplicando los correspondientes cambios de variables:

1. $z_{xx} - z_{yy} = 0$ con $x = u + v$, $y = u - v$, $z = w$;
2. $z_{xx} - z_y = 0$ con $x = -u/v$, $y = -1/v$, $z = \sqrt{-v}e^{u^2/(4v)}w$.

Un reto interesante para el lector puede ser programar con MAXIMA el caso de funciones de tres variables. Como ejemplo proponemos escribir la ecuación de Laplace $w_{xx} + w_{yy} + w_{zz} = 0$ en coordenadas esféricas: $x = r \cos(\phi) \sin(\theta)$, $y = r \sin(\phi) \sin(\theta)$, $z = r \cos(\theta)$, siendo la función $w(x, y, z) = w(r, \theta, \phi)$.

8.9. Calculando diferenciales de funciones escalares de varias variables.

Aunque el MAXIMA tiene la orden `taylor` que mencionamos al final del apartado 2.3 vamos a programar una función que calcule el diferencial $D^k f(a)$ de cualquier orden de una función escalar de dos variables en cierto punto a y que actúe sobre cierto vector h . La idea es mostrar al lector como resolver un problema paso a paso, y particularmente como corregir los distintos problemas que van a ir apareciendo. Vamos a usar la siguiente fórmula [7, pág. 463] para el k -ésimo diferencial de una función f :

$$D^k f(a)(h) = \sum_{i_1=1}^k \cdots \sum_{i_k=1}^k \frac{\partial^k f(a)}{\partial x_{i_1} \cdots \partial x_{i_k}} h_{i_1} \cdots h_{i_k} = \left(h_1 \frac{\partial}{\partial x_1} + \cdots + h_n \frac{\partial}{\partial x_n} \right)^k f(a), \quad (8.9.1)$$

donde estamos usando la notación $D^k f(a)(h) := D^k f(a)(h, h, \dots, h)$ (debemos recordar que el diferencial $D^k f(a)$ es una aplicación k -lineal). La expresión anterior en el caso de

funciones de dos variables se reduce a la siguiente:

$$\begin{aligned} D^k f(x_0, y_0)(h) &= \sum_{\substack{i, j = 0 \\ i+j = k}}^k \frac{\partial^k f(x_0, y_0)}{\partial^i x \partial^j y} (x - x_0)^i (y - y_0)^j \\ &= \sum_{i=0}^k \binom{k}{i} \frac{\partial^k f(x_0, y_0)}{\partial^i x \partial^{k-i} y} (x - x_0)^i (y - y_0)^{k-i}, \quad h = (h_x, h_y) = (x - x_0, y - y_0). \end{aligned}$$

Como punto de partida usaremos los diferenciales de orden 1 y 2, cuyas expresiones son, respectivamente,

$$\begin{aligned} Df(x_0, y_0)(x, y) &= \frac{\partial f(x_0, y_0)}{\partial x} (x - x_0) + \frac{\partial f(x_0, y_0)}{\partial y} (y - y_0), \\ D^2 f(x_0, y_0)(x, y) &= \frac{\partial^2 f(x_0, y_0)}{\partial x^2} (x - x_0)^2 + 2 \frac{\partial^2 f(x_0, y_0)}{\partial x \partial y} (x - x_0)(y - y_0) + \frac{\partial^2 f(x_0, y_0)}{\partial y^2} (y - y_0)^2. \end{aligned}$$

Lo primero que tenemos que conseguir es implementar las derivadas $\frac{\partial^{i+j} f(x, y)}{\partial x^i \partial y^j}$ evaluadas en un punto dado (x_0, y_0) :

```
(%i1) dd:diff(f(x,y),x,i,y,j);
(%o1) 'diff(f(x,y),x,i,y,j)
(%i2) ev(dd,x=0,y=0);
diff: variable must not be a number; found: 0
-- an error. To debug this try: debugmode(true);
(%i3) dd:diff(f(x,y),x,i,y,j); ev(dd,x=x0,y=y0);
(%o3) 'diff(f(x,y),x,i,y,j)
(%o4) 'diff(f(x0,y0),x0,i,y0,j)
```

