Aprenda Informática ...

tecnun

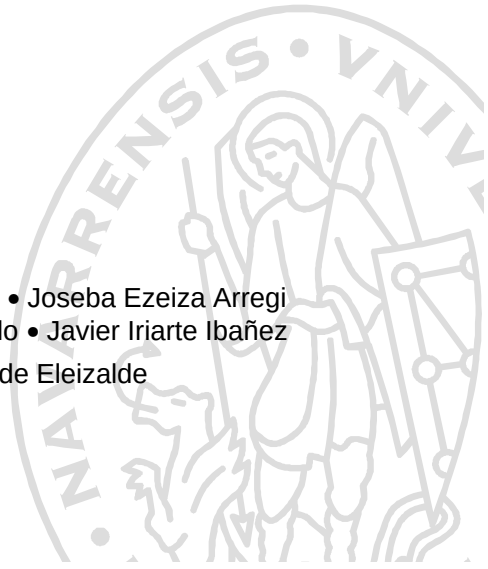
CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda Maple 9.5

como si estuviera en primero

San Sebastián, Octubre de 2004

Sara Aguarón Iraola • Unai Arrieta Salgado • Joseba Ezeiza Arregi
Aitor Erdozain Ibarra • Cristina Pastor Coello • Javier Iriarte Ibañez
Coordinado por Carlos Bastero de Eleizalde



tecnun
CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda Maple 9.5

como si estuviera en primero

Sara Aguarón Iraola

Unai Arrieta Salgado

Joseba Ezeiza Arregi

Aitor Erdozain Ibarra

Cristina Pastor Coello

Javier Iriarte Ibáñez

Coordinado por Carlos Bastero de Eleizalde

Perteneciente a la colección : *"Aprenda …, como si estuviera en primero"*

PRESENTACIÓN

El manual ***Aprenda Maple 9.5 como si estuviera en primero***, es una adaptación de los manuales que, para anteriores versiones de Maple, fueron escritas por Javier García de Jalón, Rufino Goñi, Francisco Javier Funes, Iñigo Girón, Alfonso Brazález y José Ignacio Rodríguez, en la *release* 3; por Javier García de Jalón, Amale Garay, Itziar Ricondo y Pablo Valencia, en Maple 5 y por Ainhoa Ochoa, Iñaki Esnaola, Eduardo Portu y Eduardo Granados en Maple 8.

Este manual, a diferencia de los anteriores, ha pretendido poner algunos ejemplos más para la mejor comprensión de los comandos, lo que ha llevado a un aumento significativo en el número de páginas con respecto a los anteriores. Han sido de gran utilidad las ideas sobre el modo de explicar este paquete de software obtenidas del libro de D. Richards "*Advanced Mathematical Methods with Maple®*", publicado por Cambridge University Press en 2002.

Todo tipo de sugerencias serán bienvenidas para mejorar ulteriores ediciones de este manual.

Las web sites, que a continuación se citan, pueden ser útiles para profundizar en Maple o para obtener programas y bibliotecas de dominio público:
http://www.mapleapps.com/ para obtención de programas
http://www.maple4students.com/ para tutoriales sobre temas matemáticos

Donostia-San Sebastián, 8 de octubre de 2004

1- INTRODUCCIÓN A MAPLE 9.5

El programa que se describe en este manual es probablemente muy diferente a todos los que se han visto hasta ahora, en relación con el cálculo y las matemáticas. La principal característica es que Maple es capaz de realizar cálculos simbólicos, es decir, operaciones similares a las que se llevan a cabo por ejemplo cuando, intentando realizar una demostración matemática, se despeja una variable de una expresión, se sustituye en otra expresión matemática, se agrupan términos, se simplifica, se deriva y/o se integra, etc. También en estas tareas puede ayudar el ordenador, y Maple es una de las herramientas que existen para ello. Pronto se verá, aunque no sea más que por encima, lo útil que puede llegar a ser este programa.

Además, Maple cuenta con un gran conjunto de herramientas gráficas que permiten visualizar los resultados (algunas veces complejos) obtenidos, algoritmos numéricos para poder estimar resultados y resolver problemas donde soluciones exactas no existan y también un lenguaje de programación para que el usuario pueda desarrollar sus propias funciones y programas.

Maple es también idóneo para realizar documentos técnicos. El usuario puede crear hojas de trabajo interactivas basadas en cálculos matemáticos en las que puede cambiar un dato o una ecuación y actualizar todas las soluciones inmediatamente. Además, el programa cuenta con una gran facilidad para estructurarlos, empleando herramientas tales como los estilos o los hipervínculos, así como con la posibilidad de traducir y exportar documentos realizados a otros formatos como HTML, RTF, LaTeX y XML.

¿QUÉ ES MAPLE 9.5?

Maple es un programa desarrollado desde 1980 por el grupo de Cálculo Simbólico de la Universidad de Waterloo (Ontario, CANADÁ). Su nombre proviene de las palabras *MAthematical PLEasure*. Existen versiones para los ordenadores más corrientes del mercado, y por supuesto para los PCs que corren bajo ***Windows*** de Microsoft. La primera versión que se instaló en las salas de PCs de la ESI en Octubre de 1994 fue Maple V Release 3. La versión instalada actualmente es Maple 9.5, que tiene considerables mejoras respecto a la versión anterior, además de contar con dos modos de trabajo: el denominado *Classic Worksheet* y el modo normal.

Para arrancar Maple desde cualquier versión de ***Windows*** se puede utilizar el menú ***Start***, del modo habitual. También puede arrancarse clicando dos veces sobre un fichero producido con Maple en una sesión anterior, que tendrá la extensión ****.mws*** si se trata de un fichero creado en el *Classic Worksheet* o ****.mw*** si ha sido creado en el modo normal. Nótese que ambos tipos de ficheros pueden ser abiertos con el otro modo de trabajo, es decir, un ****.mw*** puede ser abierto con el *Classic Worksheet* y viceversa. En cualquier caso, el programa arranca y aparece la ventana de trabajo (ver ilustracion 1), que es similar a la de muchas otras aplicaciones de ***Windows***. En ambos modos de trabajo, en la primera línea aparece el ***prompt*** de Maple: el carácter "mayor que" (>). Esto quiere decir que el programa está listo para recibir instrucciones.

Maple es capaz de resolver una amplia gama de problemas. De interés particular son los basados en el uso de métodos simbólicos. A lo largo de las páginas de este manual es posible llegar a hacerse una idea bastante ajustada de qué tipos de problemas pueden llegar a resolverse con Maple 9.5.

2- DESCRIPCIÓN GENERAL DE MAPLE 9.5

2.1. MANEJO DE LA AYUDA DE MAPLE

La ayuda de Maple resulta muy útil a la hora de trabajar con este programa. Gracias a ella podemos ver cómo se trabaja con los distintos comandos además de encontrar numerosos ejemplos.

Hay distintas formas de acceder a la ayuda de este programa. Una de las más rápidas consiste en anteponer un signo de interrogación al comando sobre el cual queremos información.

```
> ?Int
```

Podemos obtener la misma información a través del menú Help. Si situamos el cursor sobre el nombre del comando vemos que en dicho menú se ha activado la opción Help on "comando". También resulta útil buscar información acerca de algún tema concreto mediante el Topic Index o el Search. Al elegir Help/Using Help aparece un pequeño resumen de cómo trabajar con la ayuda del programa.

Sea cual sea la forma escogida para acceder a la ayuda siempre aparecerá una nueva ventana. Esta ventana está separada en dos partes. En la parte izquierda tenemos distintas pestañas. Si la que está activada es la de Contents entonces en la parte inferior aparecerá una lista con todos los contenidos de la ayuda. Si por el contrario es la de Topic o la de Search la que está activada, entonces en la parte inferior encontramos una lista con todos los temas relacionados con la palabra o palabras que hayamos introducido en el buscador. En la parte derecha encontramos toda la información. Es aquí donde vemos cómo se utiliza y para qué sirve el comando además de incluir algunos ejemplos y otros términos relacionados.

Si nunca se ha manejado Maple es recomendable efectuar el New User´s Tour. Existe la posibilidad de realizar el Quick Tour, que nos da una idea más general, o el Full Tour que nos proporciona una información más detallada acerca de las funcionalidades y comandos del programa más utilizados, incluyendo ejemplos ilustrativos.

Existen además dos herramientas que pueden resultar de gran ayuda a la hora de trabajar con Maple. Por un lado tenemos las Example Worksheets que son hojas ejecutables que contienen ejemplos explicados detalladamente acerca de distintos temas, y por otro lado cabe destacar la existencia de un diccionario en el que encontramos definiciones para más de 5000 términos matemáticos.

2.2. NÚCLEO, LIBRERIAS E INTERFACE DE USUARIO

Las tres partes principales que forman Maple son el núcleo, las librerías y la interface de usuario.

El núcleo es la parte principal del programa, la que lleva a cabo todas las operaciones matemáticas. Esta parte del programa está escrita en el lenguaje C.

Las librerías son otra de las partes fundamentales del programa. Maple dispone de un alto número de comandos, pero no todos se cargan al iniciar el programa, sólo aquellos que son más importantes. El resto de los comandos se encuentran en las diferentes librerías temáticas y si queremos utilizarlos debemos cargarlos previamente. A la hora de cargar una librería existen dos opciones: cargar la librería completa, lo cual resulta útil si vamos a emplear varias de las funciones que incluye, o cargar un comando o función aislada.

Para cargar la librería completa utilizamos el comando with(library). De esta forma para utilizar una de las funciones incluidas en dicha librería sólo tendremos que escribir el nombre de la función con sus correspondientes argumentos.

```
> with(LinearAlgebra):
> RandomMatrix(3);
```

Si por el contrario sólo vamos a utilizar una de las funciones podemos hacerlo de la siguiente forma: librería[función](argumentos).

```
> LinearAlgebra[RandomMatrix](3);
```

Para ver todas las librerías que Maple tiene disponibles hay que teclear Package en Topic Search del menú Help. De este modo aparecerá una lista con todas ellas. Al clicar sobre alguna de ellas obtendremos una descripción de su uso así como una lista con todos los comandos y funciones que incluye.

La interface de usuario es la parte del programa que se ocupa de las operaciones de entrada y salida de información. A la hora de arrancar Maple tenemos la posibilidad de elegir entre dos hojas de trabajo distintas, la Classic Worksheet o la Standard Worksheet. Tanto una como la otra pueden desempeñar las mismas funciones y cuentan con las mejoras introducidas en la versión 9.5. Al elegir trabajar con la Classic Worksheet encontramos una hoja de trabajo con el mismo diseño que en versiones anteriores que además utiliza menos memoria que la Standard Worksheet. Esta última tiene un aspecto nuevo y ha sido mejorada de modo que el acceso a distintas funciones es más directo ya que en la pantalla aparecen por defecto desplegadas las paletas. Estas herramientas resultan de gran utilidad y serán analizadas con más detalle más adelante.

2.3. DESCRIPCIÓN DEL ENTORNO DE TRABAJO

2.3.1. Organización de una hoja de trabajo. Grupos de regiones.

Maple cuenta con distintas herramientas que permiten organizar las hojas de trabajo de una forma ordenada y clara. Existe la posibilidad de incluir texto, comentarios, establecer accesos directos a otras hojas de trabajo o a otra parte de la propia hoja. Podemos también crear grupos o secciones.

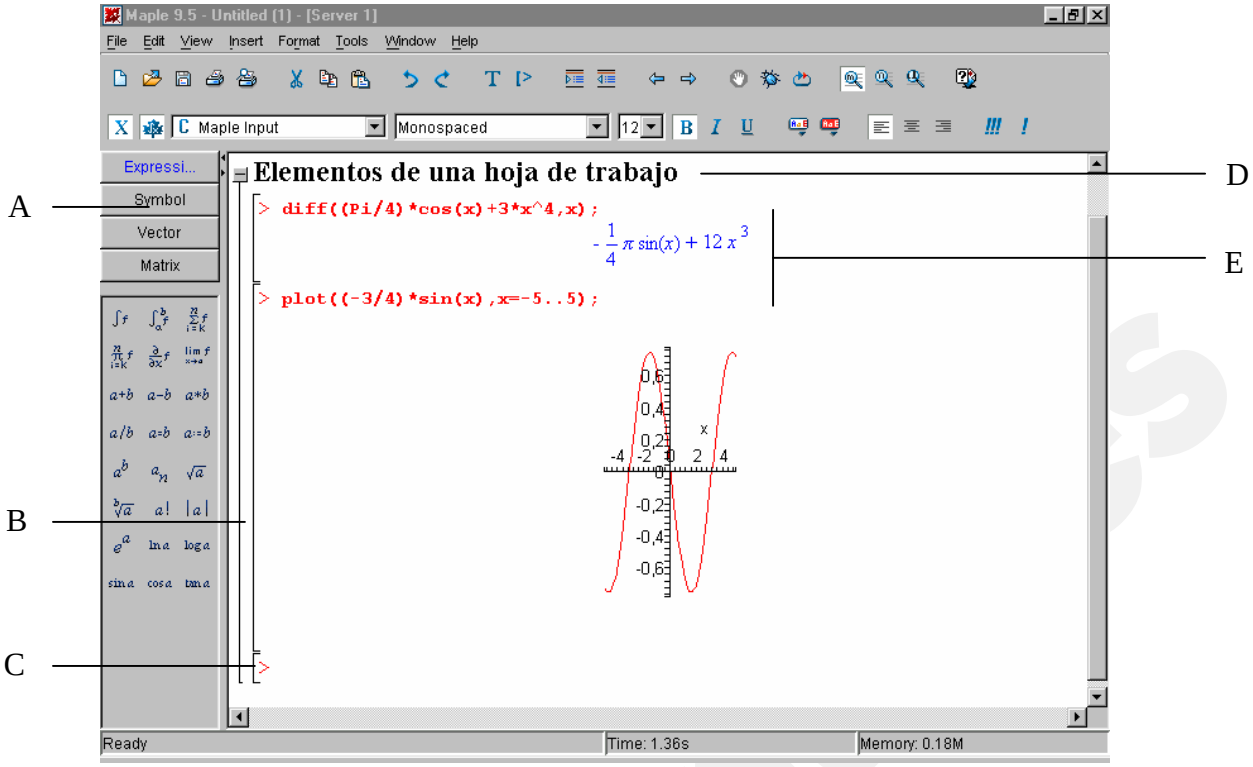


Ilustración 1

A: Paletas

B: Sección

C: Prompt (Es aquí donde se efectúa la entrada)

D: Título de la sección

E: Grupo de ejecución

En una hoja de trabajo podemos encontrar cuatro tipos distintos de regiones o zonas, cada una de ellas con su comportamiento específico. Estas regiones son la de entrada, la de salida, la de texto y por último la de gráficos. Por defecto cada una de estas regiones tiene un estilo predefinido que puede ser modificado. Para ello seleccionamos en el menú Format la opción Styles. Aparece entonces una ventana en la cual, una vez seleccionado el estilo que deseamos modificar, clicamos en Modify para hacer los cambios que queramos. Existe también la posibilidad de crear un nuevo estilo. Para ello se debe teclear en Create Character Style dentro del mismo menú. Dado que Maple también permite introducir texto también podemos crear un estilo de párrafo clicando Create Paragraph Style.

Hay dos formas distintas de realizar la entrada: Maple Notation o Standard Math, las cuales se seleccionan del menú Tools/Options/Display. En el caso de querer cambiar la notación de una línea en concreto podemos hacerlo clicando sobre ella con el botón derecho y seleccionando Convert To/ Standard Math.

Si elegimos la Maple Notation (la cual está activada por defecto) tendremos que introducir los comandos según la sintaxis empleada por Maple y finalizar las sentencias con ‘;’ o ‘:’. Si empleamos el punto y coma al presionar Enter la sentencia se ejecuta y el resultado aparece por pantalla, mientras que si utilizamos los dos puntos el resultado no aparecerá por pantalla.

Si por el contrario la opción activada es la de Standard Math aparecerá un signo de interrogación junto al prompt (>) y las sentencias que se introduzcan aparecerán en la barra de textos en vez de en la hoja de trabajo. Para que estas sentencias se ejecuten es necesario pulsar dos veces la tecla "intro".

Maple incluye cuatro paletas (símbolos, expresiones, vectores y matrices) que resultan de gran utilidad cuando no se conoce la sintaxis específica de los comandos y se quiere construir una expresión matemática. Al trabajar con la Standard Worksheet las cuatro paletas aparecen desplegadas por defecto. En el caso de querer ocultar o volver a mostrar una paleta puede accederse a ellas mediante el menú View/Palette y una vez ahí seleccionar las distintas opciones. El modo de trabajo también influye cuando utilizamos las paletas. De esta forma si estamos trabajando con Standard Math y seleccionamos en la paleta el icono de integral definida tendremos:

$$\int_{?}^{?} ? \, d? \qquad >$$

Una vez obtenida esta expresión hay que sustituir los signos de interrogación por los valores de nuestra integral. Para desplazarse de uno a otro se emplea el tabulador. Al trabajar con Maple Notation lo que obtendremos será:

```
> int(%f,%x=%a..%b);
```

En esta expresión tendremos que sustituir los elementos precedidos por % por los valores deseados. Se puede observar que dado que estamos trabajando con Maple Notation automáticamente se incluye el punto y coma al finalizar la sentencia.

Otro hecho que también puede resultar de gran ayuda al utilizar este programa es que a medida que vamos introduciendo un comando Maple compara la entrada con todas sus librerías, funciones y comandos. Si sólo encuentra una única posibilidad de coincidencia aparece una etiqueta con el nombre del comando encima de lo que estamos escribiendo. De este modo basta con presionar Intro para que el comando se complete. En el caso de que haya más de una opción no aparecerá ningún comando automáticamente pero al teclear control+espacio aparecerá una lista con todos los comandos que empiezan por las letras introducidas.

Para introducir texto en la hoja de trabajo se puede clicar en el botón

o seleccionar Insert/Text. Sobre el texto se pueden aplicar distintos estilos tal y como lo haríamos al trabajar con Word. Si dentro del texto que estamos escribiendo queremos introducir una expresión con notación matemática podemos escribirla como texto y después la seleccionamos. Una vez seleccionada en el menú Format/Convert To elegimos Standard Math. De este modo si introducimos el siguiente texto: "La ecuación es x=2/(4+y)^2" obtendremos lo siguiente,

Después de $\left[\text{La ecuación es } x = \frac{2}{(4+y)^2}\right.$ introducir el texto si queremos

volver a introducir sentencias clicamos sobre el botón

En el caso de que simplemente queramos incluir un comentario que no se ejecute con el programa basta con escribir el símbolo #. De este modo el programa ignorará todo lo que se escriba en esa línea a partir de dicho símbolo.

Para que el aspecto de nuestra hoja de trabajo sea más claro y ordenado es posible introducir secciones desde el menú Insert/Section. Gracias a las secciones podemos agrupar varios elementos. Si decidimos crear una sección una vez que ya tenemos los distintos elementos en nuestra hoja de trabajo seleccionamos dichos elementos y clicamos en el botón

Automáticamente aparecerá un espacio para introducir el nombre de la sección. Otra forma de agrupar comandos es crear grupos. Para ello una vez finalizada una sentencia en lugar de pulsar Intro, para que se ejecute la sentencia, se debe pulsar Shift+Intro. De esta forma se puede continuar escribiendo en la siguiente línea. En el momento en que se pulse únicamente Intro todas las sentencias que pertenezcan al grupo creado se ejecutarán al mismo tiempo. Para poder ver los distintos grupos aparecen unas líneas que engloban todas las sentencias pertenecientes a un mismo grupo (ver en el apartado anterior interface de usuario). Podemos hacer que estas líneas aparezcan o desaparezcan de la pantalla con el comando View/Show Contents... y una vez ahí seleccionar Execution Groups. En Maple existen comandos que permiten separar un grupo o unir dos grupos contiguos. Para ello hacemos uso del menú Edit/Split or Join. Si en el menú que aparece a continuación escogemos Split Execution Group se parte el grupo por encima de la línea en la que está el cursor. Si por el contrario elegimos Join Execution Group, se une el grupo en el que está el cursor con el posterior. Estas mismas operaciones se pueden realizar con las secciones desde el mismo menú. Para introducir un grupo entre otros ya existentes, se posiciona el cursor en el lugar donde se quiere insertar el grupo, se clica en Insert/Execution Group y se indica si se quiere insertar antes o después del cursor.

Si se quiere utilizar el resultado obtenido en una sentencia en la siguiente, podemos hacerlo con el símbolo del porcentaje (%). Si queremos hacer referencia al penúltimo o antepenúltimo resultado utilizaremos los operadores %% o %%% respectivamente. Por ejemplo para integrar el resultado obtenido tras una combinación de funciones lo haremos del siguiente modo:

```
> f:=x->x^2:
> g:=x->x+1:
> h:=f@g:
> h(x);
```

$$(x+1)^2$$

```
> int(%,x);
```

$$\frac{1}{3}(x+1)^3$$

Una de las posibilidades que ofrece Maple es la de mostrar una lista

con las operaciones más generales que se podrían realizar con el resultado obtenido. Para ello hay que clicar con el botón derecho sobre la respuesta que nos da el programa.

Se pueden crear accesos directos a otras hojas o a marcas (Bookmark). Para ello se selecciona el texto que vaya a ser el acceso directo y se va a Format/Convert to/Hyperlink. En el cuadro de diálogo que aparece, se indica la hoja de trabajo (Worksheet) o el tópico del que se quiere ayuda (Help Topic), y automáticamente se crea el acceso directo. Así pues, si se clica sobre el texto que se haya convertido en acceso directo, inmediatamente se nos posicionará el cursor donde se haya creado el acceso, ya sea hoja de trabajo, ayuda o una marca.

Por otro lado, tenemos dos opciones para salir de Maple, bien a través del menú File/Exit o simplemente tecleando Alt+F4.

2.3.2. Edición de hojas de trabajo

El programa permite desplazarse a lo largo de toda la hoja de trabajo y modificar los comandos introducidos en cualquier momento o ejecutarlos en orden distinto. Esta posibilidad proporciona una gran flexibilidad a la hora de trabajar. Sin embargo, tenemos que tener en cuenta que el modificar alguna sentencia y no volver a ejecutar la hoja completa puede provocar errores dado que el resultado obtenido tras una operación puede ser empleado en otra. Por ello para ejecutar la hoja completamente conviene utilizar el botón

o hacerlo a través del menú Edit con la opción Execute/Worksheet. Cuando se recalcula una hoja de trabajo los gráficos se recalculan. Antes de llevar a cabo esta operación conviene cerciorarse de que en el menú Tools/Options/Display la opción Replace existing output when re-executing groups está activada. Este modo será visto con más detalle en la siguiente sección.

Conviene saber que, si una vez que hemos pulsado intro para ejecutar una sentencia decidimos cambiar algo o vemos que el ordenador está tardando demasiado, podemos interrumpir el cálculo pulsando el botón

2.3.3. Modos de trabajo

Maple dispone de varios modos de trabajo, que se seleccionan en el menú Tools/Options. Existen diversos aspectos que pueden ser modificados, sin embargo hay algunos que resultan de mayor utilidad que otros.

En el apartado General podemos seleccionar el modo en el que trabaja el núcleo: en paralelo, compartido o mixto. Todo lo relacionado con el estado interno del programa se especifica en el siguiente apartado.

En el apartado Display podemos modificar varios elementos. Por un lado, es aquí donde se especifica si la entrada y salida se efectúa en Maple Notation o Standard Math. Además si queremos que cada resultado sustituya al anterior en la región de salida correspondiente debemos activar la opción Replace existing output when re-executing groups. Si este modo no está activado, cada resultado se inserta antes del resultado previo en la misma región de salida, con la posibilidad de ver ambos resultados a la vez

y compararlos. Si la opción Use insert mode está activada al pulsar intro en una línea de entrada de un grupo se crea una nueva región de entrada y un nuevo grupo inmediatamente a continuación. Si esta opción no estuviese activada el nuevo grupo sólo se crearía en el caso de que estuviéramos ejecutando el último grupo de la hoja de trabajo.

Por último, en el apartado Interface podemos elegir si , en el caso de trabajar con varias hojas de trabajo, queremos que se vean todas a la vez en la pantalla o no. Para ello debemos seleccionar la opción correspondiente en Window manager.

2.3.4. Estado interno del programa

El estado interno de Maple consiste en todas las variables e información que el programa tiene almacenados en un determinado momento de ejecución. Si por algún motivo hay que interrumpir la sesión de trabajo y salir de la aplicación, pero se desea reanudarla más tarde, ese estado interno se conserva, guardando las variables los valores que tenían antes e cerrar la hoja de trabajo. Para anular todas las definiciones y cambios que se hayan efectuado en la hoja de trabajo utilizamos el comando Restart. De este modo volvemos al estado original de la hoja de trabajo.

Es conveniente conocer el modo en el que trabaja el núcleo del programa (ver apartado anterior). Si lo hace en modo compartido (shared) las variables con el mismo nombre que pertenecen a hojas de trabajo distintas tienen el mismo valor. Si el modo seleccionado es el paralelo (parallel) las variables pertenecientes a hojas de trabajo distintas serán independientes a pesar de tener el mismo nombre. Si la opción activada es mixto (mixed) será necesario definir el carácter de cada variable.

A veces puede ser conveniente eliminar todas o parte de las regiones de salida. Esto se logra con ***Edit/Remove Output/From Worksheet*** o ***From Selection*** respectivamente. También puede ser conveniente ejecutar toda o parte de la hoja de trabajo mediante ***Edit/Execute/Worksheet*** o ***Selection*** respectivamente. Cuando se ejecuten estos comandos sobre una selección habrá que realizar la selección previamente, como es obvio.

Con el comando ***Save*** del menú ***File*** se almacenan los *resultados externos de la sesión de trabajo* (regiones de entrada, salida, texto y gráficos) en un fichero con extensión ****.mws*** en el caso de estar trabajando con la Classic Worksheet, o extensión .mw si trabajamos con la Standard Worksheet. En el caso de que la hoja de trabajo se quiera abrir con versiones anteriores de Maple puede ser conveniente guardarla en cualquier caso con extensión .mws. Para ello, en el caso de estar trabajando con la Standard Worksheet, seleccionamos la opción Save As y en el menú que aparece seleccionamos Classic Worksheet. Estos ficheros no son ficheros de texto, y por tanto no son visibles ni editables con ***Notepad*** o ***Word***.

2.4. OBJETOS EN MAPLE

Los objetos de Maple son los tipos de datos y operadores con los que el programa es capaz de trabajar. A continuación se explican los objetos más importantes.

2.4.1. Números y variables

Maple trabaja con *números enteros* con un número de cifras arbitrario. Por ejemplo, no hay ninguna dificultad en calcular números muy grandes como factorial de 100, o 3 elevado a 50. Si el usuario lo desea, puede hacer la prueba.

Maple tiene también una forma particular de trabajar con *números racionales* e *irracionales*, intentando siempre evitar operaciones aritméticas que introduzcan errores. Ejecute por ejemplo los siguientes comandos, observando los resultados obtenidos (se pueden poner varios comandos en la misma línea separados por comas, siempre que no sean sentencias de asignación y que un comando no necesite de los resultados de los anteriores):

```
> 3/7, 3/7+2, 3/7+2/11, 2/11+sqrt(2), sqrt(9)+5^(1/3);
```

$$\frac{3}{7}, \frac{17}{7}, \frac{47}{77}, \frac{2}{11}+\sqrt{2}, 3+5^{1/3}$$

Si en una sentencia del estilo de las anteriores, uno de los números tiene un punto decimal, Maple calcula todo en aritmética de punto flotante. Por defecto se utiliza una precisión de 10 cifras decimales. Observe el siguiente ejemplo, casi análogo a uno de los hechos previamente:

```
> 3/7+2./11;
```

$$.6103896104$$

La precisión en los cálculos de punto flotante se controla con la variable ***Digits***, que como se ha dicho, por defecto vale 10. En el siguiente ejemplo se trabajará con 25 cifras decimales exactas:

```
> Digits := 25;
```

Se puede forzar la evaluación en punto flotante de una expresión por medio de la función ***evalf***. Observe el siguiente resultado:

```
> sqrt(9)+5^(1/3);
```

$$3+5^{1/3}$$

```
> evalf(%);
```

$$4.7099759466766969893531$$

```
> 3*Pi;
```

$$3\,\pi$$

```
> evalf(%);
```

$$9.424777962$$

La función ***evalf*** admite como segundo argumento opcional el número de dígitos. Por ejemplo para evaluar la expresión anterior con 40 dígitos sin cambiar el número de dígitos por defecto, se puede hacer:

```
> evalf(sqrt(9)+5^(1/3), 40);
```

$$4.7099759466766969893531088725438601098 68$$

Existe también la posibilidad de trabajar con la función evalhf, la cual realiza la misma operación que evalf pero desde el hardware en lugar del software. Esta función admite un único argumento ya que el número de cifras decimales lo fija el propio hardware. El programa incluye esta opción porque resulta más rápida al ahora de hacer cálculos matemáticos.

Asimismo, Maple permite trabajar con números complejos. ***I*** es el símbolo por defecto de la unidad imaginaria, esto es $I = \sqrt{-1}$. Así, vemos:

```
> (8+5*I) + (3-2*I);
```

$$11 + 3\,I$$

```
> (8+5*I) / (3-2*I);
```

$$\frac{14}{13} + \frac{31}{13} I$$

En cuanto a las variables, Maple permite una gran libertad a la hora de definirlas. Pueden ser creadas en cualquier momento y además no tienen tipo fijo ya que el tipo de una misma variable puede cambiar varias veces a lo largo de la sesión. En cuanto al nombre, no existe límite práctico en el número de caracteres y puede estar formado por letras, números, guiones y guiones bajos, siempre que no empiecen por un número. Tampoco es conveniente empezar un nombre con un guión bajo ya que el programa usa ese tipo de nombres para su clasificación interna. También hay que tener en cuenta que existen ciertos nombres que no pueden ser empleados ya que son palabras reservadas de Maple como por ejemplo el símbolo Pi o funciones matemáticas como cos o sin. También se puede definir un nombre entre comillas simples. Teniendo todo esto en cuenta nombres válidos son “polinomio”, “cadena _ carácter”. Inválidos serían “2_fase” (empieza por número) y “x&y” (porque & no es un carácter alfanumérico).