Esta primera secuencia nos muestra que la evaluación de la derivada de forma genérica no es trivial pues en ambos casos MAXIMA intenta sustituir la variable por su valor. En el caso del cero nos da un error obvio, y en el caso de una valor genérico x_0 nos cambia todas las variables por los nuevos valores como si estos fueran variables nuevas. Lo anterior nos induce a pensar que quizás actuando sobre una función concreta vamos a obtener el resultado deseado, lo cual se confirma con la secuencia siguiente:

```
(%i5) define(f(x,y),exp(x)*sin(y)+y^2*sin(x));
dd:diff(f(x,y),x,2,y,2);
ev(dd,x=0,y=0);
(%o5) f(x,y):=%e^x*sin(y)+sin(x)*y^2
(%o6) -%e^x*sin(y)-2*sin(x)
(%o7) 0
(%i8) ev(dd,x=x0,y=y0);
(%o8) -%e^x0*sin(y0)-2*sin(x0)
```

Ahora, si nos interesa definir una función para el cálculo de la derivada evaluada en un punto, conviene poder obtener el valor $\frac{\partial^{i+j} f(x_0, y_0)}{\partial x^i \partial y^j}$ en una sola línea.

Para eso utilizaremos la orden `block` que ya vimos antes en la página 57. Así tenemos

```
(%i9) block([dd], dd:diff(f(xx,yy),xx,1,yy,1),
           ev(dd,xx=x0,yy=y0));
(%o9) %e^x0*cos(y0)+2*cos(x0)*y0
```

Vamos a hacer el primer intento de obtener el diferencial de orden k de una función de dos variables. Usaremos la función $f(x, y)$ definida anteriormente como ejemplo, pero antes calcularemos los diferenciales de orden 1 y 2 a partir del jacobiano y la matriz hessiana correspondientes, los cuales nos servirán como test

```
(%i10) jacobian([f(x,y)], [x,y]); ev(%, x=0, y=0); %. [x,y];
(%o10) matrix([%e^x*sin(y)+cos(x)*y^2, %e^x*cos(y)+2*sin(x)*y])
(%o11) matrix([0,1])
(%o12) y
(%i13) hessian(f(x,y), [x,y]); ev(%, x=0, y=0); [x,y] .%. [x,y];
(%o13) matrix([%e^x*sin(y)-sin(x)*y^2, %e^x*cos(y)+2*cos(x)*y],
[ %e^x*cos(y)+2*cos(x)*y, 2*sin(x)-%e^x*sin(y)])
(%o14) matrix([0,1],[1,0])
(%o15) 2*x*y
```

Está claro que el diferencial de orden k será una suma que contendrá todas las posibles derivadas de orden k de nuestra función. Ello nos obliga a usar un bucle para calcular todas las posibles derivadas. Para ello usaremos el comando `for` (ver página 78):

```
(%i16) k:1$
      for j:0 thru k do
        for i:0 thru k do
          if i+j = k then
            block([dd], dd:diff(f(xx,yy),xx,i,yy,j),
                  b[i,j]:=ev(dd,xx=0,yy=0)*x^i*y^j)
          else b[i,j]:=0 $
```

Así, en la orden anterior tenemos un bucle en la variable j que toma los valores de $j = 0, 1, \dots, k$ y para cada uno de ellos realiza un bucle interior en la variable i que toma los valores de $i = 0, 1, \dots, k$ y que realiza la acción del cálculo de la derivada. Ahora bien, como sabemos, para la diferencial de orden k sólo nos interesan los valores de i y j tales que $i + j = k$, es por ello que al realizar el segundo bucle la acción sólo debe darnos los valores de la derivada que nos interesan. Por ello usamos la orden `if` que ya usamos cuando definimos funciones a trozos en el apartado 2.3. El resultado es una cantidad $b_{i,j}$ que toma los valores de las derivadas parciales de orden k si $i + j = k$ o cero en caso contrario. Con esos valores podemos fácilmente calcular la diferencial de orden 1

```
(%i18) sum(sum(binomial(k,i)*b[i,j], i , 0, k) , j , 0, k);
(%o18) y
```