Una de las características más importantes de Maple es la de poder trabajar con *variables sin valor numérico*, o lo que es lo mismo, *variables no-evaluadas*. En MATLAB o en C una variable siempre tiene un valor (contiene basura informática si no ha sido inicializada). En Maple una variable puede ser simplemente una variable, sin ningún valor asignado, al igual que cuando una persona trabaja con ecuaciones sobre una hoja de papel. Es con este tipo de variables con las que se trabaja en cálculo simbólico. Suele decirse que *estas variables se evalúan a su propio nombre*. A continuación se verán algunos ejemplos. En primer lugar se va a resolver una ecuación de segundo grado, en la cual ni los coeficientes (**a**, **b** y **c**) ni la incógnita **x** tienen valor concreto asignado. A la función ***solve*** hay que especificarle que la incógnita es **x** (también podrían ser **a**, **b** o **c**):

```
> solve(a*x**2 + b*x + c, x);  # a,b,c parámetros; x incógnita
```

$$\frac{1}{2}\frac{-b+\sqrt{b^2-4\,a\,c}}{a}, \frac{1}{2}\frac{-b-\sqrt{b^2-4\,a\,c}}{a}$$

Para asignarle un valor a una variable utilizamos el operador :=. De este modo si a la variable y le queremos asociar la ecuación (x+1)^2 lo haremos escribiendo:

```
> y:=(x+1)^2;
```

De este modo hasta que la variable y vuelva a estar asignada a su propio nombre su valor será (x+1)^2. El siguiente ejemplo puede aclarar un poco los conceptos. Definimos un polinomio cuya variable es la y:

```
> polinomio:=3*y^3+2*y^2+y+6;
```

$$polinomio := 3\,y^3 + 2\,y^2 + y + 6$$

Ahora, tal y como lo habíamos hecho en el ejemplo anterior, asignamos la variable y el valor (x+1)^2 y observamos lo que ocurre con "polinomio":

```
> y:=(x+1)^2;   polinomio;
```

$$y := (x+1)^2$$

$$3\,(x+1)^6 + 2\,(x+1)^4 + (x+1)^2 + 6$$

Si además x le damos el valor 5:

```
> x:=5;  y; polinomio;
```

$$x := 5$$

$$36$$

$$142602$$

Puede suceder que queramos realizar la integral de polinomio respecto a x. Si usamos el comando int obtendremos un error, ya que después de efectuar la sustitución "polinomio" es una constante. Por lo tanto puede resultar conveniente emplear el comando subs. Con este comando podemos hacer una sustitución de x en y pero no le asignamos a la variable x ningún valor:

```
> subs(x=5,y);  y; polinomio;
```

$$36$$

$$(x+1)^2$$

$$3\,(x+1)^6 + 2\,(x+1)^4 + (x+1)^2 + 6$$

Otra forma de asignar un valor a una variable es utilizando el comando assign(name,expresión). Esta expresión equivale a name:=expresión; excepto en que en el primer argumento de assign la función se evalúa completamente (no ocurre así con el miembro izquierdo del operador de asignación :=). Esto es importante, por ejemplo, en el caso de la función solve, que devuelve un conjunto de soluciones no asignadas. Para aclarar estos conceptos véase el siguiente ejemplo, en el que comenzamos con definir un conjunto de ecuaciones y otro de variables:

```
> restart;
> ecs:= {a*x+3*y=b, c*x+1/2*y=d};  vars:= {x,y};
```

$$ecs := \left\{ a\,x + 3\,y = b,\ c\,x + \frac{1}{2}\,y = d \right\}$$

$$vars := \{y, x\}$$

A continuación se resuelve el conjunto de ecuaciones respecto al de variables, para hallar un conjunto de soluciones:

$$sols := \left\{ x = -\frac{6\,d - b}{a - 6\,c}, y = \frac{2\,(-c\,b + d\,a)}{a - 6\,c} \right\}$$

```
> sols:= solve(ecs, vars);
```

El resultado anterior no hace que se asignen las correspondientes expresiones a x e y. Para hacer asta asignación utilizamos la función assign de la siguiente forma:

```
> x, y; assign(sols);
```

$$x, y$$

```
> x, y;
```

$$\frac{-6\,d + b}{a - 6\,c}, \frac{2\,(a\,d - c\,b)}{a - 6\,c}$$

Otros ejemplos interesantes relacionados con el comando assign son los siguientes:

```
> assign(a,b);
> assign(a=c,d=2);
> a,b,c,d;
```

$$c, c, c, 2$$

```
> assign(('a','b') = (7,2));
> a,b,c;
```

$$7, 2, c$$

Una vez que tengamos una variable evaluada podemos hacer que vuelva a estar asignada a su propio nombre. Existen distintas formas de llevar a cabo este proceso. Por un lado si queremos *desasignar* todas las variables lo podemos hacer mediante el comando Restart, mientras que si sólo queremos evaluar una variable a su nombre debemos hacerlo utilizando otros modos. Uno de ellos consiste en asignar a la variable su propio nombre entre apóstrofes. Primero asignamos a la variable x el valor 3 y vemos que automáticamente la x es sustituida por su valor:

```
> x:=3; x;
```

$$x := 3$$

$$3$$

Pero si igualamos x a su nombre entre apóstrofes x vuelve estar asignada a su propio nombre:

```
> x:='x'; x;
```

$$x := x$$

$$x$$

La norma de Maple es que todas las variables se evalúan tanto o tan lejos como sea posible, según se ha visto en el ejemplo anterior, en el que a **x** se le asignaba un 10

porque estaba asignada a **variable** y a **variable** se le había dado un valor 10. Esta regla tiene algunas excepciones como las siguientes:

- Las expresiones entre apóstrofos no se evalúan
- El nombre a la izquierda del operador de asignación (:=) no se evalúa

y por ello la expresión **x := 'x';** hace que **x** se vuelva a evaluar a su propio nombre:

Otra forma de desasignar una variable es por medio la función ***evaln***, como por ejemplo:

```
> x := 7; x := evaln(x); x;
```

$$x := 7$$

$$x := x$$

$$x$$

La función ***evaln*** es especialmente adecuada para desasignar variables subindicadas **a[i]** o nombres concatenados con números **a||i**. Considérese el siguiente ejemplo:

```
> i:=1; a[i]:=2; a||i:=3;
```

$$i := 1$$

$$a_1 := 2$$

$$a1 := 3$$

Supóngase que ahora se quiere desasignar la variable a[i]:

```
> a[i] := 'a[i]'; # no es esto lo que se quiere hacer, pues se pretende que la i siga valiendo 1, pero el valor asignado a a1 no sea 2
```

$$a_1 := a_i$$

```
> a[i] := evaln(a[i]); a[i]; # ahora si lo hace bien
```

$$a_1 := a_1$$

$$a_1$$

```
> a||i; a||i:='a||i'; a||i := evaln(a||i); # con nombres concatenados
```

$$3$$

$$a1 := a \parallel i$$

$$a1 := a1$$

En Maple hay comandos o funciones para listar las variables asignadas y sin asignar, y para chequear si una variable está asignada o no. Por ejemplo:

- ***anames***; muestra las variables asignadas (*assigned names*)
- ***unames***; muestra las variables sin asignar (*unassigned names*)

- ***assigned***; indica si una variable está asignada o no a algo diferente de su propio nombre

A los resultados de estas funciones se les pueden aplicar *filtros*, con objeto de obtener exactamente lo que se busca. Observe que los comandos del ejemplo siguiente,

```
> unames(): nops({%}); # no imprimir la salida de unames()
```

permiten saber cuántas variables no asignadas hay. La salida de ***unames*** es una *secuencia* –puede ser muy larga– que se puede convertir en *set* con las llaves {}. En el siguiente ejemplo se extraen por medio de la función ***select*** los nombres de variable con un solo carácter:

```
> select(s->length(s)=1, {unames()}): # se omite el resultado
```

Como resultado de los siguientes comandos se imprimirían respectivamente todos los nombres de variables y funciones asignados, y los que son de tipo entero,

```
> anames(); # se imprimen todas las funciones cargadas en esta sesión
```

```
> anames('integer');
```

Pude resultar útil en este caso pasar como argumento de la función anames la palabra users. De esta forma se obtendrá una lista con todas las variables definidas por el usuario:

```
> anames(user);
```

El siguiente ejemplo muestra cómo se puede saber si una variable está asignada o no:

```
> evalf(%);
```

$$9.424777962$$

```
> x1; x2 := gato; assigned(x1); assigned(x2);
```

$$x1$$

$$x2 := gato$$

$$false$$

$$true$$

Otras dos excepciones a la regla de evaluación completa de variables son las siguientes:

- el argumento de la función ***evaln*** no se evalúa (aunque esté asignado a otra variable, no se pasa a la función evaluada a dicha variable)
- el argumento de la función ***assigned*** no se evalúa

Por último, otra forma de desasignar las variables es mediante el comando unassign que permite asignar varias variables a su nombre a la vez:

```
> x:=3; y:=2*x+1; x; y;
```

$$x := 3$$

$$y := 7$$

$$3$$

$$7$$

```
> unassign('x','y'); x; y;
```

$$x$$

$$y$$

En Maple es muy importante el concepto de *evaluación completa* (*full evaluation*). Cuando Maple encuentra un nombre de variable en una expresión, busca hacia donde apunta ese nombre, y así sucesivamente hasta que llega a un nombre que apunta a sí

$$a := b$$

mismo o a algo que no es un nombre de variable, por ejemplo un valor numérico. Considérense los siguientes ejemplos:

```
> a:=b; b:=c; c:=3;
> a;  # a se evalúa hasta que se llega al valor de c, a través de b
```

$$b := c$$

$$c := 3$$

La función eval puede resultar de gran utilidad al trabajar con variables. Se puede

$$3$$

$$y := \cos(2\,\pi\,\sin(x))$$

comprobar a través del siguiente ejemplo:

```
> y:=cos(2*Pi*sin(x));
> z:=subs(x=0,y);
> eval(z);
```

$$z := \cos(2\,\pi\,\sin(0))$$

$$1$$

Si simplemente nos limitamos a utilizar el comando subs el ordenador no sustituye sin(0) por su valor numérico. Pero al utilizar la función eval la expresión se evalúa completamente. Existe también la posibilidad de utilizar directamente la función eval en lugar de realizar el paso intermedio con subs:

```
> y:=cos(2*Pi*sin(x));
```

$$y := \cos(2\,\pi\,\sin(x))$$

```
> z:=eval(y, x=0);
```

$$z := 1$$

La función ***eval*** permite controlar con su segundo argumento el nivel de evaluación de una variable:

```
> eval(a,1); eval(a,2); eval(a,3);
```

$$b$$

$$c$$

$$3$$

```
> c:=5: a;  # ahora, a se evalúa a 5
```

$$5$$

Muchas veces es necesario pasar algunos argumentos de una expresión *entre apóstrofes*, para evitar una evaluación distinta de su propio nombre (a esto se le suele llamar *evaluación prematura* del nombre, pues lo que se desea es que dicho nombre se evalúe dentro de la función, después de haber sido pasado como argumento e independientemente del valor que tuviera asignado antes de la llamada). Por ejemplo, la función que calcula el resto de la división entre polinomios devuelve el polinomio cociente como parámetro:

```
> x:='x': cociente := 0; rem(x**3+x+1, x**2+x+1, x, 'cociente');
cociente;
```

$$cociente := 0$$

$$2 + x$$

$$-1 + x$$

Si la variable cociente está desasignada, se puede utilizar sin apóstrofes en la llamada a la función. Sin embargo, si estuviera previamente asignada, no funcionaría:

```
> cociente:=evaln(cociente):
> rem(x**3+x+1, x**2+x+1, x, cociente); cociente;
```

$$2 + x$$

$$-1 + x$$

```
> cociente := 1; rem(x**3+x+1, x**2+x+1, x, cociente);
  Error, (in rem) Illegal use of a formal parameter
```

Otro punto en el que la evaluación de las variables tiene importancia es en las variables internas de los sumatorios. Si han sido asignadas previamente a algún valor puede haber problemas. Por ejemplo:

```
> i:=0: sum(ithprime(i), i=1..5);
Error, (in ithprime) argument must be a positive integer
> sum('ithprime(i)', i=1..5); # esto sólo no arregla el problema
Error, (in sum) summation variable previously assigned,
second argument evaluates to, 0 = 1 .. 5
> sum('ithprime(i)', 'i'=1..5); # ahora sí funciona
```

$$28$$

Considérese finalmente otro ejemplo de supresión de la evaluación de una expresión por medio de los apóstrofos:

```
> x:=1; x+1;
```

$$x := 1$$

$$2$$

```
> 'x'+1; 'x+1';
```

$$1+x$$

$$1+x$$

```
> '''x'+1''; %; %; %; # cada "último resultado" requiere una
evaluación
```

"*x*' + 1'

x + 1

'*x*' + 1

2

2.4.2. Cadenas de caracteres

Las cadenas de caracteres son también un objeto en Maple y se crean encerrando un número indefinido de caracteres entre comillas dobles. Así, tenemos:

```
> "Esto es una cadena de caracteres";
```

"Esto es una cadena de caracteres"

Son elementos a los cuales no podemos asignar un valor pero sí un nombre. Además se puede acceder individualmente a los caracteres de una cadena indicando su posición entre corchetes. Veamos un ejemplo:

```
> cadena:= "Mi cadena"; #Se les puede asignar un nombre
```

cadena := "Mi cadena"

```
> "Mi cadena":=21; #No se les puede asignar un valor
  Error, invalid left hand side of assignment
> cadena[6]; #Acceder a un elemento de la cadena
```

"d"

```
> cadena[4..-2]; #Accediendo a varios elementos (el negativo indica
que es el 2º elemento desde la derecha)
```

"caden"

El operador de concatenación, ||, o el comando cat, nos permite unir dos cadenas de caracteres.

```
> cadena:= "Las cadenas "; cadena2:="se pueden unir";
```

cadena := "Las cadenas "

cadena2 := "se pueden unir"

```
> cadena_def:= cat(cadena, cadena2);
```

cadena_def := "Las cadenas se pueden unir"

Para obtener el mismo resultado con el operador || tendríamos que hacerlo de la siguiente manera:

```
> cadena1:="Las cadenas "||cadena2;
```

cadena1 := "Las cadenas se pueden unir"

Por lo tanto a la hora de unir cadenas resulta más eficaz utilizar el comando cat y reservar el operador || para unir dos nombres o un nombre y un número. Al emplear el comando cat el resultado será un nombre o una cadena de caracteres dependiendo de la categoría a la que pertenezca el primer elemento. Así:

```
> cat('a','b');
```

ab

```
> cat("a","b");
```

"ab"

```
> cat('a',"b");
```

ab

```
> cat("a",'b');
```

"ab"

Este comando permite también incluir otro tipo de variables, tal y como se muestra en el siguiente ejemplo:

```
> x:=2;
```

x := 2

```
> cat("El número elegido es el ", x, ".");
```

"El número elegido es el 2."

Las cadenas cuentan además con el comando length que permite conocer la longitud de nuestra cadena.

```
> cad:= "Mi cadena de caracteres";
```

cad := "Mi cadena de caracteres"

```
> length(cad);
```

23

Para poder escribir cadenas de caracteres podemos utilizar también el comando print. Mediante este comando podemos escribir una frase e incluir en ella el valor de una variable. Para ello el texto debe ir entre dos tildes y separado de las variables mediante comas:

```
> x:=3;
```

x := 3

```
> print (`El valor de x es`, x);
```

El valor de x es, 3

Podemos observar que la coma sigue apareciendo en la salida. Para evitar que aparezca podemos emplear el comando printf con el cual primero tenemos que definir el formato. Hay diversos elementos que pueden ser modificados y para una información más detallada conviene utilizar el Help.

2.4.3. Constantes predefinidas

Maple cuenta con una serie de constantes redefinidas entre las que están el número **Pi**, la unidad imaginaria **I**, los valores **infinity** y **-infinity**, y las constantes booleanas **true** y **false**. Pare ver la lista de las constantes empleadas por Maple basta con hacer lo siguiente:

```
> constants;
```

2.4.4. Expresiones y ecuaciones

Una expresión en Maple es una combinación de números, variables y operadores. Los más importantes operadores binarios de Maple son los siguientes:

+	suma	>	mayor que
-	resta	>=	mayor o igual que
*	producto	=	igual
/	división	<>	no igual
^	potencia	:=	operador de asignación
**	potencia	and	and lógico
!	factorial	or	or lógico
mod	módulo	union	unión de conjuntos
<	menor que	intersect	intersección de conjuntos
<=	menor o igual que	minus	diferencia de conjuntos

Las reglas de precedencia de estos operadores son similares a las de C. En caso de duda, es conveniente poner paréntesis.

Puede asignar a cualquier expresión o ecuación un nombre de la forma siguiente:

```
> nombre:=expresion:
```

```
> ecn:=x+y=3;
```

$$ecn := x + y = 3$$

Finalmente, es de suma importancia distinguir una expresión de una función. A este cometido dedicaremos un apartado más adelante en el manual, dada su importancia.

2.4.5. Secuencias, Sets y Listas.

2.4.5.1 Secuencias.

Maple tiene algunos tipos de datos compuestos o estructurados que no existen en otros lenguajes y a los que hay que prestar especial atención. Entre estos tipos están las **secuencias (o sucesiones)**, los **conjuntos**, las **listas**, los **vectores** y **matrices** y las **tablas**.

La estructura de datos básica de Maple es la secuencia. Se trata de un conjunto de expresiones o datos de cualquier tipo separados por comas. Por ejemplo, se puede crear una secuencia de palabras y números de la forma siguiente:

```
> sec0 := enero, febrero, marzo, 22, 33;
```

$$sec0 := enero, febrero, marzo, 22, 33$$

Las secuencias son muy importantes en Maple. No son listas ni sets, por lo tanto conservan el orden en el que son introducidos los datos y a pesar de que haya dos iguales conserva ambos. Existen algunas formas o métodos especiales para crear secuencias automáticamente. Por ejemplo, el *operador dólar* ($) crea una secuencia repitiendo un nombre un número determinado de veces:

```
> sec1 := trabajo$5;
```

$$sec1 := trabajo, trabajo, trabajo, trabajo, trabajo$$

De modo complementario, el *operador dos puntos seguidos* (..) permite crear secuencias especificando rangos de variación de variables. Por ejemplo:

```
> sec2 := $1..10;
```

$$sec2 := 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

```
> sec3 := 'i'^2$'i'=1..8;
```

$$sec3 := 1, 4, 9, 16, 25, 36, 49, 64$$

donde es necesario poner los apóstrofes para evitar errores en el caso de que la variable **i** estuviese evaluada a algo distinto de su propio nombre (es decir, tuviera un valor numérico o simbólico previo).

Existe también una función llamada ***seq*** específicamente diseñada para crear secuencias. Véase el siguiente ejemplo:

```
> sec4 := seq(i!/i^2,i=1..8);
```

$$sec4 := 1, \frac{1}{2}, \frac{2}{3}, \frac{3}{2}, \frac{24}{5}, 20, \frac{720}{7}, 630$$

Puede comprobarse que utilizando la función ***seq*** no hace falta poner apóstrofos en la variable **i**, aunque esté evaluada a cualquier otra cosa.

¿Qué operaciones permite Maple hacer con secuencias? Al ser una clase de datos tan general, las operaciones son por fuerza muy elementales. Una posibilidad es crear una secuencia concatenando otras secuencias, como en el siguiente ejemplo:

```
> sec5 := sec0, sec1;
```

$$sec5 := enero, febrero, marzo, 22, 33, trabajo, trabajo, trabajo, trabajo, trabajo$$

Maple permite acceder a los elementos de una secuencia (al igual que en las cadenas de caracteres) por medio de los corchetes [ ], dentro de los cuales se puede especificar un elemento (empezando a contar por 1, no por 0 como en C) o un rango de elementos. Si el número del elemento es negativo se interpreta como si empezáramos a contar desde la derecha. Por ejemplo:

```
> sec5[3]; sec5[3..7];
```

$$marzo$$

$$marzo, 22, 33, trabajo, trabajo$$

Si aplicamos el operador de concatenación a una secuencia, la operación afecta a cada elemento. Por ejemplo si tenemos una secuencia sec6, podemos poner la letra a delante de cada elemento de ella concatenando de la forma siguiente:

```
> sec0:=1,2,3,4;
```

$$sec0 := 1, 2, 3, 4$$

```
> a||sec0;
```

$$a1, a2, a3, a4$$

```
> cat(a,sec0);                #No obtenemos el resultado deseado
```

$$a1234$$

Podemos crear también una secuencia aplicándole una función a cad uno de los elementos de una lista o un set.

```
> lista:=[1,2,4,6];           #Definición de una lista
```

$$lista := [1, 2, 4, 6]$$

```
> seq(x^2+1,x=lista);
```

$$2, 5, 17, 37$$

Maple dispone de la función ***whattype*** que permite saber qué tipo de dato es la variable que se le pasa como argumento. Pasándole una secuencia, la respuesta de esta función es ***exprseq***.

2.4.5.2 Sets.

En Maple se llama ***conjunto*** o ***set*** a una *colección no ordenada de expresiones diferentes*. Para evitar la ambigüedad de la palabra castellana *conjunto*, en lo sucesivo se utilizará la palabra inglesa ***set***. La forma de definir un *set* en Maple es mediante una secuencia encerrada entre llaves { } donde los elementos están separados por comas. Observe los siguientes ejemplos:

```
> set1 := {1,3,2,1,5,2};
```

$$set1 := \{1, 2, 3, 5\}$$

```
> set2 := {rojo, azul, verde};
```

$$set2 := \{rojo, verde, azul\}$$

Se puede observar que Maple elimina los elementos repetidos y cambia el orden dado por el usuario (el programa ordena la salida con sus propios criterios). Un *set* de Maple es pues un tipo de datos en el que no importa el orden y en el que no tiene sentido que haya elementos repetidos. Más adelante se verán algunos ejemplos. Una vez que Maple ha establecido un orden de salida, utilizará siempre ese mismo orden. Conviene recordar que para Maple el entero 2 es distinto de la aproximación de coma flotante 2.0. Así el siguiente set tiene tres elementos y no dos:

```
> {1,2,2.0};
```

$$\{1, 2, 2.0\}$$

Existen tres operadores que actúan sobre los *sets*: ***union***, ***intersect*** y ***minus***, que se corresponden con las operaciones algebraicas de unión, intersección y diferencia de conjuntos. Observe la salida del siguiente ejemplo:

```
> set3 := {rojo,verde,negro} union {amarillo,rojo,azul};
```

$$set3 := \{amarillo, rojo, verde, azul, negro\}$$

Al igual que con las secuencias, a los elementos de los *sets* se accede con el corchete [ ]. Existen además otras funciones que actúan sobre *sets* (pero no sobre secuencias), como son la función ***op*** que devuelve todos o algunos de los elementos del *set*, ***nops*** que devuelve el número de elementos. Véanse los siguientes ejemplos:

```
> op(set3); op(5,set3); op(2..4, set3); nops(set3);
```

$$amarillo, rojo, verde, azul, negro$$

$$negro$$

$$rojo, verde, azul$$

$$5$$

Hay que señalar que los datos devueltos por la función ***op*** son una secuencia. Otra cosa que debemos considera al emplear la función op es que debido al hecho de que el ordenador almacena los elementos en el orden que él elige puede que no resulte demasiado útil. Para realizar esta misma función existe el comando select, del que se hablará más adelante. Si se pasa un *set* como argumento a la función ***whattype*** la respuesta es ***set***.

2.4.5.3 Listas

Una ***lista*** es un *conjunto ordenado de expresiones o de datos contenido entre corchetes* [ ]. En las listas se respeta el orden definido por el usuario y puede haber elementos repetidos. En este sentido se parecen más a las secuencias que a los *sets*. Los elementos de una lista pueden ser también listas y/o *sets*. Observe lo que pasa al definir la siguiente lista de *sets* de letras:

```
> lista1 := [{p,e,r,r,o},{g,a,t,o},{p,a,j,a,r,o}];
```

$$lista1 := [\{p, e, r, o\}, \{t, a, g, o\}, \{j, a, p, r, o\}]$$

Como se ha visto, a las secuencias, *sets* y listas se les puede asignar un nombre cualquiera, aunque no es necesario hacerlo. Al igual que con las secuencias y *sets*, se puede acceder a un elemento particular de una lista por medio del nombre seguido de un índice entre corchetes. También se pueden utilizar sobre listas las funciones ***op*** y ***nops***, de modo semejante que sobre los *sets*.

```
> list1:=[1,2,4,6,9,10];
```

$$list1 := [1, 2, 4, 6, 9, 10]$$

```
> op(list1);
```

$$1, 2, 4, 6, 9, 10$$

```
> op(2..5,list1);
```

$$2, 4, 6, 9$$

Los elementos de una lista pueden ser a su vez otras listas como en el siguiente ejemplo:

```
> list0:=[1,[2,3],[4,[5,6],7],8,9];
```

$$list0 := [1, [2, 3], [4, [5, 6], 7], 8, 9]$$

Para acceder a los elementos de esta lista lo haremos de una de las siguientes formas:

```
> list0[3,2,2];
```

$$6$$

```
> list0[3][2][2];
```

$$6$$

Para poder desasignar una variable a la que se le ha signado una lista podemos hacerlo igual que con cualquier otro tipo e variables.

```
> list1:='list1';
```

$$list1 := list1$$

```
> list1;
```

$$list1$$

También podemos reasignar los elementos de una lista o directamente eliminarlos empleando el comando subsop.

```
> list1;
```

$$[1, 2, 4, 6, 9, 10]$$

```
> list1[3]:=5;
```

$$list1_3 := 5$$

```
> list1;
```

[1, 2, 5, 6, 9, 10]

```
> subsop(5=3,list1);
```

[1, 2, 5, 6, 3, 10]

```
> subsop(4=NULL,list1);
```

[1, 2, 5, 9, 10]

La respuesta de la función ***whattype*** cuando se le pasa una lista como argumento es ***list***. Por otra parte, los operadores ***union***, ***intersect*** y ***minus*** no operan sobre listas. Sin embargo si que se pueden concatenar listas o añadir elementos a una lista mediante el comando op.

```
> list1:=[martes, miércoles, jueves]:
> list2:=[viernes, sábado, domingo]:
> list3:=[op(list1),op(list2)];
```

list3 := [*martes*, *miércoles*, *jueves*, *viernes*, *sábado*, *domingo*]

```
> list4:=[lunes, op(list3)];
```

list4 := [*lunes*, *martes*, *miércoles*, *jueves*, *viernes*, *sábado*, *domingo*]

Tampoco se pueden utilizar operadores de asignación o aritméticos pues pueden no tener sentido según el tipo de los elementos de la lista.

Es muy importante distinguir, en los comandos de Maple que se verán más adelante, cuándo el programa espera recibir una ***secuencia***, un ***set*** o una ***lista***. Algo análogo sucede con la salida del comando.

La función ***type*** responde ***true*** o ***false*** según el tipo de la variable que se pasa como argumento coincida o no con el nombre del tipo que se le pasa como segundo argumento. Por ejemplo:

```
> type(set1,`set`);
```

true

```
> type(lista1,`set`);
```

false

2.4.6. Vectores y Matrices

Los vectores y las matrices son una extensión del concepto de la lista. Consideremos una lista como un grupo de elementos en el cual asociamos cada uno de ellos con un entero positivo, su índice, que representa su posición en la lista. El concepto de matriz de Maple es una generalización de esa idea. Cada elemento sigue asociado con un índice, pero una matriz no se limita a una dimensión.

La forma más directa de introducir un vector o una matriz es hacerlo a través de la paleta correspondiente. En ella seleccionamos la dimensión de nuestro objeto y a

continuación introducimos los datos. Si se trabaja de esta forma los índices siempre comenzarán en el uno.

Figura 1 - Vector

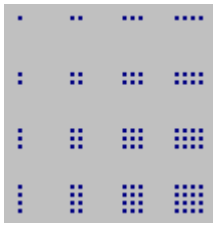


Figura 2 - Matriz

Podemos también introducir matrices mediante el comando Matrix del modo en el que se muestra en el siguiente ejemplo. En este caso también los índices empiezan en uno.

```
> matrix([[2,3],[5,6]]);
```

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

Otra forma de introducir vectores o matrices consiste en emplear el comando array. Al utilizar este comando los índices pueden valer cero o ser negativos.

Las matrices y vectores deben ser declarados. La asignación puede hacerse en la declaración o posteriormente. Veámoslo con dos ejemplos:

```
> cuadrados:=array(1..3, [1,4,9]); #declarando y asignando
```

cuadrados := [1, 4, 9]

```
> potencias:=array(1..3,1..3,[]); #declaramos una matriz 3x3
```

potencias := array(1 .. 3, 1 .. 3, [ ])

```
> potencias[1,1]:=1: potencias[1,2]:=1: potencias[1,3]:=1:
  potencias[2,1]:=2: potencias[2,2]:=4: potencias[2,3]:=8:
  potencias[3,1]:=3: potencias[3,2]:=9:potencias[3,3]:=27:
  #Asignación
```

Para poder ver los contenidos de un vector o una matriz no basta con poner su nombre, sino que hay que hacerlo mediante el comando **print()**.

```
> potencias;
```

potencias

```
> print(potencias);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 3 & 9 & 27 \end{bmatrix}$$

Hay que tener cuidado al sustituir en elemento por otro dentro de una matriz. Supongamos que queremos sustituir el 2 por un 9. Lo haremos mediante el comando **subs**.

Uno puede parecer extrañado cuando la llamada a subs siguiente no funciona:

```
> subs({2=9}, potencias);
```

$$potencias$$

```
> print(potencias);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 3 & 9 & 27 \end{bmatrix}$$

Lo que ocurre es que hay que hacer que Maple evalúe toda la matriz y no únicamente su nombre al llamar a la instrucción subs. Esto se logra utilizando el comando **evalm** (evaluar matriz). Así tenemos:

```
> subs({2=9}, evalm(potencias));
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 9 & 4 & 8 \\ 3 & 9 & 27 \end{bmatrix}$$

Para acceder a un elemento de la matriz lo hacemos a través de los índices, tal y como lo hacemos al trabajar con listas:

```
> m:=matrix ([[4,6],[5,7]]);
```

$$m := \begin{bmatrix} 4 & 6 \\ 5 & 7 \end{bmatrix}$$

```
> m[1,2];
```

$$6$$

2.4.7. Tablas

Una tabla es a su vez una extensión del concepto de matriz dentro de los objetos de Maple. La diferencia fundamental entre una matriz y una tabla es que esta segunda puede presentar como índice cualquier cosa, no solo enteros. A simple vista nos puede parecer que esto puede tener pocas ventajas sobre la estructura de la matriz, pero el trabajar con tablas nos permite utilizar una notación mucho más natural a la hora de manejarnos con los datos. Poniendo por ejemplo el caso de un modelo de vehículo:

```
>datos:=table([modelo=[LX85],potencia=[120,cv],
  precio=[1800,euros]]);
```

$$datos := \mathrm{table}([potencia = [120, cv], modelo = [LX85], precio = [1800, euros]])$$