de orden 2,

```
(%i19) k:2$
      for j:0 thru k do for i:0 thru k do if i+j = k then
        block([dd], dd:diff(f(xx,yy),xx,i,yy,j),
              b[i,j]:=ev(dd,xx=0,yy=0)*x^i*y^j) else b[i,j]:=0$
```

$$\sum_{i+j=k} \text{binomial}(k,i) b_{i,j}$$

```
      sum(sum(binomial(k,i)*b[i,j], i , 0, k) , j , 0, k);
(%o21) 2*x*y
```

y así sucesivamente. Conviene observar que los diferenciales de orden 1 y 2 coinciden con los calculados anteriormente usando el jacobiano y la matriz hessiana.

El próximo paso es definir una función para calcular el diferencial de orden arbitrario, para lo que usaremos la fórmula (8.9.1):

```
(%i22) Diff(k):=block( for j:0 thru k do for i:0 thru k do if i+j=k then
                         block([dd], dd:diff(f(xx,yy),xx,i,yy,j),
                               b[i,j]:ev(dd,xx=0,yy=0)*(x)^i*(y)^j) else b[i,j]:=0,
                         sum(sum(binomial(k,i)*b[i,j], i , 0, k) , j,0,k))$
(%i23) Diff(0);
(%o23) 0
(%i24) Diff(1);
(%o24) y
(%i25) Diff(2);
(%o25) 2*x*y
```

Ya puestos, definimos una función que dependa de los valores donde vamos a evaluar (x_0, y_0)

```
(%i26) Diff1(k,x0,y0):=block( for j:0 thru k do for i:0 thru k do
                                 if i+j=k then
                                   block([dd], dd:diff(f(xx,yy),xx,i,yy,j),
                                         b[i,j]:ev(dd,xx=x0,yy=y0)*(x-x0)^i*(y-y0)^j)
                                   else b[i,j]:=0,
                                         sum(sum(binomial(k,i)*b[i,j], i , 0, k) , j,0,k))$
(%i27) Diff1(2,x0,y0);
(%o27) (x-x0)^2*(%e^x0*sin(y0)-sin(x0)*y0^2)+(y-y0)^2*(2*sin(x0)-
           %e^x0*sin(y0))+2*(x-x0)*(y-y0)*(%e^x0*cos(y0)+2*cos(x0)*y0)
(%i28) Diff1(2,alpha,eta);
(%o28) (2*sin(alpha)-%e^alpha*sin(eta))*(y-eta)^2+2*(%e^alpha*cos(eta)-
           2*cos(alpha)*eta)*(x-alpha)*(y-eta)+(%e^alpha*sin(eta)-
           sin(alpha)*eta^2)*(x-alpha)^2
(%i29) Diff1(0,0,%pi); Diff1(1,0,%pi); Diff1(2,0,%pi);
(%o29) 0
(%o30) -y+%pi^2*x+%
```

π

$(%o31) 2*(2*\pi-1)*x*(y-\pi)$

$(%i32) \text{Diff1}(0,0,0);\text{Diff1}(1,0,0);\text{Diff1}(2,0,0);$

$(%o32) 0$

$(%o33) y$

$(%o34) 2*x*y$

Si tenemos los diferenciales podemos encontrar en polinomio de Taylor de orden m

$$P_m[f](x_0, y_0) = f(x_0, y_0) + \sum_{l=1}^m \frac{1}{l!} D^l f(x_0, y_0)(x - x_0, y - y_0).$$

```
(%i35) TAY(x,y,x0,y0,m):=expand( sum((1/l!)* Diff1(l,x0,y0),l,0,m));
(%o35) TAY(x,y,x0,y0,m):=expand(sum(1/l!*Diff1(l,x0,y0),l=0,0,m))
(%i36) ta:TAY(x,y,0,%pi,2);
```

```
(%o36) 2*%pi*x*y-x*y-y-%pi^2*x+%pi*x+%pi
(%i37) taM:expand(taylor(exp(x)*sin(y)+y^2*sin(x),[x,y],[0,%pi],2));
(%o37) 2*%pi*x*y-x*y-y-%pi^2*x+%pi*x+%pi
(%i38) ta-taM;
(%o38) 0
```

Las últimas entradas son para comparar la salida de nuestra orden TAY con la salida de la orden taylor de MAXIMA las cuales, como se ve, coinciden.