```
> datos[potencia];
```

$$[120, cv]$$

En este caso cada índice es un nombre y cada entrada es una lista. Es un ejemplo más bien simple y a menudo índices mucho más generales son empleados como por ejemplo fórmulas algebraicas mientras que las entradas son sus derivadas.

2.4.8. Hojas de cálculo

Maple dispone entre sus herramientas hojas de cálculo con el formato tradicional, con la característica de que puede operar simbólicamente.

Se obtiene del menú ***Insert/Spreadsheet.*** Se abre entonces una ventana en la que hay que introducir el título de la hoja de trabajo. A continuación aparece una parte de la hoja de cálculo que puede hacerse más o menos grande clicando justo sobre el borde de la hoja. Se recuadrará en negro y clicando en la esquina inferior derecha y arrastrando puede modificarse el tamaño.

Podemos definir las propiedades de una celda o de un conjunto de celdas antes de empezar a trabajar con ellas. En primer lugar tenemos que seleccionar dichas celdas y clicar en Spreadsheet/Properties. Aparecerá de este modo un cuadro de diálogo en el cual podemos definir la alineación del texto, el color de fondo, la precisión, tanto a la hora de efectuar los cálculos como en el momento de mostrar los resultados, y el modo de evaluación (simbólico o de punto flotante). Podemos también introducir nuevas filas o columnas mediante el menú Spreadsheet/Row/Insert o Speadsheet/Column/Insert respectivamente. La nueva fila aparecerá encima de la línea en la que esté el cursor, y en el caso de la columna a la izquierda.

Algunas de las celdas pueden hacer referencia a otras celdas. Cada vez que una de las celdas de referencia varía las celdas que dependen de ella aparecen tachadas y deben ser recalculadas (más adelante se indicará cómo). El símbolo ~ (Alt+126) indica que una celda hace referencia a otra. Las referencias pueden ser de dos tipos distintos, relativas o absolutas. Para analizar la diferencia entre ambas utilizaremos los siguientes ejemplos. Si queremos que el valor de nuestra celda sea igual al valor de la celda que tiene a la izquierda más uno utilizaremos la referencia relativa, ya que si movemos la celda siempre hará referencia a la celda que se encuentra a su izquierda (figuras 3 y 4):

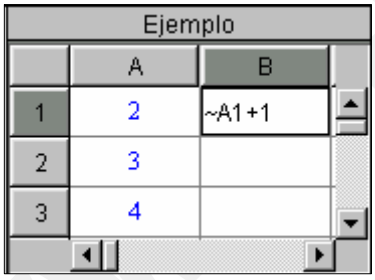


Figura 3

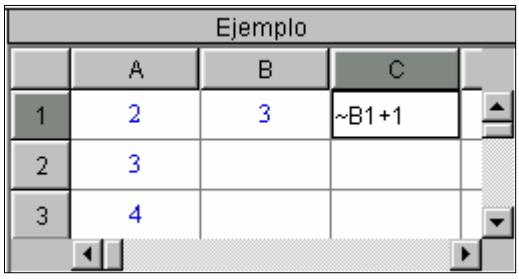


Figura 4

Sin embargo, si utilizamos una referencia absoluta la celda de referencia siempre será la misma independientemente de que copiemos nuestra celda o introduzcamos nuevas filas o columnas. Para crear una referencia absoluta es necesario anteponer el símbolo $ delante de la letra y el número que especifican la celda (figuras 5 y 6):

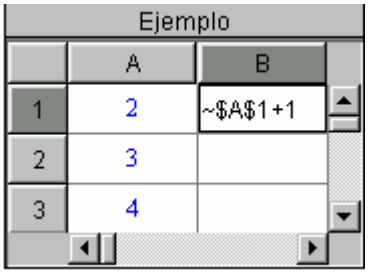


Figura 5

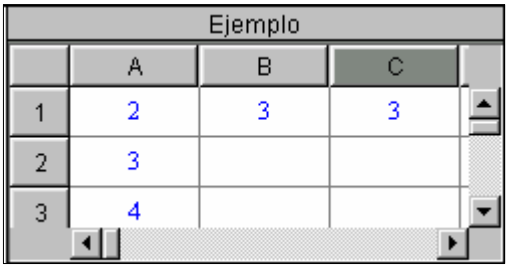


Figura 6

Se va a realizar a continuación un pequeño ejemplo que ayude a iniciar el manejo de estas hojas de cálculo. Una vez insertada la hoja de cálculo (*Spreadsheet*) en la hoja de trabajo (*worksheet*) tal como se ha indicado antes, modifique el tamaño hasta que sea de cuatro filas y cuatro columnas o mayor. En la casilla 'A1' teclee un 1 y pulse intro. Seleccione las cuatro primeras casillas de la primera columna y clique el botón.

Introduzca el valor 1 en ***Step Size*** del cuadro de diálogo que aparece. La hoja de cálculo queda de la siguiente manera:

	A	B	C	D
1	1			
2	2			
3	3			
4	4			

En la casilla 'B1' teclee: x^(~A1). Con (~A1) nos referimos a la casilla 'A1'—el símbolo ~ se puede obtener tecleando **126** mientras se mantiene pulsada la tecla **Alt** o pulsando **Alt Gr** a la vez que el 4 y luego pulsando la barra espaciadora—. Seleccione las cuatro primeras casillas de la segunda columna y repita el proceso anterior con el botón. En la casilla 'C1' teclee: int(~B1,x). En la 'D1' teclee: diff(~C1,x) y arrastre de nuevo las expresiones hacia abajo. La hoja de cálculo que se obtiene es la siguiente:

	A	B	C	D
1	1	x	$\frac{x^2}{2}$	x
2	2	x^2	$\frac{x^3}{3}$	x^2
3	3	x^3	$\frac{x^4}{4}$	x^3
4	4	x^4	$\frac{x^5}{5}$	x^4

Si se modifica alguna casilla de la hoja de cálculo que afecte a otras casillas, las casillas afectadas aparecerán tachadas. Para recalcular toda la hoja se utiliza el botón,

o se pueden seleccionar las celdas que hayan sido afectadas y posteriormente clicar Spreadsheet/EvaluateSelection.

Asimismo, existe una forma de compatibilizar las hojas de cálculo de Microsoft Excel con las de Maple 8. Para ello es necesario activar en Excel el add-in de Maple. Para ello habrá que ir al menú ***Tools/Add-Ins*** de Excel (figura 7) y ahí veremos una opción llamada *Maple Excel Add-In* que tendremos que seleccionar.

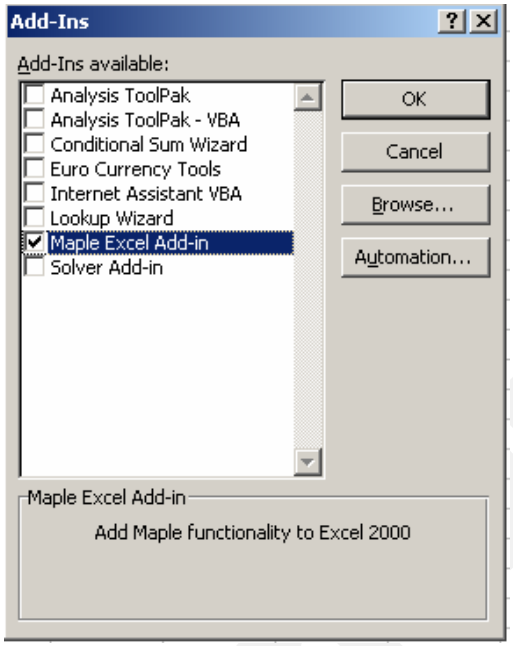


Figura 7

Una vez seleccionado aparecerán en la barra de herramientas de Excel unos iconos que se emplean para poder copiar hojas de cálculo de Excel a Maple y viceversa y manejar el programa Maple desde el Excel. Asimismo existe un icono con un signo de interrogación (figura 8) que nos ayuda a iniciarnos en el trabajo de Excel con Maple.

Figura 8

2.5. FUNCIONES MEMBER, SUBSOP, SUBS, SELECT Y REMOVE.

Se trata de funciones muy utilizadas con las listas y con los sets. La función ***member*** actúa sobre sets y listas, pero no sobre secuencias. Su finalidad es averiguar si un objeto (sea un dato o una expresión) pertenece a un set o una lista. La función tiene tres argumentos (member(x, s, 'p')): la expresión o dato x, el set o lista en el que queremos buscar, s, y una variable no evaluada, 'p', que es opcional y cuya finalidad es almacenar la posición del objeto dado dentro del set o lista. Lo veremos en el ejemplo siguiente:

```
> set1:={x,yuv,zxc,t,us,v};
```

$$set1 := \{x, v, t, yuv, zxc, us\}$$

```
> member(k,set1);
```

false

```
> member(t,set1,'pos');
```

true

```
> pos;
```

$$3$$

Un elemento de una lista se puede cambiar de formas distintas. Una de ellas es utilizando el operador de asignación dirigiéndonos al elemento de dicha lista. Así tendremos:

```
> list2:=[amarillo,rojo,verde,azul,negro];
```

$$list2 := [amarillo, rojo, verde, azul, negro]$$

```
> list2[3]:=blanco;
```

$$list2_3 := blanco$$

```
> list2;
```

$$[amarillo, rojo, blanco, azul, negro]$$

Otra forma de hacer una operación equivalente es utilizando el comando ***subsop*** que realiza la sustitución de un elemento por otro en la lista y que además es mucho más amplio ya que es también válido para expresiones de todo tipo.

```
> list2:=[amarillo,rojo,verde,azul,negro];
```

$$list2 := [amarillo, rojo, verde, azul, negro]$$

```
> subsop(3=blanco, list2);
```

$$[amarillo, rojo, blanco, azul, negro]$$

```
> pol1:=8*x^3+3*x^2+x-8; # cambiando el signo al 2º término
```

$$pol1 := 8x^3 + 3x^2 + x - 8$$

```
> subsop(2=-op(2,pol1),pol1);
```

$$8x^3 - 3x^2 + x - 8$$

Si en vez de querer reemplazar una posición se desea reemplazar un valor en toda la lista por otro, puede usarse la función ***subs***, como en el ejemplo siguiente, donde cambiamos el valor “negro” por “blanco” en toda la lista:

```
> list3:=[op(list2),negro];
```

$$list3 := [amarillo, rojo, verde, azul, negro, negro]$$

```
> subs(negro=blanco, list3);
```

$$[amarillo, rojo, verde, azul, blanco, blanco]$$

Existe también la función algsubs que resulta más potente, tal y como se muestra en el siguiente ejemplo:

```
> subs(x^2=y, x^3);
```

$$x^3$$

```
> algsubs(x^2=y,x^3);
```

$$y\,x$$

Asimismo son útiles en el trabajo con listas las funciones ***select*** y ***remove***. Nos permiten seleccionar ciertos elementos de una lista según satisfagan o no un criterio especificado. Lo veremos en el ejemplo siguiente en el que el criterio especificado es que el número sea mayor que 3:

```
> mayor:=x->is(x>3); #definimos una función booleana que nos dirá
si es mayor o no
```

$$mayor := x \rightarrow \text{is}(3 < x)$$

Definimos una lista y seleccionamos los elementos que cumplan la condición:

```
> list4:=[2,5,Pi,-1,6.34];
```

$$list4 := [2, 5, \pi, -1, 6.34]$$

```
> select(mayor,list4);
```

$$[5, \pi, 6.34]$$

Asimismo, podremos mediante el comando ***remove*** eliminar de la lista los elementos que satisfagan dicha condición:

```
> remove(mayor,list4);
```

$$[2, -1]$$

Para realizar las dos operaciones simultáneamente existe el comando ***selectremove***:

```
> selectremove(mayor,list4);
```

$$[5, \pi, 6.34], [2, -1]$$

Otro ejemplo en el cual pueden resultar útiles las funciones Select y Remove es el caso en el que queramos separar en una expresión los elementos que dependen de x y los que no:

```
> h:= 3*x+x*y+x^3+4*y^2;
```

$$h := 3x + xy + x^3 + 4y^2$$

```
> select (has,h,x);
```

$$3x + xy + x^3$$

```
> remove(has,h,x);
```

$$4y^2$$

```
> selectremove(has,h,x);
```

$$3x + xy + x^3, 4y^2$$

2.6. LOS COMANDOS ZIP Y MAP

El comando ***zip*** nos permite aplicar una función binaria f a los elementos de dos listas o vectores (u, v) creando una nueva lista, r, o vector definida de la siguiente forma: su tamaño será igual al menor de los tamaños de las listas originales y cada elemento de la

nueva lista tendrá un valor r[i]=f(u[i],v[i]). Si proporcionamos a la función ***zip*** un cuatro argumento extra, éste será tomado como argumento por defecto cuando una de las listas o vectores sea menor que el otro y entonces la longitud de la lista resultante será igual a longitud de la mayor de las listas originales. Veamos varios ejemplos:

```
> funcion:=(x,y)->x+y; #función
```

$$funcion := (x, y) \rightarrow [x + y]$$

```
> X:=[1,2,3]; Y:=[4,5,6];
```

$$X := [1, 2, 3]$$

$$Y := [4, 5, 6]$$

```
> P:=zip(funcion,X,Y);
```

$$P := [[5], [7], [9]]$$

```
> zip((x,y)->(x*y), [1,2,3],[4,5,6,7,8]); #sin cuarto argumento
```

$$[4, 10, 18]$$

```
> zip((x,y)->(x*y), [1,2,3],[4,5,6,7,8],1); #con cuarto argumento
```

$$[4, 10, 18, 7, 8]$$

Este comando puede ser también muy útil cuando necesitamos unir dos listas. Por ejemplo, supongamos que tenemos una lista que representa las coordenadas x de unos puntos y otra que representa las coordenadas y, y obtener una nueva lista de la forma [[x1,y1], [x2,y2], …] para poder luego representarlos. El comando ***zip*** nos es aquí de gran utilidad. Veámoslo en el ejemplo siguiente:

```
> relacion:=(x,y)->[x,y];
```

$$relacion := (x, y) \rightarrow [x, y]$$

```
> X:=[ seq(2*i/(i+1), i=1..8)];
```

$$X := \left[1, \frac{4}{3}, \frac{3}{2}, \frac{8}{5}, \frac{5}{3}, \frac{12}{7}, \frac{7}{4}, \frac{16}{9}\right]$$

```
> Y:=[seq(ithprime(i), i=1..8)];
```

$$Y := [2, 3, 5, 7, 11, 13, 17, 19]$$

```
> P:=zip(relacion, X, Y);
```

$$P := \left[[1, 2], \left[\frac{4}{3}, 3\right], \left[\frac{3}{2}, 5\right], \left[\frac{8}{5}, 7\right], \left[\frac{5}{3}, 11\right], \left[\frac{12}{7}, 13\right], \left[\frac{7}{4}, 17\right], \left[\frac{16}{9}, 19\right]\right]$$

```
> plot(P);
```

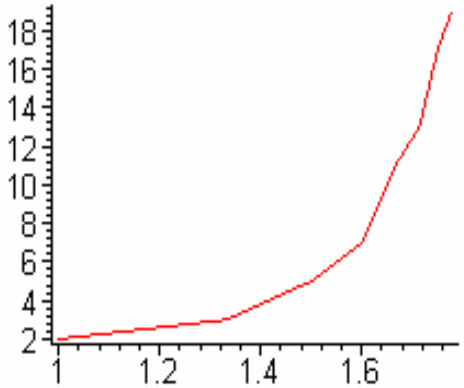


Figura 9

El comando ***map*** es muy útil en Maple. Permite aplicar una misma función a todos los elementos o expresiones de una lista o conjunto. El ejemplo más sencillo de su funcionamiento es el siguiente:

```
> lista:=[a,b,c];
```

$$lista := [a, b, c]$$

```
> map(f,lista);
```

$$[f(a), f(b), f(c)]$$

Así si, por ejemplo, queremos calcular las derivadas de una lista de expresiones tendremos que aplicar mediante ***map*** el comando ***diff*** a dicha lista. Veamos un ejemplo:

```
> lista2:=[x^2+x+2,sin(x),exp(2*x)];
```

$$lista2 := [x^2 + x + 2, \sin(x), \mathbf{e}^{(2x)}]$$

```
> map(diff,lista2,x);
```

$$[2x + 1, \cos(x), 2\,\mathbf{e}^{(2x)}]$$

En este caso hemos tenido que pasar el argumento extra 'x' para especificar la variable respecto a la que se quiere derivar

2.7. FUNCIONES DEFINIDAS MEDIANTE EL OPERADOR FLECHA (->)

2.7.1. Funciones de una variable

Cuando usamos Maple, las relaciones funcionales se pueden definir de dos modos distintos:

- mediante una expresión o fórmula
- mediante una función matemática propiamente dicha (y definida como tal)

Veremos cuanto antes un ejemplo que nos permita entender las diferencias más importantes entre una función y una expresión. Empezaremos definiendo, por ejemplo, la tensión en los bornes de un condensador en descarga, en función del tiempo, mediante una expresión:

```
> V:=V0*exp(-t/tau);
```

$$V := V0\,\mathbf{e}^{\left(-\frac{t}{\tau}\right)}$$

Si ahora queremos dar a la variable **t** un valor, para poder evaluar la tensión en ese instante, tendremos que utilizar el comando **subs** o bien cambiar el valor de las variables que intervienen en la expresión. Veremos las dos formas:

```
> subs(t=tau,V);t; #tras la operación t sigue sin estar asignada
```

$$V0\,\mathbf{e}^{(-1)}$$

$$t$$

```
> t:=tau;V;
```

$$t := \tau$$

$$V0\ \mathbf{e}^{(-1)}$$

Estos métodos han funcionado de manera correcta y hemos obtenido el resultado que esperábamos, pero en este caso **V** no es una función del tiempo propiamente dicha. En el ejemplo anterior **t** interviene del mismo modo que interviene **tau** o V0 (podríamos haber asignado de la misma forma un valor a tau). Si queremos que nuestra función **V** sea una verdadera función del tiempo **t** tendremos que definirla mediante el **operador flecha (->)** que exige una sintaxis bien definida:

```
> V:=t->V0*exp(-t/tau);
```

$$V := t \rightarrow V0\ \mathbf{e}^{\left(-\frac{t}{\tau}\right)}$$

Ahora es mucho más sencillo obtener el valor de **V** para cualquier valor de **t** deseado (**t** interviene de modo distinto que **V0** o **tau** en la función, ahora **V** es *función de t*). Por ejemplo:

```
> V(0);V(tau);
```

$$V0$$

$$V0\ \mathbf{e}^{(-1)}$$

Las funciones definidas con el operador flecha se evalúan a su propio nombre. Sin embargo, si se escribe la función seguida de la variable entre paréntesis, se obtiene la expresión de la función completa:

```
> V;V(t);
```

$$V$$

$$V0\ \mathbf{e}^{\left(-\frac{t}{\tau}\right)}$$

Muchas veces nos encontramos ante funciones definidas por tramos. Para poder introducir este tipo de funciones en Maple existe el comando ***piecewise***. Veamos su uso en un ejemplo:

```
> f:=x->piecewise(x<=1, x+1,1<x and x<3,sin(x),1);
```

$$f := x \rightarrow \text{piecewise}(x \le 1, x+1, 1 < x\ \mathbf{and}\ x < 3, \sin(x), 1)$$

```
> f(x);
```

$$\begin{cases} x+1 & x \le 1 \\ \sin(x) & -x < -1\ \mathbf{and}\ x < 3 \\ 1 & otherwise \end{cases}$$

```
> plot(f(x),x=-1..4);
```

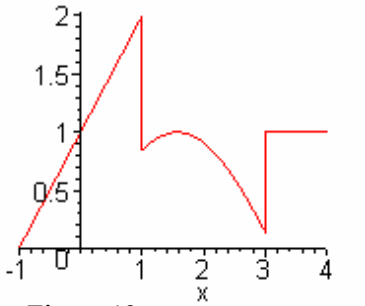


Figura 10

2.7.2. Funciones de dos variables

Las funciones de dos o más variables se definen también utilizando el operador flecha visto en la sección anterior. Veamos un ejemplo:

```
> f:=(x,y)->x^3-3*x*y^2;
```

$$f := (x, y) \to x^3 - 3\,x\,y^2$$

```
> f(3,2); #evaluamos la función para valores determinados de sus variables
```

$$-9$$

```
> plot3d(f(x,y),x=-1..1,y=-1..1,axes=FRAME,style=PATCH);
```

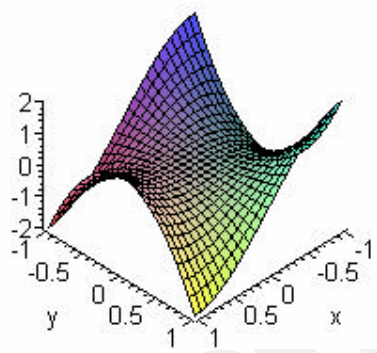


Figura 11

2.7.3. Conversión de expresiones en funciones

Hemos visto que Maple maneja expresiones y funciones de manera intercambiable a veces y muy diferente en otras. Las funciones son un poco más complicadas de definir, pero tienen muchas ventajas en algunos casos, como en la representación gráfica. Muchas veces nos encontramos ante una expresión y nos gustaría convertirla en una función. Para ello Maple dispone del comando ***unapply*** que puede convertir una expresión o fórmula en una función. Veamos cómo funciona este comando con un ejemplo:

```
> expresion:=(a^2*x^3+b*exp(t)+c^3*sin(x))/(a*x^2+c*t);
```

$$expresion := \frac{a^2\,x^3 + b\,\mathbf{e}^t + c^3\sin(x)}{a\,x^2 + c\,t}$$

```
> f:=unapply(expresion,x,t); #especificamos como argumento las variables sobre las que deseamos crear la funcion
```

$$f := (x, t) \to \frac{a^2\,x^3 + b\,\mathbf{e}^t + c^3\sin(x)}{a\,x^2 + c\,t}$$

```
> f(0,1);
```

$$\frac{b\,\mathbf{e}}{c}$$

Este tipo de conversión no puede hacerse directamente, con el operador flecha. Pruebe a ejecutar las sentencias siguientes y observe el resultado:

```
> g:=(x,t)->expresion;
```

$$g := (x, t) \rightarrow expresion$$

```
> f(u,v),g(u,v); #la u y la v no aparecen por ninguna parte en g
```

$$\frac{a^2 u^3 + b\,\mathbf{e}^v + c^3 \sin(u)}{a\,u^2 + c\,v}, \frac{a^2 x^3 + b\,\mathbf{e}^t + c^3 \sin(x)}{a\,x^2 + c\,t}$$

La única alternativa para obtener el mismo resultado que con la función ***unapply*** está basada en la función ***subs*** y es la siguiente:

```
> h:=subs(body=expresion, (x,t)->body);
```

$$h := (x, t) \rightarrow \frac{a^2 x^3 + b\,\mathbf{e}^t + c^3 \sin(x)}{a\,x^2 + c\,t}$$

```
> h(u,v); #ahora si que funciona
```

$$\frac{a^2 u^3 + b\,\mathbf{e}^v + c^3 \sin(u)}{a\,u^2 + c\,v}$$

El comando unapply puede resultar también útil cuando queremos crear una función que sea la derivada de otra. Si intentamos hacerlo empleando el operador flecha obtenemos lo siguiente:

```
> y:=4*x+cos(2*x);
```

$$y := 4\,x + \cos(2\,x)$$

```
> f:=x->diff(y,x);
```

$$f := x \rightarrow \frac{\partial}{\partial x} y$$

```
> f(1);
```

```
Error, (in f) invalid input: diff received 1, which is not valid
for its 2nd argument
```

Maple encuentra un error porque primero sustituye el valor de x en la función y luego intenta hacer la derivada. Sin embargo al utilizar unapply primero se hace la derivada y luego la expresión obtenida se convierte en función**:**

```
> y:=4*x+cos(2*x);
```

$$y := 4\,x + \cos(2\,x)$$

```
> f:=unapply(diff(y,x),x);
```

$$f := x \rightarrow 4 - 2\sin(2\,x)$$

```
> f(1);
```

$$4 - 2\sin(2)$$

2.7.4. Operaciones sobre funciones

Es fácil realizar con Maple operaciones tales como ***suma***, ***multiplicación*** y ***composición*** de funciones. Considérense los siguientes ejemplos:

```
> f := x -> ln(x)+1;  g := y -> exp(y)-1;
```

$$f := x \to \ln(x) + 1$$

$$g := y \to e^{y} - 1$$

```
> h := f+g;  h(z);
```

$$h := f + g$$

$$\ln(z) + e^{z}$$

```
> h := f*g;  h(z);
```

$$h := f\,g$$

$$(\ln(z) + 1)\left(e^{z} - 1\right)$$

La siguiente función define una ***función de función*** (composición de funciones) por medio del operador @ (el resultado es f(g)):

```
> h := f@g;  h(z);
```

$$h := f@g$$

$$\ln\left(e^{z} - 1\right) + 1$$

Considérese ahora el siguiente ejemplo, en el que el resultado es g(f):

```
> h := g@f; h(z);
```

$$h := g@f$$

$$e^{(\ln(z) + 1)} - 1$$

```
> simplify(%);
```

$$z\,e - 1$$

```
> (f@@4)(z);  # equivalente a f(f(f(f(z))));
```

$$\ln(\ln(\ln(\ln(z) + 1) + 1) + 1) + 1$$

El operador @ junto, con los ***alias*** y las ***macros***, es una forma muy potente de introducir abreviaturas en Maple.

Si se desea evaluar el resultado de la sustitución, ejecútese el siguiente ejemplo:

```
> n:='n'; Zeta(n); subs(n=2, Zeta(n));  # versión estándar de
subs()
> macro(subs = eval@subs);  # nueva versión de subs definida como
macro
```

```
> subs(n=2, Zeta(n));
```

$$n := n$$

$$\zeta(n)$$

$$\zeta(2)$$

$$subs$$

$$\frac{\pi^2}{6}$$

$$\frac{\pi^2}{6}$$

3- CÁLCULO BÁSICO CON MAPLE

3.1. OPERACIONES BÁSICAS

Antes de adentrarnos en las posibilidades que nos ofrece Maple es conveniente recordar que si trabajamos en modo *Maple Notation* (por defecto), tendremos que acabar todas nuestras sentencias mediante un carácter punto y coma (;) o dos puntos (:) según queramos que el programa nos saque en pantalla le resultado de nuestra operación o no. De hecho, si no ponemos estos caracteres de terminación y pulsamos Intro, el programa seguirá esperando a que introduzcamos más sentencia y completemos la instrucción (nos dará únicamente un *warning*). Puede también ser de utilidad recordar que para acceder al último resultado se puede utilizar el carácter porcentaje (%). De forma análoga, (%%) representa el penúltimo resultado y (%%%) el antepenúltimo. Es útil para poder emplear un resultado en el comando siguiente sin necesidad de asignarlo a una variable.

Maple puede funcionar como una calculadora convencional manejando enteros y números de coma flotante. Veamos algunos ejemplos:

```
> 1+2,76-4,5*3,120/2,54/7-6/4;
```

$$3,\ 72,\ 15,\ 60,\ \frac{87}{14}$$

Ahora bien si hacemos:

```
> sin(5.25/8*Pi);
```

$$\sin(0.6562500000\,\pi)$$

Vemos que en realidad no ha hecho lo que esperábamos que hiciera. Esto ocurre porque Maple intenta siempre no cometer errores numéricos (errores de redondeo en las operaciones aritméticas) y la mejor forma de evitarlo es dejar para más adelante las computaciones aritméticas. En este caso Maple ha efectuado la división (que no nos introduce error) pero no ha computado ni el valor de p ni el seno del resultado. Además el hecho de representar las expresiones de forma exacta nos permite conservar mucha más información sobre sus orígenes y estructuras. Por ejemplo 0.5235987758 es mucho menos claro para el usuario que el valor p/6. Eso sí, habrá que distinguir entre el entero 3 y su aproximación a coma flotante 3.0 ya que según introduzcamos uno u otro, Maple efectuará o no las operaciones aritméticas inmediatamente:

```
> 3^(1/2);
```

$$\sqrt{3}$$

```
> 3.0^(1/2);
```

$$1.732050808$$

Las operaciones aritméticas cuando trabajemos con enteros serán realizadas cuando el usuario lo decida, por ejemplo mediante el comando ***evalf.***

$$0.8819212643$$

```
> evalf(sin(5.25/8*Pi));
```

Maple permite controlar fácilmente la precisión con la que se está trabajando en los cálculos. Por defecto calcula con 10 dígitos decimales, pero este valor puede ser fácilmente modificado:

```
> Digits:=30;
```

$$Digits := 30$$

```
> evalf(sin(5.25/8*Pi));
```

$$0.881921264348355029712756863659$$

```
> Digits:=10; #devolvemos al valor por defecto
```

$$Digits := 10$$

```
> evalf(sin(5.25/8*Pi),30); #se puede pasar como argumento a eval
```

$$0.881921264348355029712756863659$$

El poder trabajar con cualquier número de cifras decimales implica que Maple no utilice el procesador de coma flotante que tiene el PC, sino que realiza esas operaciones por software, con la consiguiente pérdida de eficiencia. Si queremos que el programa utilice el procesador de coma flotante del PC podremos utilizar el comando ***evalhf*** que gana en velocidad pero no en precisión.

Además de las operaciones aritméticas básicas como suma o producto, Maple dispone de las funciones matemáticas más utilizadas. Nombraremos las más comunes:

FUNCIÓN	DESCRIPCIÓN
sin, cos, tan, etc	Funciones trigonométricas
sinh, cosh, tanh, etc	Funciones trigonométricas hiperbólicas
arcsin, arccos, arctan, etc	Funciones trigonométricas inversas
exp	Función exponencial
ln	Logaritmo neperiano
log[n]	Logaritmo en base n
sqrt	Raíz cuadrada
round	Redondeo al entero más próximo
trunc	Truncamiento a la parte entera
frac	Parte decimal
BesselI, BesselJ, BesselK, BesselY	Funciones de Bessel
binomial	Coeficientes del binomio de Newton
Heaviside	Función escalón de Heaviside
Dirac	Función delta de Dirac
Zeta	Función Zeta de Riemann

Maple es también capaz de operar con números complejos. **I** es el símbolo por defecto de Maple para designar la unidad imaginaria. Veamos algunos ejemplos de operaciones:

```
> (2+5*I)+(4+I);
```

$$6 + 6\,I$$

```
> 2*I*(5+3*I);
```

$$-6 + 10\,I$$

```
> (2+5*I)/(4+I);
```

$$\frac{13}{17} + \frac{18}{17} I$$

Para operar sobre números complejos disponemos de una serie de

comandos. El comando ***Re*** nos devuelve la parte real del número y el ***Im*** la imaginaria, mientras que el comando ***conjugate*** devuelve el conjugado del mismo:

```
> num:=4+3*I;Re(num); Im(num); conjugate(num);
```

$$num := 4 + 3\,I$$

$$4$$

$$3$$

$$4 - 3\,I$$

También podemos utilizar la función ***Complex*** para escribir numeros complejos. La forma sería la siguiente**:** `Complex(real, imaginario)`

```
> Complex(4,3);
```

$$4 + 3\,I$$

```
> Complex(2); # si solo escribimos un numero lo supone imaginario
```

$$2\,I$$

Si queremos conocer el módulo y el argumento del número complejo disponemos de los comandos ***abs*** y ***argument*** respectivamente:

```
> abs(num); argument(num);
```

$$\sqrt{34}$$

$$\arctan\left(\frac{3}{5}\right)$$

Tenemos además la función *polar*, la cual nos permite pasar a forma polar el numero complejo.

```
> polar(num);
```

$$\mathrm{polar}\left(5, \arctan\left(\frac{3}{4}\right)\right)$$

Finalmente disponemos del comando ***evalc*** que nos permite descomponer una expresión en su parte real y su parte imaginaria. Lo veremos en un ejemplo:

```
> evalc(cos(num));
```

$$\cos(4)\cosh(3) - I\sin(4)\sinh(3)$$

3.2. TRABAJANDO CON FRACCIONES Y POLINOMIOS

3.2.1. Polinomios de una y más variables

3.2.1.1 Polinomios de una variable

Se llama **forma canónica** de un polinomio a la forma siguiente:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

donde **n** es el grado, $\mathbf{a_n}$ el primer coeficiente y $\mathbf{a_0}$ el último.

Se llama forma agrupada (collected form) de un polinomio, a la forma en la que todos los coeficientes de cada potencia de x están agrupados. Los términos no tienen por qué estar necesariamente ordenados por grado descendente. Introduzca el siguiente ejemplo:

```
> poli:=5*x^2-x+2*x^3 +8;  # no tiene por qué escribirlo en orden descendente
```

$$poli := 5x^2 - x + 2x^3 + 8$$

En el caso de que quiera ordenar el polinomio de forma descendente, puede hacer uso de la función *sort* (se explica en el siguiente tema).

```
> sort(poli);  # solo le pasamos como argumento el polinomio a ordenar
```

$$2x^3 + 5x^2 - x + 8$$

La función sort implica un cambio en la estructura interna del polinomio, más en concreto en la llamada tabla de simplificación, que es una tabla de subexpresiones que Maple crea para almacenar factores comunes, expresiones intermedias, etc., con objeto de ahorrar tiempo en los cálculos. Considérense los ejemplos siguientes:

```
> p := 1 + x + x^3 + x^2;  # términos desordenados
```

$$p := 1 + x + x^3 + x^2$$

```
> x^3 + x^2 + x + 1;  # los ordenará igual que en el caso anterior
```

$$1 + x + x^3 + x^2$$

```
> q := (x - 1)*(x^3 + x^2 + x + 1);
```

$$q := (x-1)\left(1 + x + x^3 + x^2\right)$$

Puede verse que en este último ejemplo Maple aprovecha la entrada anterior en la tabla de simplificación. Si se ordena p el resultado afecta al segundo factor de q:

```
> sort(p);  # cambia el orden de p
```

$$x^3 + x^2 + x + 1$$

```
> q;  # el orden del 2º factor de q ha cambiado también
```

$$(x-1)\left(x^3 + x^2 + x + 1\right)$$

En Maple existen algunas funciones para manipular polinomios, tales como coeff, coeffs, degree, ldegree, CoefficientList y CoefficientVector. Para entender su función observen los siguientes ejemplo:

```
> p:=x^2+3*x -6;
```

$$p := x^2 + 3x - 6$$

```
> coeff(p,x,2); # como argumentos se le pasan el polinomio, la
variable, y el grado de la variable de la cual quieren que les
devuelva el coeficiente
```

$$1$$

```
> coeffs(p); #  devuelve todos los coeficientes
```

$$-6, 3, 1$$

```
> degree(p); # devuelve el grado del polinomio
```

$$2$$

```
> ldegree(p); # devuelve el menor grado del polinomio
```

$$0$$

Para utilizar los comandos CoefficientList y CoefficientVector debemos añadir "PolynomialTools". Observen:

```
> with(PolynomialTools):
> p:=x^2+3*x -6;
```

$$p := x^2 + 3x - 6$$

```
> CoefficientList(p,x); # crea una lista de coeficientes
```

$$[-6, 3, 1]$$

```
> CoefficientVector(p,x); # crea un vector de coeficientes
```

Una de las operaciones más importantes con polinomios es la división, es decir, el cálculo del cociente y del resto de la división. Para ello existen las funciones **quo** y **rem**:

```
> p1:=x^3+4*x; p2:=x^2+2*x+6;
```

$$p1 := x^3 + 4x$$

$$p2 := x^2 + 2x + 6$$

```
> c:=quo(p1,p2,x,'r');r;  # calcula cociente y resto
```

$$c := x - 2$$

$$2x + 12$$

```
> testeq(p1=expand(c*p2+r)); # comprobación del resultado
```

$$true$$

```
> rem(p1,p2,x,'c');c;  # calcula el resto y el cociente
```

$$2x + 12$$

$$x - 2$$

La función divide devuelve true cuando la división entre dos polinomios es exacta (resto cero), y false si no lo es.

```
> divide(p2,p1);
```

$$false$$

Para calcular el máximo común divisor de dos polinomios se utiliza la función gcd:

```
> gcd(p1, p2);
```

Los polinomios también se pueden sumar, restar y multiplicar como cualquier otro tipo de variables.

```
> suma:= p1+p2;
```

$$suma := x^3 + 6\,x + x^2 + 6$$

```
> resta:=p1-p2;
```

$$resta := x^3 + 2\,x - x^2 - 6$$

```
> producto:=p1*p2;
```

$$producto := (x^3 + 4\,x)\,(x^2 + 2\,x + 6)$$

Podemos utilizar la función *expand* (función que se explica en el siguiente tema) para expandir el producto.

```
> expand(producto);
```

$$x^5 + 2\,x^4 + 10\,x^3 + 8\,x^2 + 24\,x$$

Finalmente, podemos hallar las raíces y factorizar (escribir el polinomio como producto de factores irreducibles con coeficientes racionales). Para ello utilizaremos las funciones **roots** y **factor,** respectivamente.

```
> pol:=expand(p1*p2);
```

$$pol := x^5 + 2\,x^4 + 10\,x^3 + 8\,x^2 + 24\,x$$

```
> roots(pol);
```

$$[[0, 1]]$$

```
> factor(pol);
```

$$x\,(x^2 + 2\,x + 6)\,(x^2 + 4)$$

Si ha ejecutado el ejemplo anterior, habrá comprobado que la función roots sólo nos ha devuelto 2 raíces, cuando el polinomio poli es de grado 9. La razón es que roots calcula las raíces en el campo de los racionales. La respuesta viene dada como una lista de pares de la forma [[r1,m1], ..., [rn,mn]], donde ri es la raíz y mi su multiplicidad. La función roots también puede calcular raíces que no pertenezcan al campo de los racionales, siempre que se especifique el campo de antemano. Introduzca el siguiente ejemplo:

```
> roots(x^4-4,x);#No devuelve ninguna raíz exacta racional
```

$$[\ ]$$

```
> roots(x^4-4, sqrt(2));#Devuelve 2 raíces reales irracionales
                    [[sqrt(2), 1], [-sqrt(2), 1]]

> roots(x^4-4, {sqrt(2),I});#Devuelve las 4 raíces del polinomio
  [[sqrt(2), 1], [-sqrt(2), 1], [I*sqrt(2), 1], [-I*sqrt(2), 1]]
```

3.2.1.2 Polinomios de varias variables

Maple trabaja también con polinomios de varias variables. Por ejemplo, se va a definir un polinomio llamado **poli**, en dos variables **x** e **y**:

```
> poli := 6*x*y^5 + 12*y^4 + 14*x^3*y^3 - 15*x^2*y^3 +9*x^3*y^2 -
30*x*y^2 - 35*x^4*y + 18*y*x^2 + 21*x^5;
```

$$poli := 6\,x\,y^5 + 12\,y^4 + 14\,x^3\,y^3 - 15\,x^2\,y^3 + 9\,x^3\,y^2 - 30\,x\,y^2 - 35\,x^4\,y + 18\,y\,x^2 + 21\,x^5$$

Se pueden ordenar los términos de forma alfabética (en inglés, pure lexicographic ordering):

```
> sort(poli, [x,y], 'plex');
```

$$21\,x^5 - 35\,x^4\,y + 14\,x^3\,y^3 + 9\,x^3\,y^2 - 15\,x^2\,y^3 + 18\,x^2\,y + 6\,x\,y^5 - 30\,x\,y^2 + 12\,y^4$$

o con la ordenación por defecto, que es según el grado de los términos:

```
> sort(poli);
```

$$14\,x^3\,y^3 + 6\,x\,y^5 + 21\,x^5 - 35\,x^4\,y + 9\,x^3\,y^2 - 15\,x^2\,y^3 + 12\,y^4 + 18\,x^2\,y - 30\,x\,y^2$$

Para ordenar según las potencias de x:

```
> collect(poli, x);
```

$$21\,x^5 - 35\,x^4\,y + (14\,y^3 + 9\,y^2)\,x^3 + (18\,y - 15\,y^3)\,x^2 + (-30\,y^2 + 6\,y^5)\,x + 12\,y^4$$

o según las potencias de y:

```
> collect(poli, y);
```

$$6\,x\,y^5 + 12\,y^4 + (-15\,x^2 + 14\,x^3)\,y^3 + (9\,x^3 - 30\,x)\,y^2 + (-35\,x^4 + 18\,x^2)\,y + 21\,x^5$$

Otros ejemplos de manipulación de polinomios de dos variables son los siguientes:

```
> coeff(poli, x^3); coeff(poli, x, 3);
```

$$14\,y^3 + 9\,y^2$$

$$14\,y^3 + 9\,y^2$$

```
> coeffs(poli, x, 'powers'); powers;
```

$$12\,y^4,\ 21,\ -35\,y,\ -30\,y^2 + 6\,y^5,\ 18\,y - 15\,y^3,\ 14\,y^3 + 9\,y^2$$

$$1,\ x^5,\ x^4,\ x,\ x^2,\ x^3$$

3.2.2. Funciones racionales

Las ***funciones racionales*** son funciones que se pueden expresar como cociente de dos polinomios, tales que el denominador es distinto de cero. A continuación se van a definir dos polinomios **f** y **g**, y su cociente:

```
> f := x^2 + 3*x + 2; g := x^2 + 5*x + 6; f/g;
```

$$f := x^2 + 3x + 2$$

$$g := x^2 + 5x + 6$$

$$\frac{x^2 + 3x + 2}{x^2 + 5x + 6}$$

Para acceder al numerador y al denominador de una función racional existen los comandos numer y denom:

```
> numer(%), denom(%);
```

$$x^2 + 3x + 2,\ x^2 + 5x + 6$$

Por defecto, Maple no simplifica las funciones racionales. Las simplificaciones sólo se llevan a cabo cuando Maple reconoce factores comunes. Considérese el siguiente ejemplo:

```
> ff := (x-1)*f; gg := (x-1)^2*g;
```

$$ff := (x-1)\left(x^2 + 3x + 2\right)$$

$$gg := (x-1)^2\left(x^2 + 5x + 6\right)$$

```
> ff/gg;
```

$$\frac{x^2 + 3x + 2}{(x-1)\left(x^2 + 5x + 6\right)}$$

Para simplificar al máximo y explícitamente, se utiliza la función normal(se explica en el siguiente tema):

```
> f/g, normal(f/g);
```

$$\frac{x^2 + 3x + 2}{x^2 + 5x + 6},\ \frac{x+1}{x+3}$$

```
> ff/gg, normal(ff/gg);
```

$$\frac{x^2 + 3x + 2}{(x-1)\left(x^2 + 5x + 6\right)},\ \frac{x+1}{(x+3)(x-1)}$$

Existen varios motivos para que las expresiones racionales no se simplifiquen automáticamente. En primer lugar, porque los resultados no siempre son más simples;

además, se gastaría mucho tiempo en simplificar siempre y, finalmente, al usuario le puede interesar otra cosa, por ejemplo hacer una descomposición en fracciones simples.

Puede haber también expresiones racionales en varias variables, por ejemplo :

```
> f := 161*y^3 + 333*x*y^2 + 184*y^2 + 162*x^2*y + 144*x*y+ 77*y +
99*x + 88;
```

$$f := 161\,y^3 + 333\,x\,y^2 + 184\,y^2 + 162\,x^2\,y + 144\,x\,y + 77\,y + 99\,x + 88$$

```
> g := 49*y^2 + 28*x^2*y + 63*x*y +147*y + 36*x^3 + 32*x^2+ 117*x +
104;
```

$$g := 49\,y^2 + 28\,x^2\,y + 63\,x\,y + 147\,y + 36\,x^3 + 32\,x^2 + 117\,x + 104$$

```
> racexp := f/g;
```

$$racexp := \frac{161\,y^3 + 333\,x\,y^2 + 184\,y^2 + 162\,x^2\,y + 144\,x\,y + 77\,y + 99\,x + 88}{49\,y^2 + 28\,x^2\,y + 63\,x\,y + 147\,y + 36\,x^3 + 32\,x^2 + 117\,x + 104}$$

```
> normal(racexp);
```

$$\frac{18\,x\,y + 23\,y^2 + 11}{4\,x^2 + 7\,y + 13}$$

Una operación muy útil en el manejo de las funciones racionales es la descomposición en ***fracciones parciales***. Esta transformación puede llegar a ser muy interesante a la hora de realizar ciertas operaciones matemáticas como puede ser la integración indefinida. Para realizar esta transformación utilizaremos el comando ***convert*** especificando la opcion *'parfrac'*. Veámoslo en un ejemplo:

```
> fraccion:=(x^3+4*x^2+x+3)/(x^4+5*x^3+3*x^2-5*x-4);
```

$$fraccion := \frac{x^3 + 4\,x^2 + x + 3}{x^4 + 5\,x^3 + 3\,x^2 - 5\,x - 4}$$

```
> convert (fraccion, 'parfrac',x);
```

$$\frac{19}{36\,(x+1)} - \frac{5}{6\,(x+1)^2} + \frac{1}{45\,(x+4)} + \frac{9}{20\,(x-1)}$$

3.3. ECUACIONES Y SISTEMAS DE ECUACIONES. INECUACIONES

3.3.1. Resolución simbólica

Maple tiene la posibilidad de resolver ecuaciones e inecuaciones con una sola incógnita, con varias incógnitas e incluso, la de resolver simbólicamente sistemas de ecuaciones e inecuaciones. Para estas resoluciones se utiliza la función ***solve***.

La solución de una ecuación simple es una expresión o una *secuencia de expresiones*, y la solución a un sistema de ecuaciones es un sistema de expresión con las incógnitas despejadas a no ser que introduzcamos los datos en forma de *sets* caso en el que el programa nos devolverá también las soluciones en *sets o secuencias de sets*.

La función ***solve*** nos devuelve generalmente un conjunto de soluciones.

```
> ec:=x^2+2*x+1;
```

$$ec := x^2 + 2x + 1$$

```
> solve(ec,x); # nos devuelve las soluciones de la ecuación
```

$$-1, -1$$

En el caso de múltiples soluciones coloca a éstas en una lista o conjunto antes de manipularlas. Cuando ***solve*** es incapaz de encontrar soluciones devuelve la secuencia **NULL**. Esto quiere decir que no hay solución o que la función ***solve*** no ha sido capaz de encontrarla.

Cuando la expresión no se iguala a cero,es decir, cuando lo que introducimos es una expresión y no una ecuación, Maple lo supone. Si no se especifican las incógnitas (2º argumento), Maple lo resolverá para todas ellas.

En general la solución explícita en términos de raices de una ecuación polinómica de grado mayor que 4 no existe. En estos casos la solución implícita viene dada en notación de RootOf. No se da la solución explícita porque es demasiado complicada. Por un lado saber que poniendo la variable global **_EnvExplicit** como **true** solve les devolverá las soluciones explícitas para los polinomios de 4º grado en todos los casos. Por otro lado saber que poniendo **_EnvExplicit** como **false** todas las soluciones no racionales serán presentadas en la noración RootOf.

```
> _EnvExplicit:=true;
```

$$_EnvExplicit := true$$

```
> solve(x^4+x^3+9*x-2,x); # nos devolvera el resultado explícito.
```

Comprueben ustedes el resultado explícito de esta última ecuación.

El número de soluciones encontradas pueden ser controlados cambiando el valor de la variable global _MaxSols. Si se le asigna a _MaxSols un entero, solo devolverá ese número de soluciones. Por defecto _MaxSols es igual a 100. La variable _EnvAllSolutions, si se toma como true, obligará a devolver todo el conjunto de soluciones.

```
> _EnvAllSolutions:=true;
```

$$_EnvAllSolutions := true$$

Una sencilla ecuación cuadrática con coeficientes constantes en una variable se resolverá directamente sustituyéndolo dentro de la fórmula cuadrática. Sin embargo, si _EnvTryHard está como true, Maple intentará expresar las soluciones en la base de la raíz común. Puede proporcionar una respuesta más elaborada, pero puede llevar mucho tiempo.Veamos algunos ejemplos:

```
> solve({x^2=4},{x});
```

$$\{x = 2\}, \{x = -2\}$$

```
> solve({a*x^2+b*x+c},{x}); #toma la expresión igualada a 0
```

$$\left\{x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\right\}, \left\{x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\right\}$$

En estos ejemplos hemos resuelto una ecuación con una incógnita, por lo tanto cada *set* contiene un único elemento.

Se pueden asignar las soluciones a variables mediante el comando de asignación ***assign*** ya que el programa no efectúa esto por defecto:

```
> res:=solve({cos(x)+y=9},{x});x;
```

$$res := \{x = \pi - \arccos(y - 9)\}$$

$$x$$

```
> res:=solve({cos(x)+y=9},{x});
```

$$res := \{x = \pi - \arccos(y - 9)\}$$

```
> assign(res);x;
```

$$\pi - \arccos(y - 9)$$

Aunque el empleo de las llaves (denotando un set) no es obligatorio en el comando, su uso, como hemos comentado, fuerza al programa a devolver las soluciones en forma de *sets*, que habitualmente es la forma más útil. Por ejemplo, suele ser conveniente comprobar soluciones sustituyéndolas en las ecuaciones originales. Veamos un ejemplo con un sistema de ecuaciones.

```
> ecs:={x+2*y=3, y+1/x=1};
```

$$ecs := \{x + 2\,y = 3, y + \frac{1}{x} = 1\}$$

```
> sols:=solve(ecs,{x,y});
```

$$sols := \{x = -1, y = 2\}, \{x = 2, y = \frac{1}{2}\}$$

El comando nos ha producido dos soluciones:

```
> sols[1];sols[2];
```

$$\{x = -1, y = 2\}$$

$$\{x = 2, y = \frac{1}{2}\}$$

Para comprobar las soluciones basta con sustituirlas en las ecuaciones originales. La forma más apropiada es utilizando el comando ***eval*** que con esta sintaxis sustituye lo que tiene como segundo argumento en el primero:

```
> eval(ecs, sols[1]);
```

$$\{3 = 3, 1 = 1\}$$

```
> eval(ecs, sols[2]);
```

$$\{3 = 3, 1 = 1\}$$

Este mismo comando eval también puede ser utilizado con esta misma sintaxis para recuperar el valor de x, por ejemplo, de la primera solución:

```
> val_x:=eval(x,sols[1]);
```

$$val\_x := -1$$

También se podría haber utilizado el comando subs para la comprobación:

```
> subs(sols[1],ecs);
```

$$\{3 = 3, 1 = 1\}$$

```
> map(subs,[sols],ecs); #todas las soluciones
```

$$[\{3 = 3, 1 = 1\}, \{3 = 3, 1 = 1\}]$$

Maple es también capaz de resolver ecuaciones en valor absoluto:

```
> solve( abs( (z+abs(z+2))^2-1 )^2 = 9, {z});
```

$$\{z = 0\}, \{z \le -2\}$$

En el caso de trabajar con inecuaciones, el procedimiento es análogo:

```
> solve({x^2+x>5},{x});
```

$$\{x < -\frac{1}{2} - \frac{\sqrt{21}}{2}\}, \{-\frac{1}{2} + \frac{\sqrt{21}}{2} < x\}$$

```
> eqns:={(x-1)*(x-2)*(x-3)<0};
```

$$eqns := \{(x-1)(x-2)(x-3) < 0\}$$

```
> sols:=solve(eqns,{x});
```

$$sols := \{x < 1\}, \{2 < x, x < 3\}$$

3.3.2. Resolución numérica

Hay ocasiones en las que puede interesar (o no haber más remedio) resolver las ecuaciones o los sistemas de ecuaciones numéricamente, desechando la posibilidad de hacerlo simbólicamente. El comando ***fsolve*** es el equivalente numérico a ***solve***. Este comando encuentra las raíces de las ecuaciones utilizando una variación del método de Newton, produciendo soluciones aproximadas (de coma flotante).

```
> fsolve({cos(x)-x=0},{x});
```

$$\{x = 0.7390851332\}$$

La función ***fsolve*** resuelve únicamente ecuaciones. Este comando intenta encontrar una sola raíz real en una ecuación no lineal de tipo general pero, si estamos ante una ecuación polinómica, es capaz de hallar todas las raíces posibles.

```
> poly:=3*x^4-16*x^3-3*x^2+13*x+16;
```

$$poly := 3x^4 - 16x^3 - 3x^2 + 13x + 16$$

```
> fsolve({poly},{x}); #solo nos muestra las reales
```

$$\{x = 1.324717957\}, \{x = 5.333333333\}$$

Si queremos también encontrar las soluciones complejas, basta con pasar al comando como argumento adicional complex.

```
> fsolve({poly},{x},complex);
```

$$\{x = -0.6623589786 - 0.5622795121\,I\}, \{x = -0.6623589786 + 0.5622795121\,I\},$$
$$\{x = 1.324717957\}, \{x = 5.333333333\}$$

Si queremos limitar el número de soluciones, trabajando con polinomios, basta con utilizar la opción *maxsols*.

```
> fsolve({poly},{x},maxsols=1);
```

$$\{x = 1.324717957\}$$

Hay veces que nos puede ocurrir que el comando fsolve nos proporciona soluciones que no deseamos y, salvo en el caso de los polinomios, el programa no nos genera más soluciones. Para solucionar este inconveniente hay que emplear la opción avoid del comando. Veamos un ejemplo:

```
> fsolve({sin(x)=0},{x});
```

$$\{x = 0.\}$$

```
> fsolve({sin(x)=0},{x},avoid={x=0});
```

$$\{x = -3.141592654\}$$

Asimismo, se puede especificar un intervalo en el que buscar las soluciones:

```
> fsolve({poly},{x},-Pi..Pi);
```

$$\{x = 1.324717957\}$$

Considérese finalmente un ejemplo de sistema de ecuaciones no lineales:

```
> f := sin(x+y)-exp(x)*y = 0;
```

$$f := \sin(x+y) - e^x y = 0$$

```
> g := x^2-y = 2;
```

$$g := x^2 - y = 2$$

```
> fsolve({f,g},{x,y},{x=-1..1, y=-2..0});
```

$$\{y = -1.552838698, x = -.6687012050\}$$

3.4. PROBLEMAS DE CÁLCULO DIFERENCIAL E INTEGRAL

3.4.1. Cálculo de límites

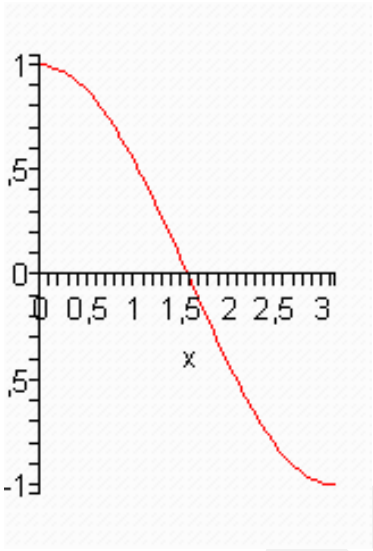
Maple tiene la posibilidad de hallar *límites* de expresiones (o de funciones). El comando ***limit*** tiene 3 argumentos. El primer argumento es una *expresión*, el segundo es una variable igualada a un *punto límite*, mientras que el tercer parámetro –que es opcional– es la *dirección* en la que se calcula el límite —es decir, aproximándose por la derecha o por la izquierda al punto límite—. Si no se indica la dirección, Maple calcula el límite por ambos lados.

Si el límite en cuestión no existe, Maple devuelve "*undefined*" como respuesta; si existe pero no lo puede calcular devuelve una forma no evaluada de la llamada al límite. En algunos casos, a pesar de no existir el límite bidireccional en un punto dado, puede existir alguno de los límites direccionales en ese punto. Utilizando el tercer argumento en la llamada a ***limit***, se pueden calcular estos límites por la derecha y por la izquierda. Un ejemplo típico es la función tangente:

```
> limit(cos(x),x=Pi/2); # nos devuelve el límite cuanto x tiende a
Pi/2
```

$$0$$

```
> plot(cos(x),x=0..Pi); # en el dibujo vemos cómo en Pi/2 la
función cos(x) es cero
```




```
> limit(tan(x),x=Pi/2);
```

$$\textit{undefined}$$

```
> limit(tan(x),x=Pi/2,left); limit(tan(x),x=Pi/2,right);
```

$$\infty$$

$$-\infty$$

```
> limit((x^2+5*x+28)/(x^4),x=0);
```

$$\infty$$

```
> limit((x^2+5*x+28)/(x^4),x=infinity);
```

$$0$$

El tercer argumento también puede ser "complex" o "real", para indicar en cual de los dos planos se quiere calcular el límite.

Otra forma de introducir límites es utilizando la notación Standard Math, la que aparece en las paletas. Para ello clique sobre Expression(se encuentra en la parte izquierda de la pantalla, y si no está vaya a View/Palette . Clique sobre el icono de límite y obtendrá:

```
> limit(%f,%x=%a);
```

Ahora sólo tiene que escribir como primer argumento la expresión de la cual quiere obtener el límite y como segundo argumento hacia que punto tiende x. De este modo construirá su propio límite.

Muchas veces puede ser útil utilizar el comando Limit (con mayúscula) al presentar una hoja de trabajo ya que este comando no evalúa el límite, sino que sólo lo deja indicado. Veamos un ejemplo:

```
> Limit(cos(x),x=Pi/2)=limit(cos(x),x=Pi/2);
```

$$\lim_{x \to \frac{1}{2}\pi} \cos(x) = 0$$

EJERCICIOS:

E-1. Calcule el límite de (cos(2x)sin(x) – xln(x)) cuando x tiende a Pi/4. Dibuje después la función para comprobar el resultado.

E-2. Haga lo mismo con la función exp(x^2) x +(x^2 +5)tan(x).

3.4.2. Cálculo de derivadas

El comando ***diff*** ofrece la posibilidad de ***derivar*** una expresión respecto a una variable dada. El primer argumento de esta función es la expresión que se quiere derivar y el segundo es la variable respecto a la cual se calcula la derivada. Debe darse al menos una variable de derivación y los parámetros siguientes se entienden como parámetros de derivación de más alto nivel. Si la expresión que se va a derivar contiene más de una variable, se pueden calcular derivadas parciales indicando simplemente las correspondientes variables de derivación.

```
> diff(x^3,x); #derivando expresiones
```

$$3x^2$$

```
> f:=x->exp(-2*x);
```

$$f := x \to \mathbf{e}^{(-2x)}$$

```
> diff(f(x),x); #derivando funciones (no olvidar incluir los
argumentos de la función)
```

$$-2\,\mathbf{e}^{(-2x)}$$

```
> diff(f(x),x,x); #derivamos respecto de x dos veces
```

$$4\,\mathbf{e}^{(-2x)}$$

```
> g:=(x,y)->x^2*y+y^2*x^3; #función de dos variables
```

$$g := (x, y) \to x^2 y + y^2 x^3$$

```
> diff(g(x,y),x); #derivamos respecto de x una vez
```

$$2xy + 3y^2x^2$$

```
> diff(g(x,y),x,x,y); #derivamos respecto de x dos veces y una
respecto de y
```

$$2 + 12xy$$

Puede resultar muy interesante al operar con derivadas el uso del carácter de repetición $ cuando tengamos que derivar varias veces respecto de la misma variable. Veamos uno ejemplo:

```
> diff(1/(x*y),x$2,y$3); #dos veces respecto de x y tres respecto
de y
```

$$-\frac{12}{x^3 y^4}$$

El operador diff devuelve siempre una expresión, aunque hayamos introducido una función como argumento. Sin embargo, hay casos en los que queremos convertir ese argumento en función por conveniencia del problema. Es entonces cuando conviene recordar la utilidad del comando unapply que nos permite convertir una expresión en una función. Lo veremos en un ejemplo:

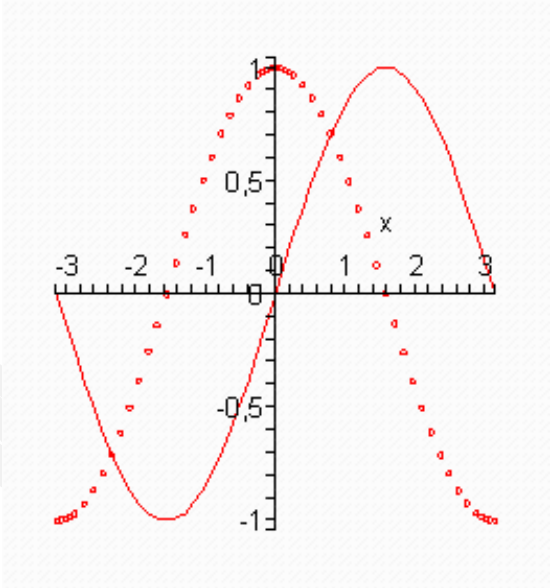
```
> diff(sin(x),x);
```

$$\cos(x)$$

```
> f:=unapply(%,x);  # convierte la expresión anterior en una
función
```

$$f := x \rightarrow \cos(x)$$

```
> plot1:=plot(f(x),x=-Pi..Pi,style=point):
> plot2:=plot(sin(x),x=-Pi..Pi,style=line):
> with(plots):
> display({plot1,plot2});
```



En este dibujo podemos observar la función sin(x) junto a su derivada.

Aunque **diff** es el comando más universal para la derivación, conviene exponer también el funcionamiento de los comandos **Diff** y **D** y sus diferencias con **diff.**

Diff (al igual que sucedía entre limit y Limit) se utiliza cuando no se quiere evaluar la expresión sino que se quiere dejar el resultado en forma de notación, haciendo que la presentación sea más elegante. Si queremos conocer el valor de una expresión donde figure este comando, tendremos que utilizar **value**.

```
> Diff(x^2+5*x+9,x)=diff(x^2+5*x+9,x);
```

$$\frac{d}{dx}(x^2 + 5x + 9) = 2x + 5$$

```
> Diff(ln(x^2+5),x);
```

$$\frac{d}{dx}\ln(x^2+5)$$

```
> value(%);
```

$$\frac{2x}{x^2+5}$$

Por otro lado, el operador D se aplica solamente sobre funciones. En funciones de una sola variable, no necesitamos especificar, por lo tanto, la variable respecto a la que queremos derivar. En el caso de trabajar con funciones de varias variables, hay que indicar al operador, mediante corchetes tras la D, la posición que ocupa la variable respecto a la que queremos derivar dentro de la función. Al utilizar el operador D para derivar una función obtenemos también una función.

```
> f:=x->ln(x)+sin(x); #una variable
```

$$f := x \to \ln(x) + \sin(x)$$

```
> f_prima:=D(f); #devuelve función
```

$$f\_prima := x \to \frac{1}{x} + \cos(x)$$

```
> g:=(x,y,z)->exp(x*y)+sin(x)*cos(z)+x*y*z; #varias variables
```

$$g := (x, y, z) \to \mathbf{e}^{(x\,y)} + \sin(x)\cos(z) + x\,y\,z$$

```
> der:=D[2](g); #respecto a y, variable que ocupa la segunda
posición
```

$$der := (x, y, z) \to x\,\mathbf{e}^{(x\,y)} + x\,z$$

Puede utilizar el botón “Expression” y clicar en el icono de derivada. De esta forma tendrá la secuencia de derivada de forma directa.

```
> diff(%f,%x);
```

EJERCICIOS:

E-1. Calcule la derivada segunda de la siguiente expresión:

```
[(x^3+5x^2+4x)(cos(2x))]/[sin(x)cox(x)]
```

E-2. Calcule la siguiente derivada:

```
exp(x^2) sin(y) + (ln(y)^2)(xy^4 +xyz)tan(z)
```

- **Pruebe a derivar respecto a las diferentes variables, derivadas segundas, etc**.

E-3. Calcule la derivada primera de la siguiente función, y convierta este resultado en una nueva función.

```
f(x)= ln(x)^3 + x^2+5x+6+exp(-x)
```

3.4.3. Cálculo de integrales

Maple realiza la ***integración definida*** y la ***indefinida*** con el comando ***int***. En el caso de la integración indefinida esta función necesita dos argumentos: una expresión y la variable de integración. Si Maple encuentra respuesta, ésta es devuelta sin la constante de integración, con objeto de facilitar su uso en posteriores cálculos. Análogamente a como sucedía en el caso de los límites, si Maple no puede integrar devuelve una llamada sin evaluar.

Estos son algunos ejemplos de integración indefinida:

```
> int(cos(x),x);
```

$$\sin(x)$$

```
> int(exp(5*x),x);
```

$$\frac{1}{5}\,\mathbf{e}^{(5\,x)}$$

```
> int((x^3+4*x^2+6*x+9),x);
```

$$\frac{1}{4}\,x^4+\frac{4}{3}\,x^3+3\,x^2+9\,x$$

En el caso de que se desee realizar una integración definida es suficiente con definir un intervalo de integración como segundo argumento del comando:

```
> int(6*x/(x^2+5),x=0..5);
```

$$3\ln(2)+3\ln(3)$$

```
> int(1/(1+x^2), x=0..infinity);
```

$$\frac{\pi}{2}$$

En el caso de integrales definidas se puede añadir una opción "continuous" para forzar a Maple a ignorar las posibles discontinuidades que se presenten en el intervalo,o podemos poner la opción **_EnvContinuous** como **true**.

```
> int(1/(x+a)^2,x=0..2,'continuous');
```

$$\frac{2}{a\,(a+2)}$$

```
> _EnvContinuous:=true;
```

$$\mathit{\_EnvContinuous} := \mathit{true}$$

La opción **'CauchyPrincipalValue'** instruye a la función ***int*** a considerar el límite por la derecha y por la izquierda de cada discontinuidad interior como un solo límite.

```
> int(1/x^4,x=-3..3,'CauchyPrincipalValue');
```

$$\frac{-2}{81}$$

A diferencia del comando diff, ahora no se pueden añadir variables extras al final de la instrucción para indicar integración múltiple. Una manera de intentarlo, aunque el éxito no esté garantizado, es encerrar unas integraciones dentro de otras:

```
> int(int((sin(y)*x),x),y); # primero integra respecto de x y luego
respecto de y
```

$$-\frac{1}{2}\cos(y)\,x^2$$

```
> int(int(int(cos(y)*x*(z^3),x=0..2),y=-Pi/4..Pi/4),z=0..6);
#integral definida respecto a las tres variables con sus
respectivos límites de integración
```

$$648\sqrt{2}$$

Cuando se aplica la función *int* a una serie, la función interna int/series es llamada para realizar la integral de un modo eficaz.