Vamos ahora a mejorar nuestra función Diff1. Vamos a intentar que nuestra orden funcione sin tener que definir previamente la función de trabajo. Ante todo limpiamos la memoria con un kill(all) y comenzamos

```
(%i1) define(f(x,y),exp(x)*sin(y)+y^2*sin(x));
(%o1) f(x,y):=%e^x*sin(y)+sin(x)*y^2
(%i2) Diff1(k,x0,y0):=block( for j:0 thru k do for i:0 thru k do
                                if i+j=k then block([dd],
                                dd:diff(f(xx,yy),xx,i,yy,j),
                                b[i,j]:ev(dd,xx=x0,yy=y0)*(x-x0)^i*(y-y0)^j)
                                else b[i,j]:0,
                                sum(sum(binomial(k,i)*b[i,j],i,0,k),j,0,k));
(%i3) d1:Diff1(1,x0,y0);
(%o3) (x-x0)*(%e^x0*sin(y0)+cos(x0)*y0^2)+(y-y0)*(%e^x0*cos(y0)
+2*sin(x0)*y0)
(%i4) d10:Diff1(1,0,0);
(%o4) y
```

Definamos ahora la función siguiente:

```
(%i5) Diferencial(expr,x,y,x0,y0,k):=block([b],
      for j:0 thru k do for i:0 thru k do if i+j=k then
      block([dd], dd:diff(expr,x,i,y,j),
      b[i,j]:ev(dd,x=x0,y=y0)*(x-x0)^i*(y-y0)^j) else b[i,j]:0,
      sum(sum(binomial(k,i)*b[i,j],i,0,k),j,0,k));
(%i6) Diferencial(f(x,y),x,y,x0,y0,1);
(%o6) (x-x0)*(%e^x0*sin(y0)+cos(x0)*y0^2)+(y-y0)*(%e^x0*cos(y0)
+2*sin(x0)*y0)
(%i7) %-d1;
(%o7) 0
(%i8) Diferencial(cos(x)+sin(y),x,y,x0,y0,1);
(%o8) (y-y0)*cos(y0)-(x-x0)*sin(x0)
```

Todo parece en orden, pero si cambiamos las variables la cosa no funciona

```
(%i9) Diferencial(f(x,y),x,y,0,0,1);
(%o9) y
(%i10) Diferencial(f(xx,yy),xx,yy,0,0,1);
(%o10) xx*(%e^xx*sin(yy)+cos(xx)*yy^2)+yy*(%e^xx*cos(yy)+2*sin(xx)*yy)
```

Para resolver esto usamos nuevas variables internas y el comando subst

```
(%i11) Diferencial(expr,u,v,u0,v0,k):=
        block([b,dd,xxx,yyy,ee,newexpr,newexpr1],
        newexpr:subst(xxx,u,expr),
        newexpr1:subst(yyy,v,newexpr),
        for j:0 thru k do
        for i:0 thru k do
        if i+j = k then
        block([dd], dd:diff(newexpr1,xxx,i,yyy,j),
        b[i,j]:ev(dd,xxx=u0,yyy=v0)*(u-u0)^i*(v-v0)^j)
        else b[i,j]:0,
        sum(sum(binomial(k,i)*b[i,j],i,0,k),j,0,k))$
```

A continuación comprobamos que funciona

```
(%i12) Diferencial(f(x,y),x,y,0,0,4);
(%o12) 4*x^3*y-4*x*y^3
(%i13) Diferencial(f(xx,yy),xx,yy,0,0,4);
(%o13) 4*xx^3*yy-4*xx*yy^3
```

Ya estamos listos para definir nuestra propia orden para calcular el polinomio de Taylor de orden k de una función de dos variables expr en las variables u y v en el punto u_0 y v_0

```
(%i14) TAYLOR(expr,u,v,u0,v0,k):=block(
        expand(sum((1/l!)*Diferencial(expr,u,v,u0,v0,l),l,0,k)))$
```

Que podemos comparar con la salida del comando `taylor` de MAXIMA

```
(%i15) TAYLOR(f(x,y),x,y,x0,y0,2);
(%i16) expand(taylor(f(x,y),[x,y],[x0,y0],2)-%);
(%o16) 0
(%i17) TAYLOR(f(x,y),x,y,0,0,2);
(%o17) x*y+y
(%i18) taylor(f(x,y),[x,y],[0,0],2);
(%o18)/T/ y*x+y+...
(%i19) TAYLOR(cos(x^2+y),x,y,%pi,0,6);
(%i20) expand(taylor(cos(x^2+y),[x,y],[%pi,0],6)-%);
(%o20) 0
```

Como ejercicio al lector se propone adaptar las funciones anteriores al caso de funciones de tres variables.

8.10. Ajuste de mínimos cuadrados con pesos.

En el apartado 4.3 hemos discutido como podemos realizar un ajuste de mínimos cuadrados de un conjunto de datos. Nuestro objetivo es ahora modificar los correspondientes comandos de MAXIMA para el caso cuando tenemos que ajustar los datos pero teniendo en cuenta los correspondientes pesos de cada uno de ellos.

Por ejemplo, sean $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ correspondientes a ciertas mediciones

y supongamos que tenemos cierta función $f_{\text{fit}} := f(x, \alpha_1, \dots, \alpha_m)$ que depende de los parámetros $(\alpha_1, \dots, \alpha_m)$ y queremos encontrar dichos parámetros de forma que la expresión

$$\sum_{i=1}^N (y_k - f_{\text{fit}}(x_k))^2$$

sea mínima. Este es el típico problema de mínimos cuadrados que resuelve MAXIMA con los comandos `lsquares_estimates_approximate` o `lsquares_estimates`. Pero cambiemos el problema. Imaginemos que ahora la expresión a minimizar es algo más complicada pues cada valor (x_k, y_k) tiene un cierto peso, no igual para cada uno, digamos ρ_k . Está claro que lo que hay que minimizar es la cantidad

$$\sum_{i=1}^N \rho_k (y_k - f_{\text{fit}}(x_k))^2$$

¿Cómo proceder? Está claro que con la orden `lsquares_estimates` no hay mucho que hacer pues su sintaxis, recordemos, es

```
lsquares_estimates(datos, lista de variables, ecuacion, lista de parametros)
```

donde `datos` es una matriz de datos, `lista de variables`, es una lista con el nombre de las variables correspondiente a cada columna de la matriz de datos, `lista de parametros` es una lista con los parámetros del modelo y `ecuacion` es la ecuación que vamos usar para el ajuste de datos. Sin embargo, teníamos la opción numérica que es más flexible. Recuperando el ejemplo de la página 85 tenemos, por un lado, el comando `lsquares_mse(datos, var, ec)` que genera la expresión del error cuadrático medio de los datos de nuestra matriz de datos para la ecuación `ec` en la lista de variables `var`. Por ejemplo, supongamos que tenemos una matriz `mm` de dos columnas y N filas y queremos ajustar los datos con la recta $y = ax + b$. Entonces `lsquares_mse` genera la siguiente salida

```
(%i1) mse : lsquares_mse (mm, [x, y], y=a*x+b);
(%i2) sum((mm[i,2]-a*mm[i,1]-b)^2, i=1,1,N)/N
```

A continuación usamos

```
lsquares_estimates_approximate(mse, [a,b])
```

que lo que hace es encontrar los valores de a y b que minimicen la expresión (%i2).