```
> int(series(sin(x),x=0,6),x);
```

$$\frac{1}{2}x^2 - \frac{1}{24}x^4 + \frac{1}{720}x^6 + O(x^7)$$

Al igual que en los casos anteriores, está a disposición del usuario el comando ***Int***, interesante a la hora de imprimir resultados, ya que devuelve los signos de integración (no evalúa la expresión):

```
> Int(x/(x^3+x^2 +1),x=2..infinity)=int(1/(x^2+x),x=2..infinity);
```

$$\int_2^{\infty} \frac{x}{x^3+x^2+1}\,dx = \ln(3) - \ln(2)$$

Puede también introducir integrales, tanto definidas como indefinidas, utilizando la notación Standard Math. Puede hacerlo directamente clicando en el botón “Expression” y eligiendo el icono de integral definida o indefinida. Le saldrán las siguientes secuencias:

```
int(%f,%x); #integral indefinida
int(%f,%x=%a..%b);# integral definida
```

EJERCICIOS:

E-1. Calcule las integrales de las siguientes funciones:

- **cos(x)sin(2x)x^2 , teniendo como límites de integración a=-Pi/2 y b=Pi/3.**
- **tan(x)exp(y)(z^2) respecto de las 3 variables.**
- **ln(x^4+2*x^2), teniendo como límites de integración a=-1 y b=2**

3.4.4. Desarrollos en serie

Maple dispone del comando ***taylor*** que nos calcula el desarrollo en serie de Taylor de una función o expresión en un punto determinado. El comando nos permite también determinar la precisión (el orden de error) del desarrollo. La sentencia es la siguiente:

taylor(expr, var=punto, n)

donde *expr* es la expresión de la que queremos conocer el desarrollo, *var=punto*, el valor de la variable en torno al cual se realiza el desarrollo, y *n* el grado hasta el cual se quieren calcular los términos. La mejor forma de ver la utilización de este comando es mediante un ejemplo:

```
> exp1:=exp(x)*sin(x);
```

$$exp1 := e^{x} \sin(x)$$

```
> taylor(exp1,x,8);
```

$$x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7 + O(x^8)$$

El resultado obtenido es un desarrollo en serie que puede convertirse en polinomio (es decir, truncar la serie) mediante la función ***convert*** (también se puede clicar con el botón derecho en la salida anterior y elegir ***Truncate Series to Polinomial***). Tras esto, convertiremos la expresión resultante en una función mediante el comando ***unapply***:

```
> convert(%,polynom);
```

$$x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7$$

```
> f1:=unapply(%,x);
```

$$f1 := x \rightarrow x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7$$

Cuanto mayor sea el número de términos mejor será su aproximación en serie de Taylor. Veremos en un ejemplo este hecho, observando la diferencia entre la función original y dos aproximaciones por serie de Taylor:

```
> e1:=cos(x);
```

$$e1 := \cos(x)$$

```
> taylor_1:=taylor(e1,x=0,5); taylor_2:=taylor(e1,x=0,10);
```

$$taylor\_1 := 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 + O(x^5)$$

$$taylor\_2 := 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 - \frac{1}{720}x^6 + \frac{1}{40320}x^8 + O(x^{10})$$

```
> pol_1:=convert(taylor_1,polynom);
```

$$pol\_1 := 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4$$

```
> pol_2:=convert(taylor_2,polynom);
```

$$pol\_2 := 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 - \frac{1}{720}x^6 + \frac{1}{40320}x^8$$

```
> f1:=unapply(pol_1,x);
```

$$f1 := x \rightarrow 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4$$

```
> f2:=unapply(pol_2,x);
```

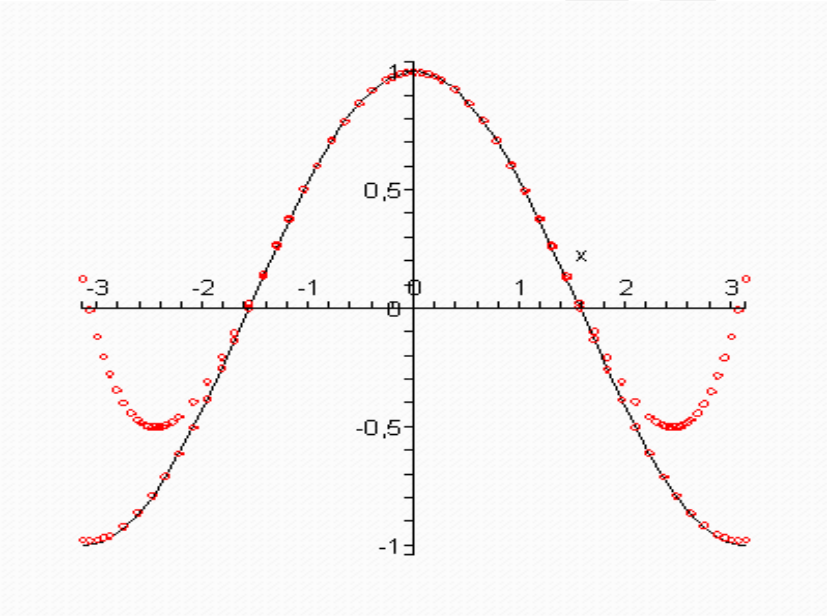
$$f2 := x \rightarrow 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 - \frac{1}{720}x^6 + \frac{1}{40320}x^8$$

```
> plot1:=plot(e1,x=-Pi..Pi,style=line,color=black):

>  plot2:=plot(f1(x),x=-Pi..Pi,style=point):

>  plot3:=plot(f2(x),x=-Pi..Pi,style=point):

>  with(plots):
> display({plot1,plot2,plot3});
```



En la gráfica se observa cómo la aproximación del polinomio que hemos obtenido de convertir la serie de taylor de 10 términos se aproxima más que la de 5 términos.

Maple también dispone del comando **mtaylor** que nos calcula el desarrollo en serie de Taylor para varias variables. Lo vemos con un ejemplo:

```
> tay_varias1:=mtaylor(cos(x+y),[x,y],3);
```

$$tay\_varias1 := 1 - \frac{1}{2}x^2 - yx - \frac{1}{2}y^2$$

```
> tay_varias2:=mtaylor(cos(x+y),[x,y],5);
```

$$tay\_varias2 := 1 - \frac{1}{2}x^2 - yx - \frac{1}{2}y^2 + \frac{1}{24}x^4 + \frac{1}{6}yx^3 + \frac{1}{4}y^2x^2 + \frac{1}{6}y^3x + \frac{1}{24}y^4$$

Además de los desarrollos en serie de Taylor, Maple tiene una función llamada **series** que crea la serie de una expresión respecto a la variable x alrededor del punto **a** hasta el orden **n**.El tercer argumento es opcional Esta sería la secuencia:

```
series(expresión,x=a,n);
```

Ejemplos:

```
> series(sin(x)*cos(x),x=0,5);
```

$$x - \frac{2}{3}x^3 + O(x^5)$$

```
> series(exp(x),x=5,6);
```

$$\mathbf{e}^5 + \mathbf{e}^5 (x-5) + \frac{1}{2}\mathbf{e}^5 (x-5)^2 + \frac{1}{6}\mathbf{e}^5 (x-5)^3 + \frac{1}{24}\mathbf{e}^5 (x-5)^4 + \frac{1}{120}\mathbf{e}^5 (x-5)^5 + O((x-5)^6)$$

EJERCICIO :

E-1. Halle el desarrollo en serie del exp(x) alrededor de x=5. Después convierta esta serie en polinomio. Finalmente obtenga su integral con límites de integración a=0 y b=5.

3.4.5. Integración de ecuaciones diferenciales ordinarias

Maple puede resolver ecuaciones diferenciales ordinarias con el comando ***dsolve***. La sintaxis del comando es la siguiente:

dsolve({ODE, ICs}, y(x), extra_args)

donde *ODE* es la ecuación diferencial deseada, *ICs* las condiciones iniciales (ya sea problema de valor inicial o condiciones de contorno), *y(x)* es la variable y *extra_args*, opciones que se comentarán más adelante.

Es importante tener bien clara la notación necesaria para escribir las ecuaciones. Recordamos que el operador derivada, a la hora de aplicarse a funciones -este caso-, puede efectuarse mediante el comando ***diff*** (aplicable a expresiones y funciones) o mediante el operador ***D*** (aplicable solo a funciones). Apuntar también que en las condiciones de contorno sólo valdrá el operador ***D***. Veamos en un principio dos ejemplos sencillos. Empecemos por una ecuación diferencial ordinaria de primer orden: y'*(x)=a*y(x)*. La resolveremos primero sin condiciones iniciales y luego con ellas. Introduzcamos la ecuación:

```
> ec1:=D(y)(x)=a*y(x);
```

$$ec1 := \mathrm{D}(y)(x) = a\, y(x)$$

Ahora podemos llamar a ***dsolve*** para obtener la solución:

```
> dsolve(ec1,y(x));
```

$$y(x) = \_C1\ \mathbf{e}^{(a\,x)}$$

Al no haber impuesto condiciones iniciales el programa nos ha devuelto la solución en función de una constante de integración. Estas constantes vendrán siempre dadas de la forma _Ci (con i, entero). Establezcamos ahora unas condiciones iniciales:

```
> init:=y(0)=1;
```

$$init := y(0) = 1$$

```
> dsolve({ec1,init},y(x));
```

$$y(x) = \mathbf{e}^{(a\,x)}$$

Si queremos comprobar que se satisface la condición inicial haremos:

```
> assign(%);  #realiza asignaciones
> y:=unapply(y(x),x);
```

$$y := x \to \mathbf{e}^{(a\,x)}$$

```
> y(0);
```

$$1$$

Ahora realizaremos otro ejemplo f''(x)+f(x)=tan(x) de resultado más complicado al ser una ecuación de orden superior, pero como podremos comprobar, el procedimiento utilizado para su resolución es completamente idéntico:

```
> restart;
> ecd:=diff(f(x),x$2)+f(x)=tan(x); #ecuación diferencial
```

$$eqn := \left(\frac{d^2}{dx^2} f(x) \right) + f(x) = \tan(x)$$

```
> init_cond1:=f(0)=1; #condiciones de contorno
> init_cond2:=D(f)(1)=0;
```

$$init\_cond1 := f(0) = 1$$

$$init\_cond2 := D(f)(1) = 0$$

```
> dsolve({eqn,init_cond1,init_cond2},f(x)); #llamamos a dsolve
```

$$f(x) = \cos(x) + \frac{1}{2}\sin(x)\,x - \frac{1}{2}\cos(x)\,x$$

```
> assign(%);
> sol:=unapply(f(x),x);
```

$$sol := x \to \cos(x) + \frac{1}{2}\sin(x)\,x - \frac{1}{2}\cos(x)\,x$$

```
> sol(0); #comprobamos que cumple las condiciones iniciales
```

$$1$$

```
> sol_prima:=D(sol); sol_prima(1);
```

$$sol\_prima := x \to -\frac{1}{2}\sin(x) + \frac{1}{2}\cos(x)\,x + \frac{1}{2}\sin(x)\,x - \frac{1}{2}\cos(x)$$

$$0$$

Otra forma de comprobar el resultado es mediante el comando ***odetest*** que nos devolverá un *0* si el resultado es correcto. Veámoslo:

```
> sol:=dsolve({eqn,init_cond1,init_cond2},f(x)); #llamamos a dsolve
```

$$sol := f(x) = \cos(x) + \frac{1}{2}\sin(x)\,x - \frac{1}{2}\cos(x)\,x$$

```
> odetest(sol,eqn);
```

$$0$$

Maple también puede resolver sistemas de ecuaciones diferenciales ordinarias:

```
> sys := (D@@2)(y)(x) = z(x), (D@@2)(z)(x) = y(x);
```

$$sys := (D^{(2)})(y)(x) = z(x), (D^{(2)})(z)(x) = y(x)$$

En este ejemplo no se especifican condiciones iniciales.

```
> dsolve( {sys}, {y(x), z(x)} );
```

Se puede convertir un sistema de ecuaciones diferenciales ordinarias, como el anterior, en un sistema de primer orden con el comando convertsys. Este comando se encuentra en una librería de funciones todas relacionadas con ecuaciones diferenciales que se llama DEtools.

Centrémonos ahora en algunas opciones extra que se le pueden pasar al comando dsolve:

- *implicit*: para evitar que ***dsolve*** intente darnos la solución de manera explícita
- *explicit*: para requerir soluciones en forma explícita en todos los casos (contando que la resolución logre aislar la variable independiente)
- *parametric*: sólo para ecuaciones de primer orden, para forzar a emplear el esquema de resolución paramétrica. ***dsolve*** intentará eliminar el parámetro utilizado durante el proceso de resolución. Para poder conservar el parámetro, tendremos que utilizar a su vez la opción *implicit*
- *useInt*: esta opción fuerza el uso de ***Int*** (la integral no evaluada) en vez del operador de integración por defecto. Es útil para ahorrar tiempo de cálculo muchas veces y para ver la forma de la solución antes de que las integrales sean evaluadas. Para evaluarlas, basta con aplicar el comando ***value*** a la solución proporcionada por ***dsolve***.

Veamos unos ejemplos:

```
> eqn:=D(y)(x)=a*y(x);
```

$$eqn := D(y)(x) = a\,y(x)$$

```
> dsolve(eqn,y(x),implicit,parametric); #resultado de forma
paramétrica
```

$$\left[y(\_T) = \frac{\_T}{a},\ x(\_T) = \frac{\ln(\_T) + \_C1\ a}{a} \right]$$

```
> restart;
> eqn:=diff(f(x),x$2)+f(x)=cos(x)+sin(x); #la ecuación diferencial
```

$$eqn := \left(\frac{d^2}{dx^2} f(x)\right) + f(x) = \cos(x) + \sin(x)$$

```
> sol:=dsolve(eqn,f(x),useInt); #sin evaluar las integrales
```

$$sol := f(x) = \sin(x)\ \_C2 + \cos(x)\ \_C1 + \int \cos(x)^2 + \cos(x)\sin(x)\, dx \sin(x) - \int \cos(x)\sin(x) + 1 - \cos(x)^2\, dx \cos(x)$$

```
> value(%); #las evaluamos
```

$$f(x) = \sin(x)\ \_C2 + \cos(x)\ \_C1 + \left(\frac{1}{2}\cos(x)\sin(x) + \frac{x}{2} - \frac{1}{2}\cos(x)^2\right)\sin(x) - \left(-\frac{1}{2}\cos(x)^2 + \frac{x}{2} - \frac{1}{2}\cos(x)\sin(x)\right)\cos(x)$$

```
> simplify(%); #simplificamos
```

$$f(x) = \sin(x)\ \_C2 + \cos(x)\ \_C1 + \frac{1}{2}\cos(x) + \frac{1}{2}\sin(x)\, x - \frac{1}{2}\cos(x)\, x$$

Finalmente se puede nombrar la opción de resolución numérica de ***dsolve***. Sus opciones son muchas y se anima al lector a explorarlas en el help del programa (***?dsolve,numeric)***. Por defecto utiliza para la resolución numérica de problemas de valor inicial el método de Runge-Kutta Fehlberg (rkf45) y para los de contorno, un método de diferencias finitas con la extrapolación de Richardson. La salida por defecto es un proceso. Este proceso acepta como argumento el valor de la variable independiente y devuelve una lista de los valores numéricos de la solución de la forma *variable=valor*, donde aparecen los valores de la variable independiente, de las dependientes y de sus derivadas. Veamos dos ejemplos:

Ejemplo de problema de valor inicial:

```
> deq1  := (t+1)^2*diff(y(t),t,t) + (t+1)*diff(y(t),t)
    + ((t+1)^2-0.25)*y(t) = 0; #la ecuación diferencial
```

$$deq1 := (t+1)^2\left(\frac{d^2}{dt^2} y(t)\right) + (t+1)\left(\frac{d}{dt} y(t)\right) + ((t+1)^2 - 0.25)\, y(t) = 0$$

```
> ic1   := y(0) = 1, D(y)(0) = 1.34252: #condiciones iniciales
> dsol1 := dsolve({deq1,ic1}, numeric); #resolvemos
```

$$dsol1 := \mathbf{proc}(x\_rkf45) \ \ldots\ \mathbf{end\ proc}$$

```
> dsol1(0); #soluciones en algunos puntos
```

$$\left[t = 0.,\ y(t) = 1.,\ \frac{d}{dt} y(t) = 1.3425200000000\right]$$

```
> dsol1(1.5);
```

$$\left[t = 1.5,\ y(t) = 1.20713070848376747,\ \frac{d}{dt} y(t) = -0.789866586936627812\right]$$

Ejemplo de problema de contorno:

```
> deq2:=diff(y(x),x,x)=4*y(x);
```

$$deq2 := \frac{d^2}{dx^2} y(x) = 4\, y(x)$$

```
> init:=y(0)=3.64,y(2)=1.3435;
```

$$init := y(0) = 3.64,\ y(2) = 1.3435$$

```
> sol2:=dsolve({deq2,init},numeric);
```

$$sol2 := \mathbf{proc}(x\_bvp)\ \ldots\ \mathbf{end\ proc}$$

```
> sol2(0);
```

$$\left[x = 0.,\ y(x) = 3.6400000000000013,\ \frac{d}{dx} y(x) = -7.18642469929130012\right]$$

```
> sol2(2);
```

$$\left[x = 2.,\ y(x) = 1.34349999999999969,\ \frac{d}{dx} y(x) = 2.42203818337829935\right]$$

EJERCICIOS:

E-1.Resuelva los siguientes problemas de valor inicial:

- **3ty''-(t+3)y'+ln(t+1)y=0 con y(0)=1 e y'(0)=0**
- **(1-t^2)y''-ty'+4y = 0 con y(0)=1 e y'(0)=0**
- **(1+t)y'' – (1+2t)y' +ty=0 con y(0)=3 e y'(0)=2**

E-2. Resuelva los siguientesproblemas de contorno:

- **y'+6y=0 con y(0)=1**
- **y'' –y=0 con y(0)=1 e y (3)=0**
- **4y''+4y=0 con y(0)=5 e y(5)= 10**

4- OPERACIONES CON EXPRESIONES

Maple dispone de muchas herramientas para modificar o, en general, para manipular expresiones matemáticas. Al intentar simplificar o, simplemente, modificar una expresión, existen dos opciones: la primera es modificar la expresión como un *todo* y la segunda es intentar modificar ciertas partes de la expresión. A las primeras se les podría denominar ***simplificaciones*** y a las segundas ***manipulaciones***. Se comenzará por las primeras.

Los procedimientos de simplificación afectan de manera distinta a las expresiones dependiendo de si las partes constitutivas de la expresión a modificar son *trigonométricas*, *exponenciales*, *logarítmicas*, *potencias*, etc.

Es muy importante tener en cuenta que no todas las simplificaciones que Maple realiza automáticamente son del todo correctas. Considérese el siguiente ejemplo:

```
> sum(a[k]*x^k, k=0..10);
```

$$a_0+a_1 x+a_2 x^2+a_3 x^3+a_4 x^4+a_5 x^5+a_6 x^6+a_7 x^7+a_8 x^8+a_9 x^9+a_{10} x^{10}$$

```
> eval(subs(x=0, %));
```

$$a_0$$

El resultado que da Maple es aparentemente correcto, pero esto es debido a que ha tomado $0^0 = 1$ y esto no es del todo cierto. Teniendo esto en cuenta (que no siempre se cumple que 0*x = 0 o que x-x =0), se verán a continuación algunas formas de simplificar expresiones.

Al final de cada apartado podrán realizar unos ejercicios que les ayudará a entender mejor la utilización de cada una de las funciones que a continuación se explican.

4.1. SIMPLIFICACIÓN DE EXPRESIONES

4.1.1. Función expand

La principal función del comando ***expand*** es la de distribuir una expresión en forma de suma de productos de otras funciones más sencillas. El comando puede trabajar tanto con polinomios, potencias, como con la mayoría de funciones matemáticas. En el primer caso expandirá el polinomio en forma de suma de términos:

```
> poli:=(x+1)*(x+3)*(x+5)*(x+7);
```

$$poli := (x+1)(x+3)(x+5)(x+7)$$

```
> expand(poli);
```

$$x^4+16x^3+86x^2+176x+105$$

En el caso de trabajar con fracciones, Maple expandirá el numerador de la fracción.

```
> fra:=((x+1)*(x+3)*x)/(y*(z+1));
```

$$fra := \frac{(x+1)(x+3)x}{y(z+1)}$$

```
> expand(fra);
```

$$\frac{x^3}{y(z+1)} + \frac{4x^2}{y(z+1)} + \frac{3x}{y(z+1)}$$

Si trabajamos con funciones matemáticas, el programa utilizará reglas de expansión que lleven a expresiones más sencillas (siempre en forma de suma de productos):

```
> cos(2*x): %=expand(%); #función trigonométrica sencilla
```

$$\cos(2x) = 2\cos(x)^2 - 1$$

```
> cos(x*(y+z)): %=expand(%); #función más complicada
```

$$\cos(x(y+z)) = \cos(xy)\cos(xz) - \sin(xy)\sin(xz)$$

A la hora de trabajar con logaritmos, hay que especificar el signo de las variables para que la expansión pueda efectuarse, garantizando su existencia. Esto se consigue mediante el comando ***assume*** que se emplea con la forma *assume(expr)*. Este comando nos permite establecer unas condiciones, y éstas irán dentro del paréntesis.

Ejemplo:

```
> ln(x/y):   (%)=expand(%); #la función expand no hace efecto ya que
debemos poner la condición de que x >0 ó y >0 al tratarse de logarítmos
neperianos.
```

$$\ln\left(\frac{x}{y}\right) = \ln\left(\frac{x}{y}\right)$$

Utilizaremos por ello la función **assume** mencionada anteriormente:

```
> assume(x>0, y>0): ln(x/y): %=expand(%);
```

$$\ln\left(\frac{x\sim}{y\sim}\right) = \ln(x\sim) - \ln(y\sim)$$

Las variables *x* e *y* aparecen marcadas con un ~ al estar *condicionadas*.

A la hora de trabajar con este comando es también posible expandir expresiones de un modo parcial, indicando como segundo argumento la parte de la expresión que no se quiere expandir. Veamos esta característica en un ejemplo:

```
> poli:=(x+1)*(x+3)*(y+z^2);
```

$$poli := (x+1)\,(x+3)\,(y+z^2)$$

```
> expand(poli);
```

$$x^2 y + x^2 z^2 + 4\,x\,y + 4\,x\,z^2 + 3\,y + 3\,z^2$$

```
> expand(poli,(x+1)); #después de la coma indicamos el factor que no
queremos expandir
```

$$(x+1)\,x\,y + (x+1)\,x\,z^2 + 3\,(x+1)\,y + 3\,(x+1)\,z^2$$

En el segundo ejemplo apreciamos la diferencia respecto al primero. Vemos cómo podemos expandir de una forma parcial, indicádole despúes de la coma cual es el factor que no queremos que se expanda.

EJERCICIOS:

E-1. Vea cómo funciona la función *expand* en estas expresiones:

- **(x+1)*(x-1)^2**
- **(x+2)*(x-2)/(((x+2)^4)*(x+5))**
- **ln((x^3)/y) (Recuerde la función *assume*)**
- **sin(5*x) ; cos(5*x)**
- **sin(x)*cos(x)* sin(x+y)*cos(x+y)**
- **exp((x+y)*(x+z)*(y+z))**

E-2.Halle la expresión del factorial de(x+8) y una vez obtenida esta expresión dele diferentes valores a la variable x .

E-3.Expanda la siguiente expresión:

- **sqrt(x^3)*exp(z*(2*x+y))*(x+1)^4/((exp(z*y))*sqrt(x^2))**

-Comprobará la importancia de utilizar la función *assume* .

4.1.2. Función combine

Es el comando que realiza la tarea inversa a la que hace ***expand*** en muchas ocasiones**.** La función ***combine*** combina varias expresiones para conseguir una más compacta o reducida. Para ello, generalmente, las transformaciones utilizadas son las inversas que en ***expand***. Por ejemplo, si tenemos la siguiente identidad conocida:

$$\sin(a+b) = \sin(a)*\cos(b) + \cos(a)*\sin(b)$$

expand la utilizaría de izquierda a derecha, mientras que ***combine*** de derecha a izquierda.

Un sencillo ejemplo de esta función es el siguiente:

```
> combine(exp(x)*exp(y)*exp(z));
```

$$e^{(x+y+z)}$$

Vemos cómo la función ha transformado la expresión a una forma más compacta.

Al utilizar ***combine*** se pude indicar como argumento qué tipo de elementos son los que se desean combinar, para que Maple tenga en cuenta las reglas apropiadas en cada caso. Los posibles tipos de combinación son: ***trig***, ***exp***, ***ln***, ***power***, ***radical*** y ***arctan*** .

Aquí se muestran algunas reglas de combinación a modo de ejemplo:

trig:

```
sin x sin y = 1/2 cos(x-y)-1/2 cos(x+y)
sin x cos y = 1/2 sin(x-y)+1/2 sin(x+y)
cos x cos y = 1/2 cos(x-y)+1/2 cos(x+y)
```

exp, ln:

```
exp x exp y = exp (x+y);
exp (x + ln y) = y^n exp(x), para n ∈ Z
(exp x)^y = exp (x*y)
a ln x = ln(x^a)
ln x + ln y = ln (x*y)
```

powers:

```
x^y*x^z= x^(y+z)
(x^y)^z = x^(yz)
```

radical:

```
sqrt(2x)*sqrt(6y)= 2*sqrt(3)*sqrt(xy)
```

arctan:

```
arctan(x)+arctan(y) = arctan[(x+y)/(1-xy)]
```

A continuación iremos mostrando ejemplos de cada una de las combinaciones de la función ***combine*** . La forma de la función quedaría de la siguiente manera:

```
combine(expresión,combinación)
```

```
> combine(cos(x)^2,trig);
```

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

```
> combine(sin(x)*cos(x)+sin(x)^2*cos(x)^2,trig);
```

$$\frac{1}{2}\sin(2x) + \frac{1}{8} - \frac{1}{8}\cos(4x)$$

```
> combine(exp(x)*exp(x+y+z)*exp(2*z),exp);
```

$$\mathbf{e}^{(2x + y + 3z)}$$

```
> combine(x^a*x^b*x^c,power);
```

$$x^{(a + b + c)}$$

```
> combine(arctan(6)+arctan(3),arctan);
```

$$-\arctan\left(\frac{9}{17}\right) + \pi$$

En los siguientes dos ejemplos se muestra cómo distinguiendo la combinación, la función ***combine*** actua diferente.

```
> combine(exp(a)*exp(b)+sin(x)^2,exp); #utilizamos la combinación
"exp"
```

$$\mathbf{e}^{(a + b)} + \sin(x)^2$$

```
> combine(exp(a)*exp(b)+sin(x)^2,trig); #utilizamos la combinación
"trig"
```

$$\mathbf{e}^{a}\,\mathbf{e}^{b} + \frac{1}{2} - \frac{1}{2}\cos(2x)$$

Pero también tenemos la opción de utilizar varias combinaciones a la vez, de este modo podremos compactar toda la expresión. La forma sería la siguiente:

```
combine(expresión,[combinación,combinación,...])
```

```
> combine(exp(a)*exp(b)+sin(x)^2,[exp,trig]);
```

$$\mathbf{e}^{(a + b)} + \frac{1}{2} - \frac{1}{2}\cos(2x)$$

Vemos cómo el ejemplo anterior lo hemos compactado utilizando varias combinaciones.

En el caso de compactar expresiones con logaritmos es necesario (como en el caso de ***expand***) especificar la naturaleza de los términos para asegurarnos que el logaritmo exista. En este caso contamos con otra posibilidad: añadir la opción ***symbolic*** como tercer argumento de la función ***combine***. Esta opción asume que todos los términos son reales y positivos. La forma de la función quedaría del siguiente modo:

```
combine(expresión,combinación,symbolic)
```

También podremos utilizar la función ***assume*** mencionada con anterioridad.

```
> expr:=ln(x)-ln(y);
```

$$expr := \ln(x) - \ln(y)$$

```
> expr=combine(expr,ln); #desconoce la naturaleza de x e y
```

$$\ln(x) - \ln(y) = \ln(x) - \ln(y)$$

```
> expr=combine(expr,ln,symbolic); #opción 'symbolic'
```

$$\ln(x) - \ln(y) = \ln\left(\frac{x}{y}\right)$$

```
> assume(x>0,y>0): expr=combine(expr,ln); #con condiciones
```

$$\ln(x\sim) - \ln(y\sim) = \ln\left(\frac{x\sim}{y\sim}\right)$$

En el caso de la combinación ***radical*** también utilizaremos la función ***assume*** ó la opción ***symbolic*** para concretar la naturaleza de los radicandos, ya que dependiendo del signo podría cambiar nuestro resultado.

```
> combine(sqrt(x)*sqrt(y),radical); # no reconoce la naturaleza de los
radicandos y no varía nada
```

$$\sqrt{x}\,\sqrt{y}$$

```
> combine(sqrt(x)*sqrt(y),radical,symbolic); #utilizamos "symbolic" y
de esta manera asumimos x e y real y positivos
```

$$\sqrt{x\,y}$$

```
> assume(x>0,y>0):combine(sqrt(x)*sqrt(y),radical); # asumimos x e y
positivos
```

$$\sqrt{x\sim\,y\sim}$$

```
> combine(sqrt(6)*sqrt(2)*sqrt(3*x),radical,symbolic);
```

$$6\sqrt{x}$$

```
> assume(x<0):combine(sqrt(6)*sqrt(2)*sqrt(3*x),radical);
```

$$6\,I\sqrt{-x\sim}$$

En los dos últinos ejemplos el resultado depende del signo del radicando, por eso es muy importante especificarlo de la manera explicada.

También podemos utilizar la función ***combine*** complementando a la función ***picewise***. Es muy sencillo, observen el siguiente ejemplo.