Visto esto está claro lo que podemos hacer: construir la suma `mse` correspondiente, por ejemplo de la siguiente forma

```
mse:sum( rho[k]*(pp(mm[i,1])-mm[i,2])^2, i, 1, length(mm))/length(mm);
```

donde `rho[k]` son los valores de los pesos.

En el caso cuando se tiene una lista de mediciones de la forma (x_k, y_k, e_k) donde e_k es el error del valor x_k , se suele tomar como peso el valor $\rho_k = 1/e_k$. Veamos un ejemplo de aplicación de lo anterior para un caso de ajuste con y sin pesos.

Supongamos que tenemos la matriz de datos

$$\begin{pmatrix} 1 & 2 & 3 & 5 & 7 & 10 \\ 109 & 149 & 149 & 191 & 213 & 224 \\ 20 & 20 & 4 & 4 & 4 & 4 \end{pmatrix}$$

donde la primera fila le corresponde a las x , la segunda a las y y la tercera a los errores de y . Vamos a hacer un ajuste con la función $f(x) = a(1 - \exp(-bx))$.

```
(%i1) load (lsquares)$ load(descriptive)$ load(draw)$
(%i4) xx:[1, 2, 3, 5, 7, 10];
      yy:[109, 149, 149, 191, 213, 224];
      ee:[20,20,4,4,4,4];
      mm:transpose(matrix(xx,yy))$ 
(%o4) [1,2,3,5,7,10]
(%o5) [109,149,149,191,213,224]
(%o6) [20,20,4,4,4,4]
```

Definimos la función de ajuste y usamos `lsquares_mse` para preparar la expresión a minimizar y luego `lsquares_estimates_approximate` para minimizar.

```
(%i8) pp(x):=a*(1-exp(-b*x));
(%i9) mse : lsquares_mse (mm, [x, y], y=pp(x));
      sol:lsquares_estimates_approximate(mse, [a, b]);
(%o9) sum((mm[i,2]-a*(1-%e^(-b*mm[i,1])))^2,i=1,1,6)/6
(%o10) [[a=213.8094059887529,b=0.547237523991729]]
```

Con los valores obtenidos definimos la función de ajuste y calculamos el coeficiente de correlación

```
(%i11) for k:1 thru length(sol[1]) do c[k-1]:float( second(sol[1][k]))$ 
      define(fit(x),c[0]*(1-exp((-c[1])*x)));
(%i13) SSE:sum( (yy[k]-fit((xx[k])))^2 , k,1,length(mm))$ 
      SST:sum( (yy[k]-mean(yy))^2 , k,1,length(mm))$ 
      float(corre:sqrt(1-SSE/SST));
(%o15) 0.9383324579333591
```

Resolvamos el problema sin usar `lsquares_mse` para poder controlar la expresión a minimizar. Para ello definimos la suma

$$\sum_{i=1}^N (y_k - f_{\text{fit}}(x_k))^2,$$

y usamos `lsquares_estimates_approximate` para minimizarla:

```
(%i16) ss:sum( (pp(mm[i,1])-mm[i,2])^2, i, 1, length(xx))/length(xx)$ 
      sol1:lsquares_estimates_approximate(ss, [a,b]);
(%o17) [[a=213.8094059887529,b=0.5472375239917291]]
(%i18) sol-sol1;
(%o18) [[0=0.0,0=-1.110223024625156*10^-16]]
```

Como vemos, el resultado es esencialmente el mismo. Si queremos tener en cuenta los pesos entonces habrá que definir una nueva suma $\sum_{i=1}^N \rho_k (y_k - f_{\text{fit}}(x_k))^2$, donde ρ_k son los pesos que, como ya dijimos, escogeremos inversamente proporcionales a los errores. Así, tenemos

```
(%i19) ssw:sum((1/ee[i])*(pp(mm[i,1])-mm[i,2])^2,
                  i,1,length(mm))/length(mm);
      solw:lsquares_estimates_approximate(ssw, [a,b]);
(%o20) [[a=225.1712692206121, b=0.4007843817891243]]
```

Definimos la función de ajuste y el coeficiente de correlación, que ahora varían ligeramente pues hay que tener en cuenta los pesos ρ_k

$$\text{SSE} = \sum_{i=1}^N \rho_k (y_k - f_{\text{fit}}(x_k))^2, \quad \text{SST} = \sum_{i=1}^N \rho_k (y_k - \bar{y})^2, \quad C_{\text{corr}} = \sqrt{1 - \frac{\text{SSE}}{\text{SST}}}.$$

```
(%i21) for k:1 thru length(solw[1]) do cw[k-1]:float(second(solw[1][k]))$;
      define(fitw(x),cw[0]*(1-exp((-cw[1])*x))) ;
(%i23) SSE:sum(1/ee[k]*(yy[k]-fitw((xx[k])))^2, k,1,length(mm))$;
      SST:sum(1/ee[k]*(yy[k]-mean(yy))^2, k,1,length(mm))$;
      float(correw:sqrt(1-SSE/SST));
(%o25) 0.9614815236898409
```

Finalmente, dibujamos los resultados

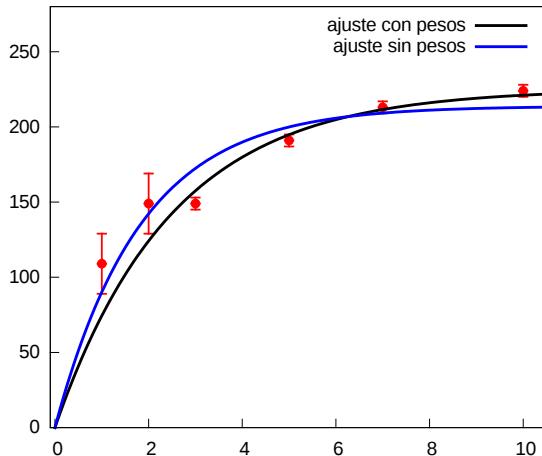


Figura 8.21: Gráfica del ajuste con y sin pesos.

```
(%i26) datos:makelist([xx[k],yy[k]],k,1,length(mm))$;
      er:makelist([xx[k],yy[k],ee[k]],k,1,length(mm))$;
      wxdraw2d(font = "Arial", font_size = 20, yrange = [0,270],
      point_size = 2,point_type = 7, color=red, points(datos),
      error_type = y, line_width=3, errors(er),
      key="ajuste con pesos",line_width=3,color=black,
      explicit(fitw(x),x,-.1,10.5),
      key="ajuste sin pesos",line_width =3,color=blue,
      explicit(fit(x),x,-.1,10.5),
      dimensions = [800,600]) $;
(%t28) << Graphics >>
```

Problema 8.10.0.1 Sea la matriz de datos

$$\begin{pmatrix} 0,9 & 3,9 & 0,3 & 1 & 1,8 & 9 & 7,9 & 4,9 & 2,3 & 4,7 \\ 6,1 & 6 & 2,8 & 2,2 & 2,4 & 1,7 & 8 & 3,9 & 2,6 & 8,4 \\ 9,5 & 9,7 & 6,6 & 5,9 & 7,2 & 7 & 10,4 & 9 & 7,4 & 10 \\ 0,4 & 0,4 & 0,2 & 0,4 & 0,1 & 0,3 & 0,1 & 0,2 & 0,2 & 0,2 \end{pmatrix}$$

donde la primera fila le corresponde a las x , la segunda a las y , la tercera a z y la cuarta a los errores de z . Encontrar los parámetros a, b, c de mejor ajuste de los datos a la función $z = f(x, y) = a \log(bx + cy)$ teniendo en cuenta los errores.