```
> combine(piecewise(x>0,exp(2)*exp(3),1-cos(x)^2-sin(x)^2));
```

$$\begin{cases} 0 & x \le 0 \\ \mathbf{e}^{5} & 0 < x \end{cases}$$

De esta forma podemos compactar las expresiones de las diferentes funciones y definirlas en un único paso.

EJERCICIO:

Haga uso de la función combine y sus combinaciones en las siguientes expresiones y observe que ocurre.

- **sin(x)^2cos(x) + (1/4)cos(3x)**
- **exp(x+y)exp(x+z)exp(y+z) +sin(x)^3+arctan(7x) – arctan(2x)**
- **2sqrt(x^3) sqrt(4*x)**

4.1.3. Función simplify

Es el comando general de simplificación de Maple. En el caso de las funciones *trigonométricas* tiene unas reglas propias, pero para funciones *exponenciales*, *logarítmicas* y *potencias* produce los mismos resultados que la función ***expand*** en casi todos los casos. Al igual que en combine podemos especificar el tipo de simplificación que deseamos hacer (*trig*, *exp*, *ln*, *radical*…), pero en este caso sólo se aplicará esa regla, mientras que si no se le especifica ningún argumento adicional, Maple intentará utilizar el mayor número posible de reglas de simplificación. Para simplificar funciones racionales es mejor utilizar el comando ***normal*** que se describe posteriormente, ya que al aplicar simplify sólo se simplifica el numerador.

A continuación vemos un ejemplo en el que se muestra la diferencia de utiliza ***expand*** y ***simplify***. Se aprecia lo comentado de que la función simplify en lo referido a las reglas trigonométricas no funciona como expand, pero sí en las reglas para exponenciales. Observen:

```
> simplify(exp(a)*exp(b)+cos(x)^2);
```

$$\mathbf{e}^{(a+b)} + \cos(x)^2$$

```
> combine(exp(a)*exp(b)+cos(x)^2);
```

$$\mathbf{e}^{(a+b)} + \frac{1}{2}\cos(2x) + \frac{1}{2}$$

La función ***simplify*** tiene una opción llamada ***size*** muy interesante. Ésta sirve para simplificar el tamaño de la expresión, realizando simples descomposiciones de potencias de fracciones en los coeficientes, y en ocasiones se aprovecha de los factores lineales cuando estos existen para hacer simplificaciones, es decir, reducir el tamaño de la expresión. En primer lugar se calculan numerador y denominador, luego se simplifican en tamaño numerador y denominador por separado, y finalmente nos devuelve el resultado de la expresión obtenida. La forma sería la siguiente:

simplify(expresión, size)

Un ejemplo:

```
>e:=[sqrt(cos(x))*(sin(x))*exp(x)+5*sin(x)*sqrt(cos(x))+sin(x)*sqrt(co
s(x))]/[sqrt(cos(x))*exp(y+z)+10*sqrt(cos(x))];
```

$$e := \frac{[\sqrt{\cos(x)}\ \sin(x)\ \mathbf{e}^{x} + 6\ \sin(x)\ \sqrt{\cos(x)}\,]}{[\sqrt{\cos(x)}\ \mathbf{e}^{(y+z)} + 10\ \sqrt{\cos(x)}\,]}$$

```
> simplify(e,size);
```

$$\frac{[\sin(x)\ (\mathbf{e}^{x} + 6)\ \sqrt{\cos(x)}\,]}{[(\mathbf{e}^{(y+z)} + 10)\ \sqrt{\cos(x)}\,]}$$

Ha reducido el tamaño del numerador y el denominador por separado, factorizando cada expresión. Es una opción muy útil para reducir grandes expresiones.

Hay veces que Maple no efectúa las simplificaciones que deseamos. Muchas veces aunque conocemos propiedades de las variables, el programa las trata de forma mucho más general. En este caso utilizaremos también ***assume*** para especificar la naturaleza de las mismas. Veamos en algunos ejemplos como influye esto en la simplificación:

```
> expr:=sqrt(x^2*y^2):%=simplify(%);
```

$$\sqrt{x^2\,y^2} = \sqrt{x^2\,y^2}$$

```
> expr:=sqrt(x^2*y^2):%=simplify(%,assume=real);
```

$$\sqrt{x^2\,y^2} = |x\,y|$$

assume aplicado como argumento de ***simplify*** se aplica a todas las variables de la expresión.

```
>  assume(x>0):(-x)^y: % = simplify(%);    # x>0
```

$$(-x\sim)^{y} = x\sim^{y}\,(-1)^{y}$$

```
>  assume(y/2,integer,x>0):(-x)^y: % = simplify(%); #x>0 e y es par ya
que y/2 es un número entero
```

$$(-x\sim)^{y\sim} = x\sim^{y\sim}$$

Maple permite también especificar nuestras propias normas de simplificación. En el caso de querer usarlas, tendremos que pasarlas a la función ***simplify*** como argumento dentro de un *set*. Veamos un ejemplo:

```
> rel:={x*z=1}; expr:=x*y*z+x*y+x*z+y*z; #supongamos que sabemos que
x*z=1
```

$$rel := \{x\,z = 1\}$$

$$expr := x\,y\,z + x\,y + x\,z + y\,z$$

```
> simplify(expr,rel);
```

$$x\,y + y\,z + y + 1$$

EJERCICIO:

Simplifique las siguientes expresiones:

- **exp(a+b+c) exp(a) exp(b-3c)**
- **sqrt(x^3) sqrt(y^5) sin(x) + cos(x)^2sin(x) sqrt(x^5) sqrt(z) + sin(x) 5ln(2x+y) sqrt(exp(x^2)) + sin(x) cos(x)**
- **exp(sqrt(xy))ln(xy)^2 + tan(xy) sqrt(xy) 6sqrt(sin(xy)) sabiendo que xy=1**

4.2. MANIPULACIÓN DE EXPRESIONES

Se verán ahora los comandos que al principio de la sección se denominaban *manipulaciones.*

4.2.1. Función normal

Si una expresión contiene fracciones, puede resultar útil expresarla como una sola fracción y luego simplificar numerador y denominador, cancelando factores comunes hasta llegar a lo que se denomina *forma normal factorizada*, que son polinomios primos (indivisibles) con coeficientes enteros. Recalcar que sólo simplifica expresiones algebraicas.

```
> normal( (x^2-y^2)/(x-y)^3 );
```

$$\frac{x+y}{(x-y)^2}$$

```
> normal((f(x)-1)/(f(x)^2-1));
```

$$\frac{1}{\mathrm{f}(x)+1}$$

Si queremos que ***normal*** expanda en su resultado tanto el numerador como el denominador hay que proporcionarle el segundo argumento *expanded*:

```
> normal( (x^2-y^2)/(x-y)^3, expanded ); #expandido
```

$$\frac{x+y}{x^2-2\,x\,y+y^2}$$

EJERCICIO:

Simplifique las siguientes fracciones:

- **(x^2+4x+4)/(x+2)^2**
- **(x+1)*(x-1)*(x^2+2*x+1)/((x^2-1)*((x+1)^2)*sin(5*x) , y expanda numerador y denominador**

4.2.2. Función factor

El comando ***factor*** permite descomponer un polinomio en factores. Veamos algún ejemplo:

```
> factor(6*x^2+18*x-24);
```

$$6\,(x+4)\,(x-1)$$

Como segundo argumento se le puede asignar el campo en el cual debe realizar la factorización. Si no se le indica ninguno, toma el de los coeficientes del polinomio, como en el caso anterior, siendo éste el de los enteros. Si se le aplica como argumento *real* o *complex*, se realiza la factorización con una aproximación de coma flotante. Hoy en día esta opción sólo está presente para polinomios de una sola variable.

```
> pol:=x^5-x^4-x^3-x^2-2*x+2;
```

$$pol := x^5 - x^4 - x^3 - x^2 - 2\,x + 2$$

```
> factor(pol); #no consigue en los enteros
```

$$x^5 - x^4 - x^3 - x^2 - 2\,x + 2$$

```
> factor(pol,real);
```

$$(x+1.209285532)\,(x-0.6374228562)\,(x-1.924445452)$$
$$(x^2+0.3525827766\,x+1.348242153)$$

El comando ***factor*** no descompone un número entero en factores primos. Para ello hay que utilizar el comando ***ifactor***.

```
> ifactor(21456);
```

$$(2)^4\,(3)^2\,(149)$$

```
> ifactor(902/24);
```

$$\frac{(11)\,(41)}{(2)^2\,(3)}$$

EJERCICIO:

Factorice las siguientes expresiones:

- **x^7+2x^6+3x^5+4x^4+5x^3+6x^2+7x+8**
- **5x^4+50x^3+175x^2+250x+120**

4.2.3. Función convert

Se puede descomponer una fracción algebraica en ***fracciones simples*** con el comando ***convert***. Este comando necesita 3 argumentos: el primero es la fracción a descomponer, el segundo indica el tipo de descomposición y el tercero corresponde a la variable respecto de la cual se realiza la descomposición (opcional si no hay más que una variable). El segundo argumento puede tomar los siguientes valores:

`+`	`*`	D	array	base	binary
confrac	decimal	degrees	diff	double	eqnlist
equality	exp	expln	expsincos	factorial	float
GAMMA	hex	hypergeom	list	listlist	ln
matrix	metric	mod2	multiset	name	octal
parfrac	piecewise	polar	polynom	radians	radical
rational	Ratpoly	RootOf	set	sincos	sqrfree
tan	vector				

A continuación veremos unos ejemplos sencillos para ver como funcionan algunos de los argumentos que se colocan en segunda posición.

```
> convert(cos(x)*(x-y)*z,`+`); # convierte los productos en sumas
```

$$\cos(x) + x - y + z$$

```
> convert(%,`*`);  #convierte las sumas en productos
```

$$-\cos(x)\, x\, y\, z$$

```
> convert(exp(x),confrac,x);  #convierte un numero, una serie, una
función racional o cualquier otra expresión algebraica en una
aproximación a una fracción continua.Es necesario el tercer argumento.
```

$$1 + \cfrac{x}{1 + \cfrac{x}{-2 + \cfrac{x}{-3 + \cfrac{x}{2 + \frac{1}{5}x}}}}$$

```
> convert(Pi,degrees);  #convierte de radianes a grados
```

$$180\ degrees$$

```
> convert(cos(x),exp);  # convierte las expresiones trigonométricas en
sus correspondientes formas exponenciales
```

$$\frac{1}{2}\,\mathbf{e}^{(I\,x)} + \frac{1}{2\,\mathbf{e}^{(I\,x)}}$$

```
> convert(arctan(x),ln); #convierte las funciones trigonométricas
inversa en sus correspondientes formas logarítmicas
```

$$\frac{1}{2} I (\ln(1 - I x) - \ln(1 + I x))$$

```
> convert((x^2+5)/(x^3+2*x),parfrac,x); # convierte la expresión en
sus correspondientes fracciones parciales.
```

$$\frac{5}{2x} - \frac{3x}{2(x^2+2)}$$

```
> convert(2 +3*I,polar);  # convierte a la expresión en su forma polar
```

$$\text{polar}\left(\sqrt{13}, \arctan\left(\frac{3}{2}\right)\right)$$

```
> convert(180*degrees,radians);  #convierte grados en radianes
```

$$\pi$$

```
> convert(0.359862,rational);  # convierte un número de coma flotante
en un número racional aproximado
```

$$\frac{12204}{33913}$$

```
> convert(tan(x),sincos);   # convierte las funciones trigonométricas
en función de sin, cos,sinh o cosh
```

$$\frac{\sin(x)}{\cos(x)}$$

```
> convert(sin(x),tan);  # convierte las funciones trigonométricas en
función de la tangente
```

$$\frac{2\tan\left(\frac{1}{2}x\right)}{1+\tan\left(\frac{1}{2}x\right)^2}$$

Para comprender lo que hace el argumento ***RootOf*** debemos entender primero lo que hace la función ***RootOf***. Esta función nos permite representar todas las raices de una ecuación en una variable. Un sencillo ejemplo:

```
> RootOf(x^2-3=0,x);
```

$$\text{RootOf}(\_Z^2 - 3)$$

En el caso que no pusiesemos la expresión igualada a cero, Maple lo asumiría.

El argumento ***RootOf*** nos permite convertir las raices a la notación ***RootOf***. Para que lo entiendan mejor observen el siguiente ejemplo:

```
> convert(3^(1/2),RootOf);
```

$$\text{RootOf}(\_Z^2 - 3, \mathit{index} = 1)$$

Nos da como resultado una ecuación, la cual tiene como raices el 3^(1/2) .

En relación con lo mencionado de ***RootOf*** tenemos el argumento ***radical***, el cual convierte a RootOf y a las funciones trigonométricas en raices. Lo vemos con unos ejemplos:

```
> convert(RootOf(_z^2-3),radical);
```

$$\sqrt{3}$$

```
> convert(cos(Pi/5),radical);
```

$$\frac{1}{4} + \frac{1}{4}\sqrt{5}$$

Esta función, también se utiliza para transformar desarrollos en serie de funciones polinómicas o de otro tipo, para convertir funciones trigonométricas o hiperbólicas a formas diversas, para cambiar de tipo de objeto, para cambiar de base un número o incluso para cambiar una lista o una matriz a un vector.

```
> desarrollo:= taylor(exp(x),x,5);
```

$$\mathit{desarrollo} := 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + O(x^5)$$

```
> polinomio:=convert(desarrollo,polynom);  # convierte un desarrollo
en serie en un polinomio
```

$$\mathit{polinomio} := 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4$$

```
> lista:=[1,2,3,4];
```

$$\mathit{lista} := [1, 2, 3, 4]$$

```
> conjunto:=convert (lista,set); #cambia el tipo de objeto
```

$$\mathit{conjunto} := \{1, 2, 3, 4\}$$

```
> convert([[1,2,3],[4,5,6]],array);  # convierte una lista en una
matriz
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
> M:=array([[1,2],[3,4]]);
```

$$M := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> convert(M,vector); # convierte una matriz en vector (también
convierte lista a vector)
```

$$[1, 2, 3, 4]$$

```
> num_dec:=32;
```

$$num\_dec := 32$$

```
>num_bin:=convert(num_dec,binary);num_hex:=convert(%,hex);
```

$$num\_bin := 100000$$

$$num\_hex := 186A0$$

En el último ejemplo vemos cómo el argumento ***binary*** convierte un número decimal a su binario correspondiente y cómo el argumento ***hex*** convierte un número decimal(este debe ser entero y positivo) en su hexadecimal .

EJERCICIO:

Defina el desarrollo es serie del sin(x) (y=f(x)) alrededor de x=0. Después ponga la x en función de la y (x=f(y)), y finalmente convierta esa serie en polinomio.

4.2.4. Función sort

El comando ***sort*** se utiliza para ordenar los términos de un polinomio dependiendo del exponente de las variables de mayor a menor. Si no se indica lo contrario, Maple realiza la suma de exponentes antes de la ordenación.

```
> p := y^3+y^2*x^2+x^3+x^5;
```

$$p := y^3 + y^2 x^2 + x^3 + x^5$$

```
> sort(p, [x,y]); # ordena según la suma de exponentes
```

$$x^5 + x^2 y^2 + x^3 + y^3$$

```
> sort(p, y); # ordena según el exponente de y
```

$$y^3 + x^2 y^2 + x^5 + x^3$$

```
> sort(p,[x,y], plex); # ordena alfabéticamente
```

$$x^5 + x^3 + x^2 y^2 + y^3$$

EJERCICIO:

Ordene los términos del siguiente polinomio:

```
x*y^2*z^3+x^4*y^3 +z^3*y^5+x^2*y^4*z^8 +x*z*y
```

- **según el exponente de x, de y** y **de z.**
- **según la suma de exponentes.**
- **alfabeticamente.**

5- FUNCIONES ADICIONALES

5.1. INTRODUCCIÓN

Cuando iniciamos Maple, éste carga sólo el núcleo (kernel), es decir, la base del sistema de Maple. Contiene comandos primitivos y fundamentales como, por ejemplo, el intérprete de lenguaje de Maple, algoritmos para la base del cálculo numérico, rutinas para mostrar resultados y poder realizar operaciones de entrada y salida.

El núcleo es un código en C altamente optimizado (aproximadamente un 10% del total del sistema), éste implementa las rutinas más empleadas para aritmética de enteros y racionales, y para cálculo simple de polinomios.

El 90% restante está escrito en lenguaje Maple y reside en la librería Maple. La librería Maple se divide en dos partes: la principal y los paquetes.

La principal contiene los comandos que más habitualmente se emplean en Maple, además de los que van con el kernel, estos comandos se cargan cuando son requeridos. Los demás comandos se encuentran en los paquetes, cada paquete (package) de Maple contiene una serie de comandos de una determinada área.

Existen 3 maneras de usar un comando de un paquete:

1) Podemos usar el nombre entero del paquete y el comando deseado:

```
paquete[comando](...)
```

Si el paquete tiene un subpaquete se usan los nombres completos del paquete, subpaquete y el comando:

```
paquete[subpaquete][comando](...)
```

2) Podemos activar los nombres cortos de todos los comandos usando el comando with:

```
with(paquete);
```

Al poner punto y coma, Maple mostrará el nombre de todas las funciones adicionales que carga. Si se termina con dos puntos, únicamente indicará, si es el caso, las nuevas definiciones que introduce.

Y si el paquete tiene subpaquetes:

```
with(paquete[subpaquete]);
```

Después de esto es suficiente con teclear el nombre para acceder a un comando.

3) Activar el nombre corto para un solo comando del paquete:

```
with(paquete[subpaquete],cmd);
```

Después de esto es suficiente con teclear el nombre para acceder al comando.

Maple tiene una amplia variedad de paquetes que realizan tareas de distintas disciplinas, a continuación se comentan algunos que pueden resultar de interés. Algunas de ellas son:

- **Codegen**: Funciones que traducen el lenguaje Maple a otros códigos como C, Java...

- **combinat**: Funciones de combinatoria, trabajo con listas...
- **CurveFitting**: Comandos para la aproximación de curvas.
- **finance**: Comandos para computos financieros.
- **LinearAlgebra**: Se estudia en una de las siguientes secciones.
- **Matlab**: Comandos para usar funciones numéricas de Matlab. Sólo accesible si está Matlab instalado en el sistema.
- **networks**: Herramientas para construir, dibujar y analizar redes combinacionales.
- **OrthogonalSeries**: Comandos para manipular series de polinomios ortogonales, o más generalmente, polinomios hipergeométricos.
- **PDEtools**: Para resolver, manipular y visualizar ecuaciones diferenciales en derivadas parciales.
- **plots**: Se estudia en una de las siguientes secciones
- **powseries**: Comandos para crear y manipular series de potencias representadas de la forma general.
- **stats:** Se estudia en una de las siguientes secciones
- **Student**: Se estudia en una de las siguientes secciones
- **VectorCalculus**: Cálculo multivariable y vectorial.

5.2. GRÁFICOS EN 2 Y 3 DIMENSIONES. *(plots)*

La visualización de resultados es una de las capacidades más utilizadas del álgebra computacional. Poder ver de manera gráfica los resultados de expresiones de una o dos variables ayuda mucho a entender los resultados. En cuanto a gráficos, Maple dispone de una gran variedad de comandos. Para representar gráficamente una expresión puede utilizarse el menú contextual o introducir la función correspondiente en la línea de comandos.

El concepto básico de todo comando gráfico de Maple es representar una expresión de una o dos variables en un determinado rango de éstas.

Al ejecutar un comando de dibujo, la gráfica correspondiente queda insertada en la hoja de Maple, como si se tratara de la salida de cualquier otro comando. Basta con clicar sobre la gráfica para que ésta quede seleccionada y aparezcan unos botones adicionales en la barra de herramientas.

Botones adicionales para opciones gráficas 2-D

Estos botones permiten modificar las características del dibujo. Por ejemplo, puede hacerse que la función aparezca representada con trazo continuo o por medio puntos, se pueden dibujar ejes de distinto tipo, y se puede obligar a que la escala sea la misma en ambos ejes. Asimismo, Maple devuelve la posición *(x,y)* de cualquier punto clicando sobre la gráfica.

Además de las opciones hasta ahora mencionadas, en las gráficas de Maple se pueden controlar otros aspectos para ajustar las salidas a las necesidades reales de cada momento. Por ejemplo, estos son los *colores predefinidos* de Maple, aunque el usuario tiene completa libertad para crear los nuevos colores que desee (Para ello, usar el ***help*** tecleando ***?color***)

aquamarine	black	blue	navy	coral
cyan	brown	gold	green	gray
grey	khaki	magenta	maroon	orange
pink	plum	red	sienna	tan
turquoise	violet	wheat	white	yellow

Con la opción *style* se decide si en la gráfica van a aparecer sólo puntos (opción POINT) o si éstos van a ir unidos mediante líneas (opción LINE). En el caso de los polígonos, se puede hacer que el interior de ellos aparezca coloreado con la opción PATCH.

Para añadir títulos a las gráficas existe la opción ***title***. Se puede determinar el tipo de ejes con la opción ***axes***. Los posibles valores de esta última son: FRAME, BOXED, NORMAL y NONE. Se pueden probar estas opciones para establecer las diferencias entre todas ellas.

La opción ***scaling*** puede tener los valores CONSTRAINED y UNCONSTRAINED; esta última opción es la que toma por defecto. Indica si la escala es la misma en ambos ejes (*constrained*) o si es diferente.

Puesto que tenemos muchos tipos de gráficos distintos y cada uno tiene distintas opciones, lo más cómodo para cambiar las propiedades es hacer click con el botón derecho sobre el gráfico una vez que lo tenemos en la hoja de cálculo, se desplegará una lista en la que podemos acceder a las distintas propiedades y opciones.

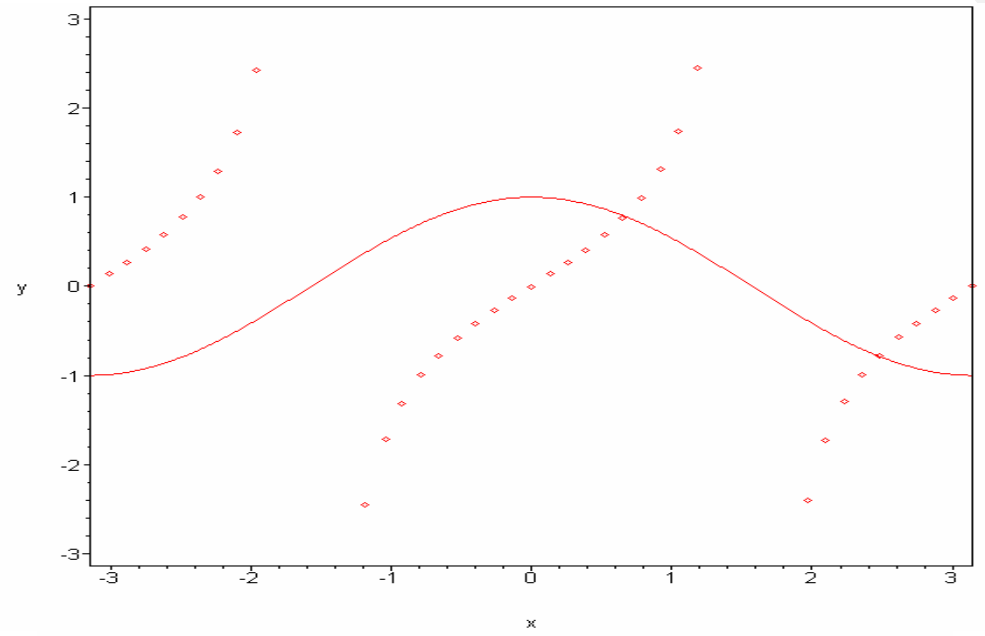
En la librería plots podemos encontrar funciones de mucha utilidad, a continuación se describen algunas. De todas maneras, la lista de todas las funciones existentes es:

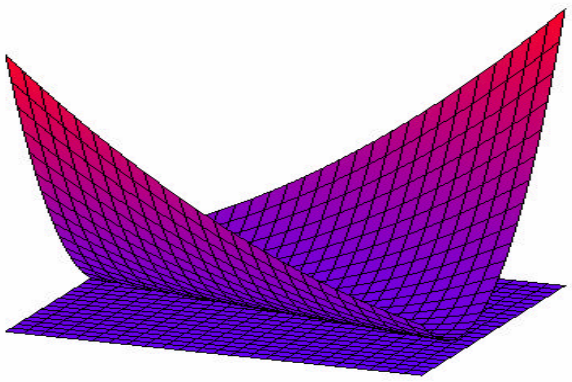
```
> with(plots);
```

[*animate, animate3d, animatecurve, arrow, changecoords, complexplot, complexplot3d, conformal, conformal3d, contourplot, contourplot3d, coordplot, coordplot3d, cylinderplot, densityplot, display, display3d, fieldplot, fieldplot3d, gradplot, gradplot3d, graphplot3d, implicitplot, implicitplot3d, inequal, interactive, interactiveparams, listcontplot, listcontplot3d, listdensityplot, listplot, listplot3d, loglogplot, logplot, matrixplot, multiple, odeplot, pareto, plotcompare, pointplot, pointplot3d, polarplot, polygonplot, polygonplot3d, polyhedra_supported, polyhedraplot, replot, rootlocus, semilogplot, setoptions, setoptions3d, spacecurve, sparsematrixplot, sphereplot, surfdata, textplot, textplot3d, tubeplot*]

- **display**: Está función nos permite visualizar una lista de estructuras plot, la sintaxis es display(**L**) donde L es la lista, conjunto o array de estructuras plot que se quieren visualizar. Ej:

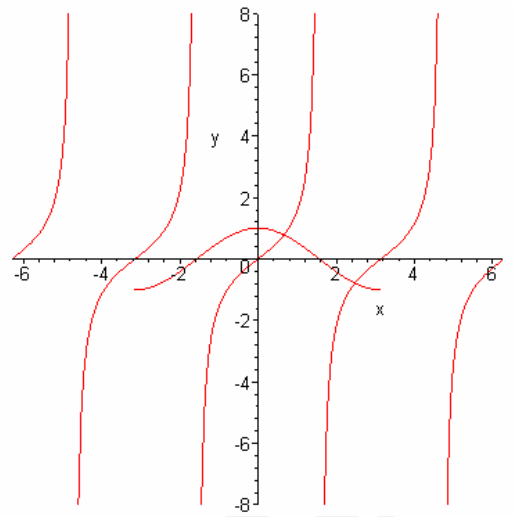
```
> with(plots):
  F:=plot(cos(x),x=-Pi..Pi,y=-Pi..Pi,style=line):
  G:=plot(tan(x),x=-Pi..Pi,y=-Pi..Pi,style=point):
  display({F,G},axes=boxed,scaling=constrained);
  F:=plot3d(4*x^2-4*x*y+y^2,x=-Pi..Pi,y=-Pi..Pi):
  G:=plot3d(x + y,x=-Pi..Pi,y=-Pi..Pi):
  display({F,G});
```

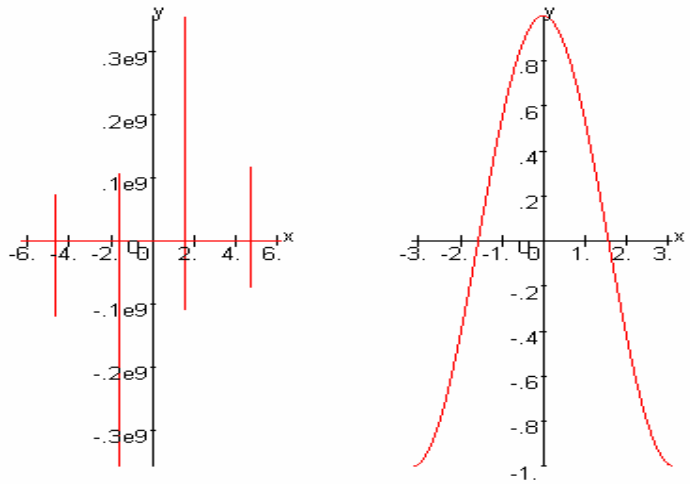




Ejemplo usando arrays:

```
> A := array(1..2):
  A[1]:= plot(tan(x),x=-2*Pi..2*Pi,y =-8..8,discont=true):
  A[2]:= plot(cos(x),x=-Pi..Pi,y=-1..1):
  display(seq(A[j],j=1..2));
  display(A);
```





***NOTA:** Vemos que al dibujar el array (display(A)), dibuja en dos planos diferentes, y sin embargo al pasarle las dos gráficas (plot) como argumentos dibuja las dos funciones en el mismo plano. El problema al pasarle el array como argumento es que no podemos establecer el rango de "y" que queremos dibujar.

Por otro lado, hemos pasado el argumento **discont=true**. Esto sirve para que dibuje bien cerca de las asintotas verticales (si no ponemos este argumento en las discontinuidades dibujará una linea vertical).

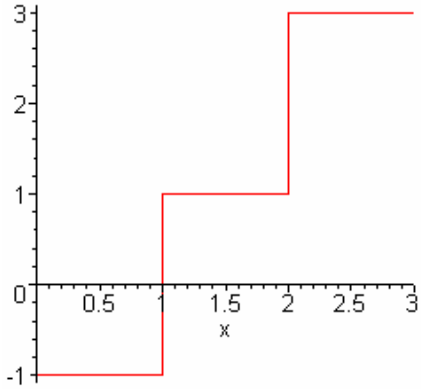
5.2.1. 2 Dimensiones

Ya hemos visto en los ejemplos anteriores como dibujar funciones. Veamos un ejemplo con una función por partes:

```
> f:=x->piecewise(x<1,-1,x<2,1,3);
```

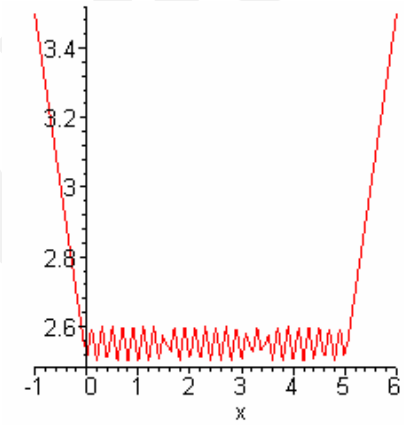
$$f := x \rightarrow \text{piecewise}(x < 1, -1, x < 2, 1, 3)$$

```
> plot(f(x),x=0..3);
```

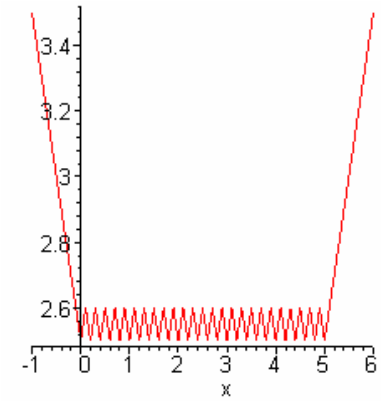


Maple para dibujar las funciones calcula el valor de la función en unos puntos. Para aumentar el número de puntos se puede poner como argumento numpoints=n. Por ejemplo:

```
> plot(sum((-1)^(i)*abs(x-i/10),i=0..50),x=-1..6);
```



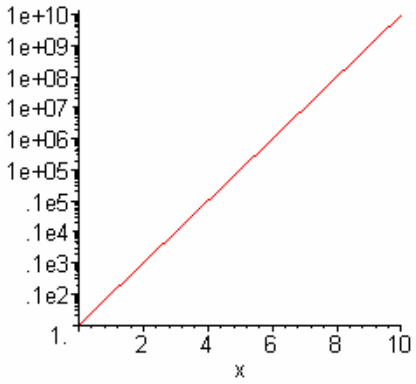

```
> plot(sum((-1)^(i)*abs(x-i/10),i=0..50),x=-1..6,numpoints=500);
```



- **Funciones en ejes logarítmicos**

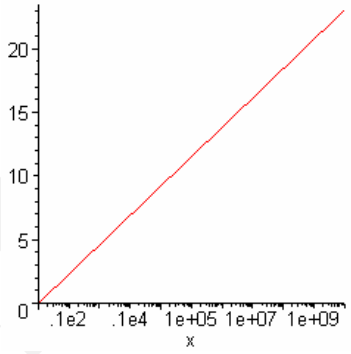
Para que el eje de coordenadas sea de escala logarítmica, se usa la función *logplot*(f(x),xrange, yrange). xrange e yrange(opcional) son los rangos de x e y respectivamente.

```
> logplot(10^x,x=0..10);
```



Para que el eje de abcisas sea de escala logarítmica se utiliza la función semilogplot(f(x),xrange,yrange) siendo yrange opcional.