8.11. Un reto para el lector: la interpolación de Hermite.

Como hemos visto en (4.4.3), para la construcción del polinomio interpolador de Lagrange sólo usamos los valores $(x_i, f(x_i))$, no obstante, en gran cantidad de problemas reales además de los valores $y_i = f(x_i)$ se conocen los valores de las derivadas de la función $y'_i = f'(x_i)$. Así, tenemos en general el siguiente problema: Dado un intervalo (a, b) , sean $x_i, i = 1, 2, \dots, n$ ciertos valores prefijados del mismo y supongamos que a cada uno de ellos le corresponden la lista de valores $y_i, y'_i, \dots, y_i^{(k-1)}$. El problema consiste en encontrar un polinomio $H_m(x)$, de orden a lo más $m = kn - 1$, que coincida en los puntos x_i con los valores y_i y el de sus respectivas derivadas hasta orden $k - 1$, $H_m^{(k-1)}(x)$, con los valores $y'_i, \dots, y_i^{(k-1)}$, o sea, $H_m(x)$ es tal que para $i = 1, 2, \dots, n$,

$$H_m(x_i) = y_i, \quad H'_m(x_i) = y'_i, \quad \dots \quad H_m^{(k-1)}(x_i) = y_i^{(k-1)}.$$

En general se tiene el siguiente resultado: Dados n puntos diferentes x_1, x_2, \dots, x_n y n números enteros a_1, a_2, \dots, a_n , mayores que 1, existe un único polinomio de orden $m = a_1 + a_2 + \dots + a_n - 1$ que cumple con:

$$H_m(x_i) = f(x_i) = y_i, \quad H'_m(x_i) = f'(x_i) = y'_i, \quad \dots \quad H_m^{(a_i-1)}(x_i) = f^{(a_i-1)}(x_i) = y^{(a_i-1)}.$$

Dicho polinomio, es conocido como polinomio de interpolación de Hermite y se puede calcular de la siguiente forma:

$$H_m(x) = \sum_{i=1}^n \sum_{k=0}^{a_i-1} y_i^{(k)} l_{ik}(x), \quad \text{donde} \\ w(x) = \prod_{i=1}^n (x - x_i)^{a_i}, \quad l_{ik}(x) = w(x) \frac{(x - x_i)^{k-a_i}}{k!} \frac{d^{a_i-k-1}}{dx^{a_i-k-1}} \left[\frac{(x - x_i)^{a_i}}{w(x)} \right]_{x=x_i}, \quad (8.11.1)$$

donde usaremos la notación $y_i^{(0)} = y_i$.

Aquí, al igual que en el caso de la interpolación de Lagrange conviene tener en cuenta que al aumentar el número de puntos la interpolación de Hermite también falla. Además, al ser un polinomio, tiene el mismo problema de inestabilidad a la hora de evaluarlo numéricamente.

Como reto para el lector se propone implementar una rutina que calcule dicho polinomio usando la fórmula (8.11.1) y que la use en algunos ejemplos con un número no muy grande de puntos.

Introducción al MAXIMA CAS con algunas aplicaciones
Prof. Dr. Renato Álvarez Nodarse

Bibliografía

- [1] *Maxima 5.42.0 Manual* (versión original inglés, más actualizada) o *Manual de Referencia (versión 5.35.1)* (traducción al castellano) Ver online o descargar el fichero pdf desde <http://maxima.sourceforge.net/documentation.html>
- [2] M. Abramowitz y I. A. Stegun, *Handbook of Mathematical Functions* Dover, 1964.
- [3] R. Álvarez-Nodarse. *Notas del curso Ampliación de Análisis Matemático: Ecuaciones diferenciales ordinarias*. Disponible en formato pdf en <https://renato.ryn-fismat.es/clases/aam/>
- [4] M. Braun, *Differential Equations and their Applications*. Springer Verlag, 1993.
- [5] A. F. Nikiforov y V. B. Uvarov, *Special Functions of Mathematical Physics*. Birkhäuser Verlag, 1988.
- [6] G.F. Simmons, *Ecuaciones diferenciales: con aplicaciones y notas históricas*. McGraw-Hill, 1993.
- [7] V.A. Zorich, *Mathematical Analysis*, tomos I y II, Springer Verlag, 2004.