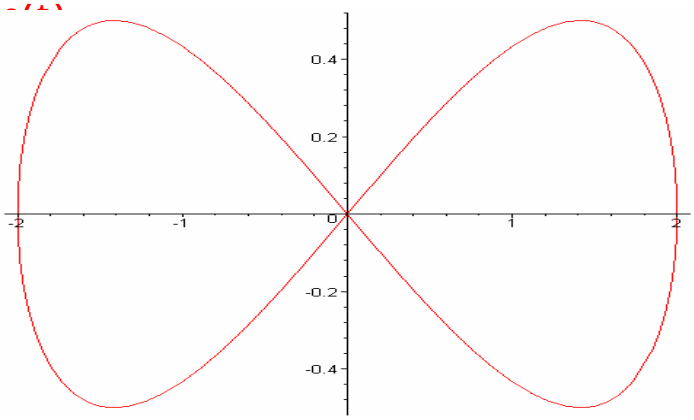
```
> semilogplot(log(x),x=1..1e10);
```



- **Funciones Paramétricas**

Se pueden representar también funciones paramétricas (dos ecuaciones, función de un mismo parámetro) definiéndolas en forma de lista (atención a los corchetes [ ], que engloban tanto a las expresiones como al parámetro y su rango de valores):

```
> plot([2*sin(t), sin(t)*cos(t), t=0..2*Pi]);     # x=2*sin(t)
y=sin(t)*co
```

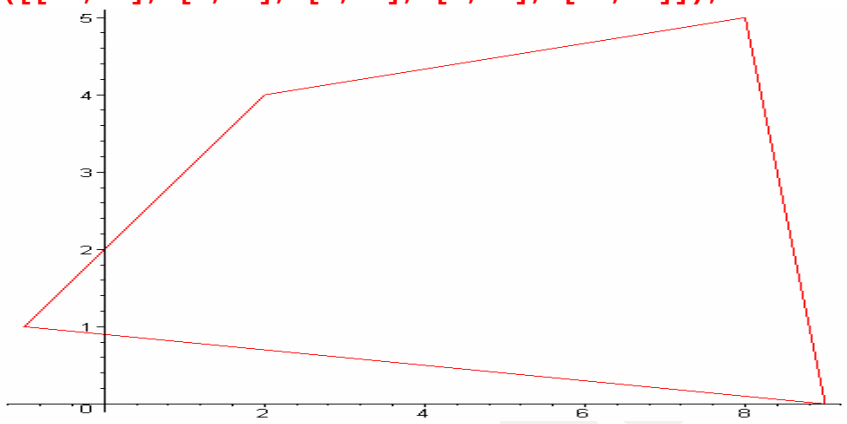


Para no ver el dibujo distorsionado tiene que añadir la opción **scaling=constrained** o bien clicar sobre el icono correspondiente.

- **Lineas Poligonales**

Son conjuntos de puntos unidos por líneas rectas bien mediante la función ***plot,*** bien mediante ***polygonplot*** de la siguiente manera: las dos coordenadas de cada punto se indican de forma consecutiva entre corchetes [ ], en forma de lista de listas. Obsérvese el siguiente ejemplo, en el que se dibuja un cuadrilátero:

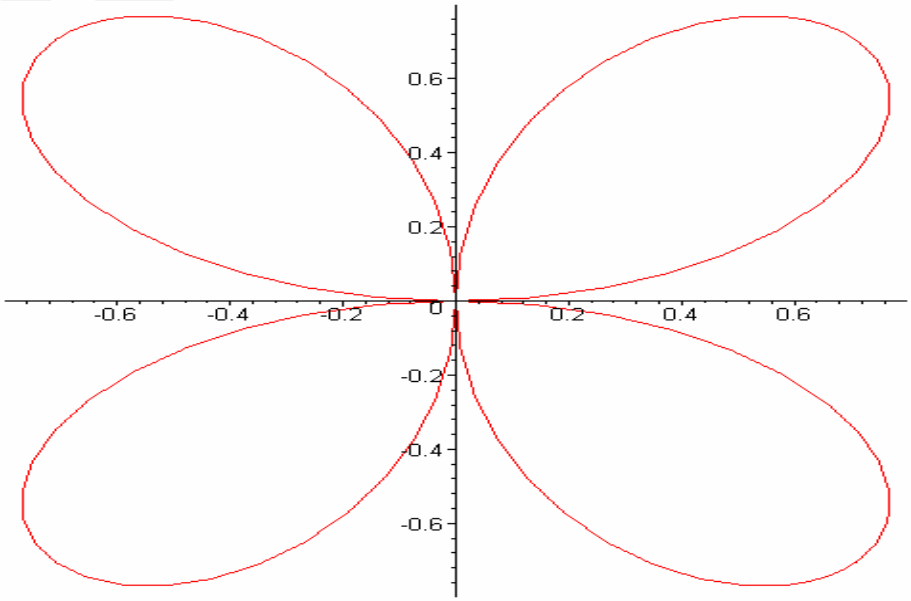
```
> plot([[-1, 1], [2, 4], [8, 5], [9, 0], [-1, 1]]);
```



- **Funciones en polares**

Se hace mediante la función polarplot. Esta función nos permite dibujar una o más curvas en un espacio bidimensional dadas unas coordenadas polares. La sintaxis es polarplot(L, options), donde L es un conjunto de curvas bidimensionales y options son las opciones del gráfico a las que se puede acceder mediante el botón derecho del ratón. Lo que representa la función de L es el radio (distancia al centro), y la variable independiente es el angulo. Ej:

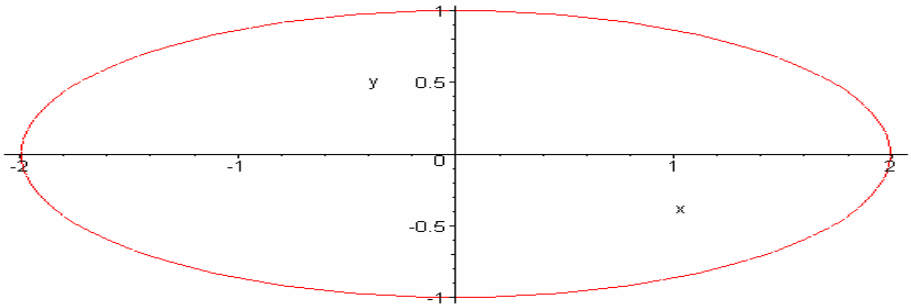
```
> with(plots):
polarplot(sin(2*theta));   #en vez de theta se puede poner
                           cualquier otra variable
```



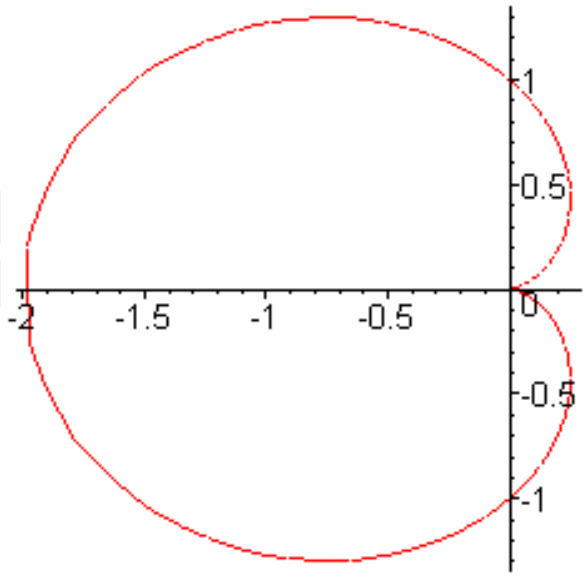
- **Funciones en implicitas**

Se utiliza la función ***implicitplot.*** Nos permite dibujar curvas en dos dimensiones de expresiones dadas de manera implícita. La sintaxis es ***implicitplot*** *(***expr1**, **x**=**a**..**b**, **y**=**c**..**d**, **options***)*. En ambas definimos los rangos en lo que se quiere trabajar. Ejemplo:

```
> with(plots):
implicitplot((x^2)/4 + y^2 = 1,x=-4..4,y=-1..1);
```



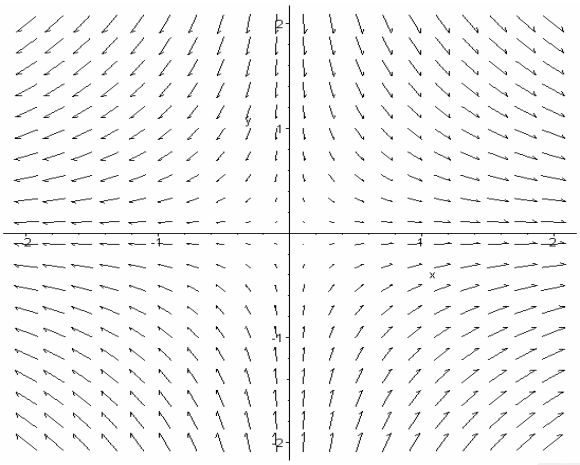

```
> implicitplot(r = 1 - cos(theta), r=0..2,theta=0..2*Pi,coords=polar);
```



- **Campos vectoriales**

Se dibujan mediante la función ***fieldplot***. La sintaxis es **fieldplot(f**, **r1**, **r2).** f es el vector o conjunto de vectores que se quiere representar, r1 y r2 son los rangos del campo vectorial. En el ejemplo se ve mejor:

```
> with(plots):
fieldplot( [x/(x^2+y^2+4)^(1/2),-y/(x^2+y^2+4)^(1/2)],x=-2..2,y=-
2..2);  #donde los dos primeros elementos son las componentes de
los vectores.
```



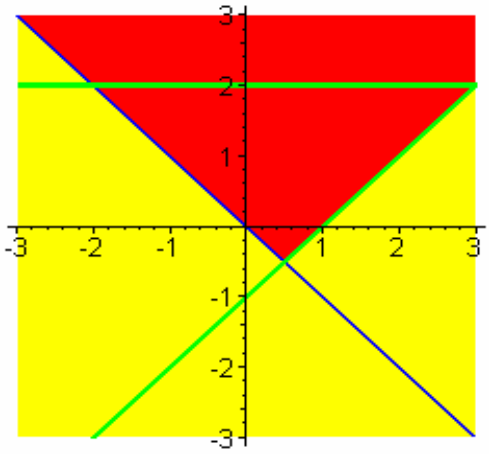
- **Sistemas de inecuaciones**

Los sistemas de inecuaciones de 2 variables se representan mediante la función ***inequal***. La sintaxis es inequal(ineqs**,** xspec**,** yspec**,** options), xspec e yspec son los rangos en los que se representa y options son las siguientes 4 opciones que tenemos en el gráfico:

feasible region	región factible, esto es, que satisface todas las inecuaciones.
excluded regions	región excluida, que no cumple al menos una inecuación.
open lines	para representar una línea frontera abierta,que no pertenece al campo de la solución
closed lines	para representar una línea frontera cerrada, que pertenece a la solución.

Ejemplo:

```
> inequal( { x+y>0, x-y<=1,y=2}, x=-3..3, y=-3..3,
 optionsfeasible=(color=red),
 optionsopen=(color=blue,thickness=2),
 optionsclosed=(color=green, thickness=3),
 optionsexcluded=(color=yellow) );
```



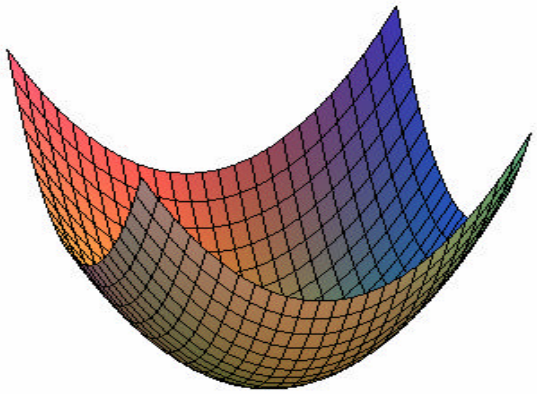
5.2.2. 3 Dimensiones

- **plot3d:** Podemos representar gráficos tridimensionales definiendo una expresión de 2 variables en cuyo caso hay que definir los rangos en los que queremos representarla, o bien definiéndola como función de 2 variables, en este caso pondremos los rangos sin indicar la variable, se entiende que van en el orden de definición de la función. Por ejemplo:

```
> f:=(x,y)->x^2+y^2;
```

$$f := (x, y) \rightarrow x^2 + y^2$$

```
> plot3d(f,-2..2,-2..2);
```



Si clicamos sobre la gráfica anterior aparecerán unos botones en la barra de herramientas:

En primer lugar aparecen, de izquierda a derecha, 2 botones para girar la figura respecto 2 direcciones. Otra forma de cambiar el punto de vista de los gráficos 3-D es clicar sobre la figura y ? sin soltar el botón del ratón? arrastrar en cualquier dirección.

Después aparecen 7 botones que permiten controlar cómo se dibuja la superficie 3-D correspondiente. Se puede dibujar con polígonos, con líneas de nivel, en hilo de alambre (wireframe), simplemente con colores, o en algunas combinaciones de las formas anteriores. Si la imagen no se redibuja automáticamente en el nuevo modo, hay que hacer un doble clic sobre ella.

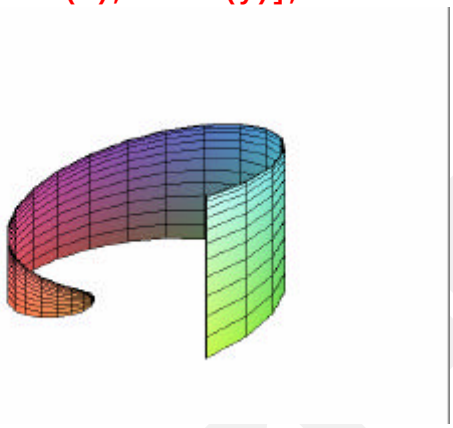
A continuación aparecen 4 botones que permiten controlar la forma en la que aparecen los ejes de coordenadas.

Finalmente hay un botón para controlar que se dibuje con la misma escala según los tres ejes.

En la barra de menús aparecen nuevas e interesantes posibilidades, tales como cambiar los criterios de color utilizados, pasar de perspectiva paralela a cónica, etc. La mejor forma de conocer estas capacidades de Maple es practicar sobre ellas, observando con atención los resultados de cada opción. Estas opciones pueden también introducirse directamente en el comando ***plot3d*** con el que se realiza el dibujo.

Se pueden mostrar ***funciones paramétricas*** (dependientes de dos parámetros) análogamente a como se hacía en el caso bidimensional (obsérvese que en este caso los rangos se definen fuera de los corchetes [ ]):

```
> plot3d([x*sin(x), x*cos(x), x*sin(y)], x=0..2*Pi, y=0..Pi);
```



En el caso de las gráficas tridimensionales aumenta notablemente el número de opciones o parámetros de Maple que puede controlar el usuario. Aquí sólo se van a citar dos, pero para más información se puede teclear ?plot3d[options].

La opción shading permite controlar el coloreado de las caras. Sus posibles valores son: XYZ, XY, Z, Z_GREYSCALE, Z_HUE o NONE.

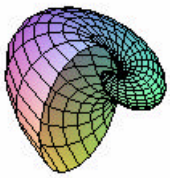
Con la opción *light* se controlan las luces que enfocan a la figura. Los dos primeros valores son los ángulos de enfoque en coordenadas esféricas, y los tres siguientes definen el color de la luz, correspondiendo los coeficientes –entre 0 y 1– al rojo, verde y azul, respectivamente. A continuación se presentan dos ejemplos para practicar, pudiendo el usuario modificar en ellos lo que le parezca.

```
> plot3d((x^2-y^2)/(x^2+y^2), x=-2..2, y=-2..2, shading=XYZ,
title='saddle');
> plot3d(sin(x*y), x=-2..2, y=-2..2, color=BLUE, style=PATCH,
light=[45, 45, 0, 1, 0.4]);
```

- **Coordenadas esféricas**

  Se utiliza la función ***sphereplot(r-expr, theta=range, phi=range)***. El primer argumento es la expresión que define el radio y los dos siguientes definen el rango de los ángulos. Veamos dos ejemplos:

```
> sphereplot((4/3)^theta*sin(phi),theta=-1..2*Pi,phi=0..Pi);
```



También se pueden representar las funciones en esféricas paramétricamente:

```
> sphereplot([exp(s)+t,cos(s+t),t^2],s=0..2*Pi,t=-2..2);
```

- **Coordenadas cilíndricas**

  Se utiliza ***cylinderplot (r-expr, angle=range, z=range).*** Veamos un ejemplo:

```
> cylinderplot(theta,theta=0..4*Pi,z=-1..1);
```

También se puede expresar en paramétricas.

5.2.3. Animaciones

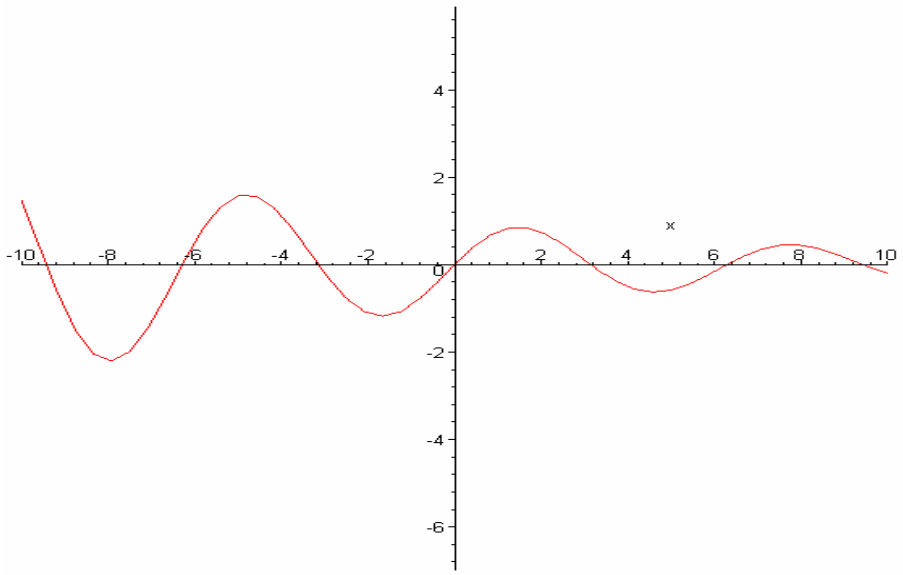
Maple realiza *animaciones* con gran facilidad. En las animaciones se representa una función que varía en el tiempo o con algún parámetro. Este parámetro es una nueva

variable que hace falta introducir. Las animaciones bidimensionales tienen una variable espacial y otra variable temporal, y ambas son independientes. Para obtener una animación hay que definir los rangos de esas dos variables. La sintaxis de la función es la siguiente, animate(**F**, **x**, **t**), donde F es la función que se desea visualizar, x el rango en el que se trabaja y t es el rango del parámetro de frames. Las animaciones de Maple quedan insertadas, al igual que las gráficas, en la hoja de Maple. Si clicamos sobre ella, queda seleccionada y aparecen unos botones en la barra de herramientas, junto con unos menús adicionales en la barra de menús.

Como puede observarse, los botones son parecidos a los de un vídeo. Los dos primeros botones cambian la orientación de la figura. Los siguientes dos son el ***Stop*** y ***Start.*** Las funciones de los siguientes tres botones son, respectivamente: mover al siguiente frame, establecer la dirección de la animación hacia atrás y establecer la dirección hacia delante. Los siguientes dos botones decrecen y aumentan la velocidad

de animación (frames/segundo). Finalmente, los dos últimos botones establecen la animación como de único ciclo o ciclo continuo. Veamos un ejemplo en el cual mediante la función display, vista anteriormente, podemos visualizar los distintos frames de una animación. Ej:

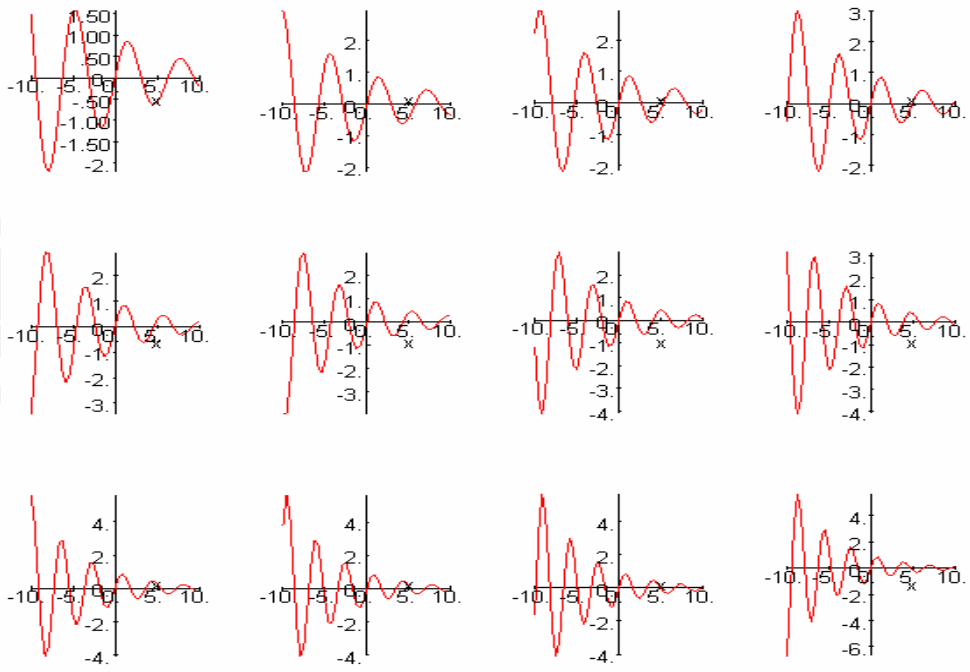
```
> with(plots):
animate( exp(-0.1*x*t)*sin(x*t),x=-10..10,t=1..2,frames=12);
```



Cuando ejecutamos este comando obtenemos una animación convencional que podemos ejecutar con los controles anteriormente explicados. En cambio para poder verlos sobre papel la siguiente manera resulta muy útil, lo que hacemos es visualizar cada frame por separado mediante la función display:

```
> display(%);
```

Estamos haciendo referencia al comando anterior.



- **animate3d:** El caso tridimensional es análogo al bidimensional, la sintaxis es en este caso animate3d(**F**, **x**, **y**, **t**), F puede ser una función, un procedimiento o una función paramétrica, los demás argumentos son las dos variables espaciales y la temporal. Ej:

```
> with(plots):
animate3d(cos(t*x)*sin(t*y),x=-Pi..Pi, y=-Pi..Pi,t=1..2, frames=4):
> display(%);
```

5.3. FUNCIONES PARA ESTUDIANTES. *(STUDENT)*

En este paquete se ofrecen una serie de subpaquetes con los que se pretende ayudar al estudiante de matemática en el estudio y la comprensión de la materia dada. El paquete Student ha sustituido a su antecesor, student (con minúscula), ya que contiene las mismas funciones y algunas mejoras respecto al anterior.

5.3.1. Subpaquete *Calculus1*

Contiene material para el estudio del análisis de una variable. Este paquete presenta dos características principales, el *single step computation* y la visualización, aunque podemos encontrar también funciones que no tienen relación con estas componentes.

```
> with(Student[Calculus1]);
```

[*AntiderivativePlot, AntiderivativeTutor, ApproximateInt, ApproximateIntTutor, ArcLength, ArcLengthTutor, Asymptotes, Clear, CriticalPoints, CurveAnalysisTutor, DerivativePlot, DerivativeTutor, DiffTutor, ExtremePoints, FunctionAverage, FunctionAverageTutor, FunctionChart, FunctionPlot, GetMessage, GetNumProblems, GetProblem, Hint, InflectionPoints, IntTutor, Integrand, InversePlot, InverseTutor, LimitTutor, MeanValueTheorem, MeanValueTheoremTutor, NewtonQuotient, NewtonsMethod, NewtonsMethodTutor, PointInterpolation, RiemannSum, RollesTheorem, Roots, Rule, Show, ShowIncomplete, ShowSteps, Summand, SurfaceOfRevolution, SurfaceOfRevolutionTutor, Tangent, TangentSecantTutor, TangentTutor, TaylorApproximation, TaylorApproximationTutor, Understand, Undo, VolumeOfRevolution, VolumeOfRevolutionTutor, WhatProblem*]

A continuación veremos algunas de las funciones más interesantes.

- **Rule:** Aplica una determinada regla a un problema de cálculo. La sintaxis de la función es Rule[**rule**](**expr**), donde *rule* es la regla que se quiere aplicar y *expr* es la expresión algebraica del. Las reglas que se pueden aplicar son las siguientes:

- **Reglas de diferenciación:**

Rule	**Alternate Names**	**Description**
`Chain`		`f(g(x)))' = f'(g(x))*g'(x)`
`constant`		`c' = 0`
`constantmultiple`	`` `c*` ``	`(c*f)' = c*f'`
`difference`	`` `-` ``	`(f-g)' = f'-g'`
`identity`	`` `^` ``	`x' = 1`
`int`	`Int`	`Int(f(t),t=c..x)' = f(x)`
`power`	`` `^` ``	`(x^n)' = n*x^(n-1)`
`product`	`` `*` ``	`(f*g)' = f'*g + f*g'`
`quotient`	`` `/` ``	`(f/g)' = (g*f' - f*g')/g^2`
`sum`	`` `+` ``	`(f+g)' = f'+g'`

- **Reglas de integración**: (se pueden llamar por el nombre entre paréntesis también)

Constant	Int(c, x)= c*x
	Int(c, x=a..b) = c*b - c*a
constantmultiple (`c*`)	I(c*f(x)) = c*I(f(x))
diff (Diff)	Int(Diff(f(x), x), x) = f(x)
	Int(Diff(f(t), t), t=a..x) = f(x)

difference (`-`)	I(f(x)-g(x)) = I(f(x)) - I(g(x))
identity	Int(x, x) = x^2/2
	Int(x, x=a..b) = b^2/2 - a^2/2
partialfractions	I(f(x)) = I(R1(x)+R2(x)+...) where
	R1(x)+R2(x)+... es una descomposición en fracciones parciales de f(x)
power (`^`)	Int(x^n, x) = x^(n+1)/(n+1)
	Int(x^n,x=a..b) = b^(n+1)/(n+1)-a^(n+1)/(n+1)
revert	deshace un cambio de variables
solve	resuelve una ecuación en la que aparece la misma integral más de una vez
sum	I(f(x)+g(x)) = I(f(x)) + I(g(x))

y exclusivamente para integrales definidas:

flip	Int(f(x),x=a..b) = -Int(f(x),x=b..a)
join	Int(f(x),x=a..c) + Int(f(x),x=c..b) = Int(f(x),x=a..b)
split	Int(f(x),x=a..b) = Int(f(x),x=a..c) + Int(f(x),x=c..b)

Para aplicar split hay que especificar el punto intermedio ('c' en este caso). Rule[split,c].

Después hay tres reglas más que no pueden utilizarse con "understand"(más adelante veremos lo que es) porque necesitan más parametros que el resto de las reglas.Las 3 reglas son:

parts: Aplica la regla de descomposición por partes, es decir:

$$\int f(x)\left(\frac{d}{dx}g(x)\right)dx = f(x)\,g(x) - \int g(x)\left(\frac{d}{dx}f(x)\right)dx$$

Hay que especificarle cuales son f(x) y g(x). Su forma es: Rule[parts,f(x),g(x)].

rewrite: Cambia la forma del integrando, pero no varía la variable de integración. Su forma es: Rule[rewrite, f1(x) = g1(x), f2(x) = g2(x), ...]. Así, cambiará f1(x) por g1(x), f2(x) por g2(x).Por ejemplo podríamos hacer el cambio sin^2(x)=(1-cos(2x))/2.

change: Sirve para hacer un cambio de variable. Sus formas son:

```
[change, F(x,u) = G(x,u), u, siderels]
[change, F(x,u) = G(x,u), u = H(x), siderels]
```

El primer parámetro F(x,u)=G(x,u) establece cuál es la relación entre x y u. Puede tener la esta forma general, aunque será más común que tenga la forma x=G(u) o u=F(x). El segundo parámetro es opcional y puede utilizarse solamente para especificar cuál es la nueva variable (el primer caso), o podemos especificar cuál es el cambio inverso de variable (segundo caso), es decir si el primer parámetro es x=F(u), podemos poner en el segundo parámetro $u=F^{-1}(x)$. Hay que tener cuidado, ya que el programa no verifica si la relación es correcta. Al hacer el cambio hay que procurar que la nueva variable 'u' no esté previamente utilizada, ya que puede darnos problemas.
En las integrales definidas, la rutina Rule determina si los extremos cogerán los valores de los nuevos valores o si dejará en función de la variable antigua (x=a , donde x es la variable original). Para deshacer el cambio habrá que utilizar la regla *revert* en la forma Rule [revert]. Veamos un ejemplo:

```
> Understand(int,`c*`); #luego veremos para qué sirve.
> Rule[change,u=sqrt(x)](Int(sin(sqrt(x))/sqrt(x), x=a..b));
Creating problem #4
Applying substitution x = u^2, u = x^(1/2) with dx = 2*u*du, du =
1/2/x^(1/2)*dx
```

$$\int_a^b \frac{\sin(\sqrt{x})}{\sqrt{x}}\,dx = 2\int_{x=a}^{x=b} \sin(u)\,du$$

```
> Rule[sin](%);
```

$$\int_a^b \frac{\sin(\sqrt{x})}{\sqrt{x}}\,dx = 2\,(-\cos(u))\Big|_{x=a\,..\,b}$$

```
> Rule[revert](%);
Reverting substitution using u = x^(1/2)
```

$$\int_a^b \frac{\sin(\sqrt{x})}{\sqrt{x}}\,dx = -2\cos(\sqrt{b}) + 2\cos(\sqrt{a})$$

- **Reglas de límites:**

constant	L(c)= c
constantmultiple (`c*`)	L(c*f(x)) = c*L(f(x))
difference (`-`)	L(f(x)-g(x)) = L(f(x)) - L(g(x))
identity	L(x) = a
power (`^`)	L(f(x)^n) = L(f(x))^n
	L(f(x)^g(x)) = L(f(x)) ^ L(g(x))
product (`*`)	L(f(x)*g(x)) = L(f(x)) * L(g(x))

quotient (`/ `)	L(f(x)/g(x)) = L(f(x)) / L(g(x))
sum (`+ `)	L(f(x)+g(x)) = L(f(x)) + L(g(x))
lhopital	aplica la regla de l'Hopital
rewrite	cambia la forma de la expresión del límite
change	cambio de variable

Estas tres últimas reglas no se pueden aplicar con 'understand'. *rewrite* y *change* se aplican como en las integrales. La regla rewrite se puede utilizar para sustituir expresiones por sus asintóticamente equivalentes.

- **Hint:** Esta función nos devuelve qué regla podemos aplicar para solucionar el problema que se le plantea. La sintaxis es **Hint(expr).** Conviene tener en cuenta que Maple está limitado, muchas veces nos sugerirá hacer cosas que, aunque sean correctas, pueden complicar el problema en exceso. Un enfoque distinto del problema tal vez lo solucione de manera más rápida y limpia.

  Para utilizar la propuesta que nos hace Maple, bastará con hacer:

```
> Limit((2^n-3^n)/(ln(n)),n=infinity);
```

$$\lim_{n \to \infty} \frac{2^n - 3^n}{\ln(n)}$$

```
> Hint(%);
```

$$[\,lhopital,\ 2^n - 3^n\,]$$

```
> Rule[%](%%);
```

$$\lim_{n \to \infty} \frac{2^n - 3^n}{\ln(n)} = \lim_{n \to \infty} \left(2^n \ln(2) - 3^n \ln(3)\right) n$$

A veces puede suceder que Maple nos aconseje más de una regla. Por ejemplo:

```
> h := Hint(%);   #% no se refiere al limite anterior, sino a otro
problema #vemos que le damos un nombre a la propuesta.
```

$$h := [\,change,\ u = 1 - x^2,\ u\,],\ [\,change,\ 1 - x^2 = u^2,\ u\,],\ \left[\,change,\ u = \sqrt{\frac{1-x}{1+x}},\ u\,\right]$$

Entonces para acceder a la segunda regla por ejemplo, se realizaría:

```
> Rule[h[2]](%);
```

Para acceder a la primera bastaría con poner Rule[%](%);

CONSEJOS

Si se quiere conocer lo que se está haciendo en Maple hay que escribir `>infolevel[Calculus1] := 1:`. Así, si se desea que utilice una regla que no se puede aplicar, saldrá un mensaje informativo. En los ejemplos se verán algunos mensajes comunes.

Por otro lado, es conveniente cuando el problema es bastante complicado, indicar al programa que aplique unas reglas automáticamente, sin que le tengamos que mandar aplicarla. Por ejemplo, si tenemos que calcular una derivada podemos pedir que aplique automáticamente la regla de la constante, la de la potencia, la del seno y la de la multiplicación por una constante. Veamos un ejemplo:

```
> Understand(Diff, constant, `c*`, sin, power);
```

$$Diff = [constant, constantmultiple, sum, identity]$$

```
> Diff(sin(x)/(5*x^2),x);
```

$$\frac{d}{dx}\left(\frac{1}{5}\frac{\sin(x)}{x^2}\right)$$

```
> Rule[quotient](%);
Creating problem #5
```

$$\frac{d}{dx}\left(\frac{1}{5}\frac{\sin(x)}{x^2}\right)=\frac{1}{25}\frac{5\cos(x)x^2-10\sin(x)x}{x^4}$$

Si no hubiésemos puesto la sentencia de *Understand*:

```
> Diff(sin(x)/(5*x^2),x);
```

$$\frac{d}{dx}\left(\frac{1}{5}\frac{\sin(x)}{x^2}\right)$$

```
> Rule[quotient](%);
Creating problem #4
```

$$\frac{d}{dx}\left(\frac{1}{5}\frac{\sin(x)}{x^2}\right)=\frac{1}{25}\frac{5\left(\frac{d}{dx}\sin(x)\right)x^2-\sin(x)\left(\frac{d}{dx}(5x^2)\right)}{x^4}$$

Hay que especificarle el tipo de problema (derivada, integral o límite) en la primera posición del paréntesis.

También puede ser útil la secuencia "Undo", ya que nos permitirá deshacer cambios que le hayamos hecho al problema, por ejemplo cuando aplicamos una regla que nos complica más la expresión:

```
> Int(x/sqrt(1-x^2),x=a..b);
```

$$\int_a^b \frac{x}{\sqrt{1-x^2}}\,dx$$

```
> Rule[change,u=1-x^2,u](%);
Creating problem #3

Applying substitution x = (1-u)^(1/2), u = 1-x^2
with dx = -1/2/(1-u)^(1/2)*du, du = -2*x*dx
```

$$\int_a^b \frac{x}{\sqrt{1-x^2}}\,dx = \int_{1-a^2}^{1-b^2} -\frac{1}{2\sqrt{u}}\,du$$

```
> Undo(%);
```

$$\int_a^b \frac{x}{\sqrt{1-x^2}}\,dx$$

- **Special Points:** Son puntos especiales de una función. Pueden obtenerse *CriticalPoints*, *ExtremePoints*, *InflectionPoints y Roots* (raíces). La forma de todas estas funciones son: SpecialPoints(f(x),x=a..b). En lugar de x=a..b también podemos poner x a secas para que saque los puntos especiales en todo el campo real. Ejemplos:

```
> f:=x->sin(x);
```

$$f := x \to \sin(x)$$

```
> Roots(f(x),x=0..8);
```

$$[0, \pi, 2\,\pi]$$

```
> CriticalPoints(f(x),0..8);
```

$$\left[\frac{\pi}{2}, \frac{3\,\pi}{2}, \frac{5\,\pi}{2}\right]$$

```
> ExtremePoints(f(x),0..8);
```

$$\left[0, \frac{\pi}{2}, \frac{3\,\pi}{2}, \frac{5\,\pi}{2}, 8\right]$$

```
> InflectionPoints(f(x),0..8);
```

$$[0, \pi, 2\,\pi]$$

- **Assymptotes**: Nos devuelve las ecuaciones que describen las asíntotas de la función. Su forma es: Asymptotes(**f(x)**, **x** = **a**..**b**, **y**). El segundo argumento funciona igual que en el apartado anterior. y es la variable dependiente que queremos que esté en las ecuaciones de las asíntotas. Ej:

```
> Asymptotes(1/(X - 3) + 2*x, x);
```

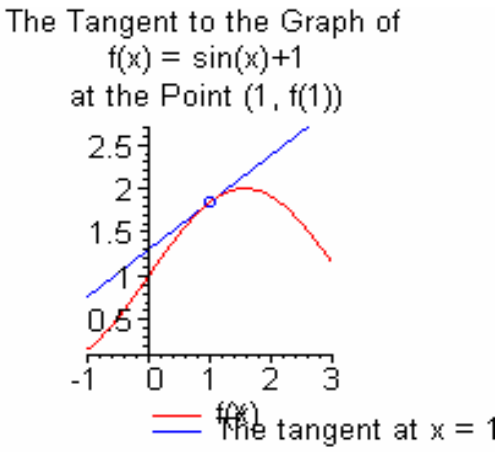
$$[y = 2\,x, x = 3]$$

- **Tangent**: Devuelve la recta tangente de la función en el punto que le hayamos especificado. Su forma es: Tangent(**f(x)**, **x** = **c).**

```
> Tangent(sin(x) + 1, x = 1);
```

$$x\cos(1) + \sin(1) + 1 - \cos(1)$$

```
> Tangent(sin(x) + 1, x = 1,-1..3,output=plot);
```



- **TaylorApproximation**: Hace la aproximación de una función alrededor de un punto. Su forma es: TaylorApproximation(**f(x)**, **x** = **c,order=n),** siendo order el orden que queremos que tenga la función.

```
> TaylorApproximation(sin(x),x=1,order=5);
```

$$\frac{13}{24}\sin(1)+\frac{13}{24}x\cos(1)-\frac{101}{120}\cos(1)-\frac{1}{4}\sin(1)x^2+\frac{5}{6}\sin(1)x-\frac{1}{12}\cos(1)x^3$$
$$+\frac{5}{12}\cos(1)x^2+\frac{1}{24}\sin(1)x^4-\frac{1}{6}\sin(1)x^3+\frac{1}{120}\cos(1)x^5-\frac{1}{24}\cos(1)x^4$$

5.3.2. Subpaquete MultivariateCalculus

Este subpaquete es nuevo en Maple 9.5. En él hay funciones para tratar funciones multivariable, es decir, funciones de R^n en R.

```
> with(Student[MultivariateCalculus]);
```

[*ApproximateInt, ApproximateIntTutor, CenterOfMass, ChangeOfVariables, CrossSection, CrossSectionTutor, DirectionalDerivative, DirectionalDerivativeTutor, FunctionAverage, Gradient, GradientTutor, Jacobian, LagrangeMultipliers, MultiInt, Revert, SecondDerivativeTest, SurfaceArea, TaylorApproximation, TaylorApproximationTutor*]

Veamos algunas de estas funciones.

- **DirectionalDerivative:** Calcula la derivada direccional de una función de 2 ó 3 variables, es decir, calcula el producto escalar del gradiente con la dirección especificada. El primer argumento es la función, el segundo es el punto en el que queremos calcular la derivada y el tercer argumento es la dirección en que queremos calcular la derivada. Veamos un ejemplo:

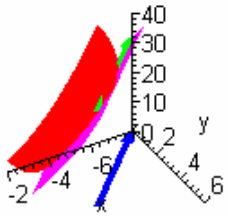
```
> with(Student[MultivariateCalculus]):
```

```
> DirectionalDerivative(x^2+y^2,[x,y]=[1,2],[3,4]);
```

$$\frac{22}{5}$$

Se pueden representar gráficamente el plano tangente, la dirección y el resultado:

```
> DirectionalDerivative(x^2+y^2, [x,y]=[-4,4], [-6,-6], x=-8..-2,
y=0..6, z=0..40, output = plot);
```



- **Gradient**: Calcula el vector gradiente en un punto especificado. Su forma es: Gradient(**f(x,y,..)**, **[x,y,..]=[a1,b1,..]),** donde f(x,y,…) es la función y [a1,b1,…] es el punto donde queremos conocer el gradiente. Ej.:

```
> with(Student[MultivariateCalculus]):
> Gradient(x^2+y^2,[x,y]=[0,1]);
```

$$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

También hay opcion de dibujarlo. Para más información ver el help.

- **Jacobian**: Calcula el jacobiano de una función vectorial de variables múltiples. Su forma es Jacobian([**f(x,y,..)**, **g(x,y,..)**, **..**], [**x**, **y**, **..**]), siendo [f(x,y…),g(x,y,..)..] las funciones escalares de la función vectorial, y [x,y,..] las variables independientes (si se igualan a un punto, se calcula el jacobiano en ese punto). Se puede calcular también el determinante poniendo como argumento 'output=determinant'. Ej:

```
> Jacobian([x+y^2, x-y],[x,y]);
```

$$\begin{bmatrix} 1 & 2\,y \\ 1 & -1 \end{bmatrix}$$

```
> Jacobian([z*x+y-4, z+x-y, z^2],[x,y,z]=[1,2,C]);
```

$$\begin{bmatrix} C & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 2\,C \end{bmatrix}$$

```
> Jacobian([z*x+y-4, z+x-y, z^2],[x,y,z]=[1,2,C], output =
determinant);
```

$$-2\,C^2 - 2\,C$$

- **LagrangeMultipliers:** Nos dan los máximos, mínimos y puntos silla de funciones condicionadas utilizando el método de Lagrange. La forma es: LagrangeMultipliers(**f(x,y,..)**, **[g(x,y,..), h(x,y,..),..]**, **[x,y,..]),** donde f(x,y,..) es la función en la que queremos buscar los puntos, y [g(x,y,..),h(x,y,…),…] son las condiciones, siendo éstas las funciones igualadas a 0 (condiciones: g(x,y,..)=0 , h(x,y,…)=0,…). [x,y..] son las variables independientes. Ej:

```
> LagrangeMultipliers(x*y,[x^2/8+y^2/2-1],[x,y]);
```

$$[2, 1], [-2, -1], [-2, 1], [2, -1]$$

También se pueden dibujar las funciones.

- **MultiInt**: Calcula integrales dobles y triples. Su forma es: MultiInt(**f(x,y,z)**, **x=a..b**, **y=c..d**, **z=e..f**, **opts**). El primer argumento es la función, los siguientes son los limites de integración. En opts, podemos especificar por ejemplo las coordenadas en las que trabajamos, que pueden ser: **coordinates =** *cartesian[x,y]*, *polar[r,theta]* (2-D), *cartesian[x,y,z]*, *cylindrical[phi,theta,z]*, *spherical[r,theta,phi]* . En polares, cilíndricas y esféricas la primera componente será siempre el radio. Si en opts ponemos output=integral, nos mostrará la integral sin sacar el valor.

```
> MultiInt(3*x^2+3*y^2,x=1..4,y=-
1..6,output=integral)=MultiInt(3*x^2+3*y^2,x=1..4,y=-1..6);
```

$$\int_{-1}^{6}\int_{1}^{4} 3x^2+3y^2\,dx\,dy = 1092$$

```
> MultiInt(r,r=1..4,t=0..Pi/2, coordinates=polar[r,t], output =
integral);
```

$$\int_{0}^{\frac{\pi}{2}}\int_{1}^{4} r^2\,dr\,dt$$

*NOTA: Vemos como multiplica la función por el determinante del jacobiano (en este caso r).

- **SecondDerivativeTest:** Nos dice si el punto en el que el gradiente era nulo es un mínimo, un máximo o un punto silla. Tiene la forma: SecondDerivativeTest(**f(x,y,...)**, **[x,y,...] = [[a,b,...], [c,d,...]]**, **opts**). Como vemos se pueden evaluar más de un punto a la vez. En opts, si ponemos output=hessian nos devuelve la matriz ***hessiana*** del punto. Ej.:

```
> f:=(x,y)->x^3-3*y^2+x*y;
```

$$f := (x, y) \rightarrow x^3 - 3y^2 + xy$$

```
> grad:=Gradient(f(x,y),[x,y]=[a,b]);
```

$$grad := \left[\begin{bmatrix} 3a^2+b \\ -6b+a \end{bmatrix}\right]$$

```
> sols:=solve({3*a^2+b=0,-6*b+a=0},{a,b});
```

$$sols := \{b = 0, a = 0\}, \{a = \frac{-1}{18}, b = \frac{-1}{108}\}$$

Estos son los puntos críticos, ahora veremos si son máximos, mínimos o puntos silla.

```
> SecondDerivativeTest(f(a,b),[a,b]=[[0,0],[-1/18,-1/108]]);
```

$$LocalMin = [\ ],\ LocalMax = \left[\left[\frac{-1}{18}, \frac{-1}{108}\right]\right],\ Saddle = [[0, 0]]$$

- **TaylorApproximation**: Nos hace la aproximación de Taylor. Su forma es: TaylorApproximation(**f(x,y,...)**, **[x,y,...]=[a,b,...]**, order), donde f(x,y) es la función, [a,b,…] es el punto en el que hacemos la aproximación, y 'order' es el orden del polinomio que queremos conseguir. Veamos un ejemplo:

```
> TaylorApproximation(sin(x+y),[x,y]=[1,0],5);
```

$$\frac{1}{24}\sin(1)\,y^4 + \frac{1}{6}\sin(1)\,(x-1)\,y^3 - \frac{1}{6}\cos(1)\,y^3 - \frac{1}{2}\cos(1)\,(x-1)\,y^2$$
$$+ \frac{1}{4}\sin(1)\,(x-1)^2\,y^2 - \frac{1}{2}\sin(1)\,y^2 + \cos(1)\,y - \frac{1}{2}\cos(1)\,(x-1)^2\,y$$
$$+ \frac{1}{6}\sin(1)\,(x-1)^3\,y - \sin(1)\,(x-1)\,y - \frac{1}{6}\cos(1)\,(x-1)^3 + \frac{1}{24}\sin(1)\,(x-1)^4$$
$$- \frac{1}{2}\sin(1)\,(x-1)^2 + \sin(1) + \cos(1)\,(x-1)$$

5.4. FUNCIONES ÁLGEBRA LINEAL. (*LinearAlgebra)*

Casi todas las funciones de Álgebra Lineal están en una librería que se llama ***LinearAlgebra***. En esta nueva versión se ha añadido este paquete, que en cierto modo sustituye a *linalg*, debido a esto nos limitaremos al nuevo paquete, aunque también se dispone del paquete *linalg* en esta versión. Si se intenta utilizar alguna función de esta librería sin cargarla previamente, Maple se limita a repetir el nombre de la función sin realizar ningún cálculo.

Para cargar todas las funciones de esta librería, se teclea el comando siguiente:

```
> with(LinearAlgebra);
```

[*&x, Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm, CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation, ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, Copy, CreatePermutation, CrossProduct, DeleteColumn, DeleteRow, Determinant, Diagonal, DiagonalMatrix, Dimension, Dimensions, DotProduct, EigenConditionNumbers, Eigenvalues, Eigenvectors, Equal, ForwardSubstitute, FrobeniusForm, GaussianElimination, GenerateEquations, GenerateMatrix, GetResultDataType, GetResultShape, GivensRotationMatrix, GramSchmidt, HankelMatrix, HermiteForm, HermitianTranspose, HessenbergForm, HilbertMatrix, HouseholderMatrix,*

IdentityMatrix, IntersectionBasis, IsDefinite, IsOrthogonal, IsSimilar, IsUnitary, JordanBlockMatrix, JordanForm, LA_Main, LUDecomposition, LeastSquares, LinearSolve, Map, Map2, MatrixAdd, MatrixExponential, MatrixFunction, MatrixInverse, MatrixMatrixMultiply, MatrixNorm, MatrixPower, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor, Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent, Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank, RationalCanonicalForm, ReducedRowEchelonForm, Row, RowDimension, RowOperation, RowSpace, ScalarMatrix, ScalarMultiply, ScalarVector, SchurForm, SingularValues, SmithForm, SubMatrix, SubVector, SumBasis, SylvesterMatrix, ToeplitzMatrix, Trace, Transpose, TridiagonalForm, UnitVector, VandermondeMatrix, VectorAdd, VectorAngle, VectorMatrixMultiply, VectorNorm, VectorScalarMultiply, ZeroMatrix, ZeroVector, Zip]

Algunos de esos nombres resultan familiares (como inverse, det, etc.) y otros no tanto. En cualquier caso, poniendo el cursor sobre uno cualquiera de esos nombres, en el menú Help se tiene a disposición un comando para obtener información sobre esa función concreta. Además, con el comando:

```
> ?LinearAlgebra;
```

Si sólo se desea utilizar una función concreta de toda la librería LinearAlgebra, se la puede llamar sin cargar toda la librería, dando al programa las "pistas" para encontrarla. Esto se hace con el comando siguiente:

```
> LinearAlgebra[funcion](argumentos);
```

Por ejemplo, para calcular el determinante de una matriz A, basta teclear:

```
> LinearAlgebra[Determinant](A);
```

5.4.1. Vectores y matrices

LinearAlgebra trabaja con matrices de todo tipo, además trabaja con datos de tipo numérico como pueden ser enteros, datos en coma flotante tanto reales como complejos y con datos simbólicos. Para construir una matriz disponemos del comando **Matrix**, la sintaxis es la siguiente, *Matrix(**r**, **c**, **init**, **ro**, **sc**, **sh**, **st**, **o**, **dt**, **f**, **a**)*. El primer argumento *r (opcional)* es el número de filas de la matriz, mientras que *c (opcional)* es el número de columnas, *init (opcional)* es el estado inicial de la matriz que se puede especificar mediante diversas maneras (para ver las distintas posibilidades es aconsejable acudir a la ayuda que Maple proporciona sobre el comando **Matrix**). El siguiente parámetro *ro (opcional)* es una variable de tipo boolean para definir si la matriz puede ser alterada o no y *f (opcional)* son los datos con los que se va a rellenar la matriz. Veamos unos ejemplos en los que construimos matrices:

```
> Matrix(2);
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

```
> Matrix(2,3);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
> Matrix(1..2,1..3,5);
```

$$\begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix}$$

```
> Matrix([[1,2,3],[4,5,6]]);
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
> Matrix(3,a);
```

$$\begin{bmatrix} a(1,1) & a(1,2) & a(1,3) \\ a(2,1) & a(2,2) & a(2,3) \\ a(3,1) & a(3,2) & a(3,3) \end{bmatrix}$$

Podemos rellenar la matriz por columnas del siguiente modo:

```
> A:=Matrix(<<1,2,3>|<4,5,6>|<7,8,9>>);
```

$$A := \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Podemos definir una función con la que definimos los elementos de la matriz

```
> f:= (i,j) -> x^(i+j-1): Matrix(2,2,f);
```

$$\begin{bmatrix} x & x^2 \\ x^2 & x^3 \end{bmatrix}$$

o definir los elementos independientemente.

```
> s:={(1,1)=0,(1,2)=1}: Matrix(1,2,s);
```

$$\begin{bmatrix} 0 & 1 \end{bmatrix}$$

Se puede acceder a los elementos de una matriz con sus índices de fila y columna separados por una coma y encerrados entre corchetes. Por ejemplo:

```
> s:={(1,1)=2, (1,2)=1, (2,1)=a, (2,2)=Pi}:
> H:=Matrix(2,2,s):
> H[2,1]; H[2,2]; H[1,1];
```

$$a$$

$$\pi$$

$$2$$

Las reglas para definir vectores en Maple son similares a las de las matrices, pero, teniendo en cuenta que hay un único subíndice, la sintaxis es muy parecida a la del comando Matrix, Vector[o](d, init, ro, sh, st, dt, f, a, o).

```
> Vector(2);
```

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
> Vector(1..3,u);
```

$$\begin{bmatrix} u(1) \\ u(2) \\ u(3) \end{bmatrix}$$

```
> Vector[row]([1,x^2+y,sqrt(2)]);
```

$$[1, x^2+y, \sqrt{2}]$$

```
> f:= (j) -> x^(j-1):Vector(3,f);
```

$$\begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}$$

```
> s:={1=0,2=1}:Vector(2,s);
```

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Si queremos sustituir unos solos elementos de la matriz, lo que hay que hacer es lo siguiente:

```
> A := Matrix([[9,9,9,9],[9,9,9,9],[9,9,9,9],[9,9,9,9]]);
```

$$A := \begin{bmatrix} 9 & 9 & 9 & 9 \\ 9 & 9 & 9 & 9 \\ 9 & 9 & 9 & 9 \\ 9 & 9 & 9 & 9 \end{bmatrix}$$

```
> A[1..2, 2..4] := Matrix([[5, 6], [7, 8]]);
```

$$A_{1\,..\,2,\,2\,..\,4} := \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```
> A;
```

$$\begin{bmatrix} 9 & 5 & 6 & 0 \\ 9 & 7 & 8 & 0 \\ 9 & 9 & 9 & 9 \\ 9 & 9 & 9 & 9 \end{bmatrix}$$

Para extraer filas y columnas de una matriz o vector se hace lo siguiente:

- Si es un vector llamado V: V[[1,3,5]] o V[[1,3..5]]. *(¡Atención, hay que poner doble corchete!*).

```
> A := Matrix([[9,9,9,9],[8,9,7,9],[9,6,5,4],[1,2,2,3]]);
```

$$A := \begin{bmatrix} 9 & 9 & 9 & 9 \\ 8 & 9 & 7 & 9 \\ 9 & 6 & 5 & 4 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

- Si se quiere obtener a partir de esta matriz una nueva matriz constituida por los dos primeros elementos de la primera fila y los dos primeros elementos de la fila tercera, escribiríamos:

```
> A[[1,3],1..2];
```

$$\begin{bmatrix} 9 & 9 \\ 9 & 6 \end{bmatrix}$$

```
> A[[2,1],1..3];
```

$$\begin{bmatrix} 8 & 9 & 7 \\ 9 & 9 & 9 \end{bmatrix}$$

Se puede observar que al utilizar la forma [2,1], al extraer respeta el orden que le hemos impuesto, extrayendo primero la segunda fila y después la primera (no funcionaría así si pusiesemos: [2..1,]).

Matrices Especiales

Mediante Maple podemos crear unas matrices especiales. Todas estas matrices se pueden encontrar en el Help, dentro de 'LinealAlgebraic package', en 'Constructors'. A continuación se comentan unas de las más utilizadas:

- **Matriz Diagonal:** DiagonalMatrix(**V**). V es el vector o lista que contiene los elementos de la diagonal. Por ejemplo:

```
> L := [<1, 2>, 3, <<4, 5>|<6,7>>];
```

$$L := \left[\begin{bmatrix} 1 \\ 2 \end{bmatrix}, 3, \begin{bmatrix} 4 & 6 \\ 5 & 7 \end{bmatrix}\right]$$

```
> DiagonalMatrix(L);
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 6 \\ 0 & 0 & 5 & 7 \end{bmatrix}$$

- **Matriz Identidad:** IdentityMatrix(r,c). Aunque existen más opciones entre las variables, estos dos son los más importantes. Aunque no sea una matriz identidad, se pueden crear matrices rectangulares con unos en la diagonal mediante r (filas) y c (columnas).

```
> IdentityMatrix(3,4);
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
> IdentityMatrix(2);
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

5.4.2. Sumas y productos de matrices y vectores

5.4.2.1 Suma de matrices y vectores

No se puede operar con matrices y vectores como con variables escalares. Por ejemplo, considérense las matrices siguientes:

```
> A:= Matrix(3,3,f); #recuérdese que f(i,j)=x^(i+j-1)
```

$$\begin{bmatrix} x & x^2 & x^3 \\ x^2 & x^3 & x^4 \\ x^3 & x^4 & x^5 \end{bmatrix}$$

```
> B:=Matrix(3,3,1);
```

$$B := \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
> A+B;
```

$$\begin{bmatrix} x+1 & x^2+1 & x^3+1 \\ x^2+1 & x^3+1 & x^4+1 \\ x^3+1 & x^4+1 & x^5+1 \end{bmatrix}$$

```
> evalm(B*A);
Error, (in rtable/Product) invalid arguments
```

Nos da error porque el operador * no actúa correctamente sobre matrices.

```
> evalm(B&*A);  #Ahora con &*
```

$$\begin{bmatrix} x+x^2+x^3 & x^2+x^3+x^4 & x^3+x^4+x^5 \\ x+x^2+x^3 & x^2+x^3+x^4 & x^3+x^4+x^5 \\ x+x^2+x^3 & x^2+x^3+x^4 & x^3+x^4+x^5 \end{bmatrix}$$

Lo primero que se observa en estos ejemplos es que los operadores normales no actúan correctamente cuando los operandos son matrices (o vectores). Algunos operadores, como los de suma (+) o resta (-), actúan correctamente como argumentos de la función evalm.

El operador producto (*) no actúa correctamente sobre matrices, ni siquiera dentro de evalm. Maple dispone de un operador producto -no conmutativo y que tiene en cuenta las dimensiones- especial para matrices es el operador &*. El ejemplo anterior muestra que este operador, en conjunción con evalm, calcula correctamente el producto de matrices. También se emplea este operador en el producto de matrices por vectores. En la ventana de la función evalm puede ponerse cualquier expresión matricial.

El producto también se puede realizar con el operador ' . ':

```
> M1 := Matrix([[1,0],[0,1]]);
```

$$M1 := \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
> M2 := Matrix([[3,3],[4,4]]);
```

$$M2 := \begin{bmatrix} 3 & 3 \\ 4 & 4 \end{bmatrix}$$

```
> M1 . M2;
```

$$\begin{bmatrix} 3 & 3 \\ 4 & 4 \end{bmatrix}$$

Y en el caso que queramos multiplicar una matriz por un escalar, entre el punto y el escalar habrá que interponer un espacio (en el caso de producto de dos matrices el espacio no es necesario aunque en el ejemplo se haya puesto):

```
> 3 .M2;
```

$$\begin{bmatrix} 9 & 9 \\ 12 & 12 \end{bmatrix}$$

La función ***evalm*** permite mezclar en una expresión matrices y escalares. En Maple el producto de una matriz por un escalar se realiza mediante el producto de cada elemento de la matriz por el escalar. Por el contrario, la suma o resta de una matriz y un escalar se realiza sumando o restando ese escalar a los elementos de la diagonal (aunque la matriz no sea cuadrada).

```
> evalm(A/x); evalm(A+y);
```

$$\begin{bmatrix} \frac{1}{x} & \frac{1}{x} & \frac{1}{x} \\ 1 & 1 & 1 \\ x & x & x \end{bmatrix}$$

$$\begin{bmatrix} 1+y & 1 & 1 \\ x & x+y & x \\ x^2 & x^2 & x^2+y \end{bmatrix}$$

5.4.2.2 Producto vectorial de vectores:

Se realiza mediante el operador &x. Veamos un ejemplo:

```
> u := <2,-1,4>;
```

$$u := \begin{bmatrix} 2 \\ -1 \\ 4 \end{bmatrix}$$

```
> v := <3,-2,1>;
```

$$v := \begin{bmatrix} 3 \\ -2 \\ 1 \end{bmatrix}$$

```
> u &x v;
```

$$\begin{bmatrix} 7 \\ 10 \\ -1 \end{bmatrix}$$

```
> v &x u;
```

$$\begin{bmatrix} -7 \\ -10 \\ 1 \end{bmatrix}$$

Se puede crear también el vector con la función *Vector*. Entre los vectores y el símbolo **&x** hay que dejar espacio.

5.4.3. Copia de matrices

Tampoco las matrices y vectores se pueden copiar como las variables ordinarias de Maple. Obsérvese lo que sucede con el siguiente ejemplo:

```
> B:=A;
```

$$B := \begin{bmatrix} x & x^2 & x^3 \\ x^2 & x^3 & x^4 \\ x^3 & x^4 & x^5 \end{bmatrix}$$

Aparentemente todo ha sucedido como se esperaba. Sin embargo, la matriz B no es una copia de A, sino un "alias", es decir, un nombre distinto para referirse a la misma matriz. Para comprobarlo, basta modificar un elemento de B e imprimir A:

```
> B[1,2]:=alpha;A;
```

$$B_{1,2} := \alpha$$

$$\begin{bmatrix} x & \alpha & x^3 \\ x^2 & x^3 & x^4 \\ x^3 & x^4 & x^5 \end{bmatrix}$$

Si se quiere sacar una verdadera copia de la matriz A hay que utilizar la función copy, en la forma:

```
> B:=copy(A):
```

Es fácil comprobar que si se modifica ahora esta matriz B, la matriz A no queda modificada.

5.4.4. Inversa y potencias de una matriz

Una matriz puede ser elevada a una potencia entera –positiva o negativa– con el operador (^), al igual que las variables escalares. Por supuesto, debe aplicarse a través de la función evalm. Por otra parte, la matriz inversa es un caso particular de una matriz elevada a (-1). Considérese el siguiente ejemplo:

```
> A:=Matrix([[23,123,7],[22,17,18],[1,2,6]]);
```

$$A := \begin{bmatrix} 23 & 123 & 7 \\ 22 & 17 & 18 \\ 1 & 2 & 6 \end{bmatrix}$$

```
> evalm(A^(-1));
```

$$\begin{bmatrix} \frac{-22}{4105} & \frac{724}{12315} & \frac{-419}{2463} \\ \frac{38}{4105} & \frac{-131}{12315} & \frac{52}{2463} \\ \frac{-9}{4105} & \frac{-77}{12315} & \frac{463}{2463} \end{bmatrix}$$

```
> evalm(A^3);
```

$$\begin{bmatrix} 185531 & 487478 & 126008 \\ 87904 & 163117 & 64252 \\ 5476 & 12010 & 4027 \end{bmatrix}$$

5.4.5. Funciones básicas del álgebra lineal

A continuación se describen algunas de las funciones más importantes de la librería Linear Algebra. Esta librería dispone de un gran número de funciones para operar con matrices, algunas de las cuales se describen a continuación. Además, existen otras funciones para casi cualquier operación que se pueda pensar sobre matrices y vectores: extraer submatrices y subvectores, eliminar o añadir filas y columnas, etc.

- **Adjoint**: Adjoint(A) nos permite calcular la matriz adjunta de la matriz cuadrada A.

```
> with(LinearAlgebra):
  A1 := <<9,4,1>|<1,3,-1>|<0,8,1>>:
  C1 := Adjoint(A1, outputoptions=[datatype=float]);
```

$$C1 := \begin{bmatrix} 11. & -1. & 8. \\ 4. & 9. & -72. \\ -7. & 10. & 23. \end{bmatrix}$$

- **Basis, SumBasis e IntersectionBasis**: En Basis le pasamos como argumento una lista de vectores, y la función nos devolverá una base del subespacio que forman los vectores. En SumBasis le pasamos una lista de subespacios, definiendo cada subespacio por una lista de vectores, y la función nos devolverá una base de la suma de los subespacios. En IntersectionBasis sucede lo mismo solo que en vez de la suma nos devuelve la intersección de los subespacios. Ej:

```
> with(LinearAlgebra):
  v1 := <1|0|0>:
  v2 := <0|1|0>:
  v3 := <0|0|1>:
  v4 := <0|1|1>:
  v5 := <1|1|1>:
  v6 := <4|2|0>:
  v7 := <3|0|-1>:
  Basis([v1,v2,v2]);
```

$$[[1, 0, 0], [0, 1, 0]]$$

```
> Basis({v4,v6,v7});
```

$$\{[0, 1, 1], [4, 2, 0], [3, 0, -1]\}$$

```
> Basis(v1);
```

$$\{[1, 0, 0]\}$$

```
> SumBasis([ [v1,v2], [v6, <0|1|0>] ]);
```

$$[[1, 0, 0], [0, 1, 0]]$$

```
> SumBasis([ {v1}, [v2,v3], v5 ]);
```

$$\{[1, 0, 0], [0, 1, 0], [0, 0, 1]\}$$

```
> IntersectionBasis([ [v1,v2,v3], {v7,v4,v6}, [v3,v4,v5] ]);
```

$$\{[0, 1, 1], [1, 1, 1], [0, 0, 1]\}$$

```
> IntersectionBasis([ v1, {v3,v7} ]);
```

$$\{[3, 0, 0]\}$$

```
> IntersectionBasis([ [v1,v2], [v3] ]);
```

$$[\ ]$$

- **CharacteristicMatrix:** Esta función permite construir la matriz característica de la matriz A (es decir, lambda*I–A, siendo I la matriz identidad). La sintaxis es *CharacteristicMatrix(**A**, **lambda**)*, donde A es una matriz cuadrada, y lambda es la variable que se usa. Ej:

```
> A:=Matrix([[23,123,7],[22,17,18],[1,2,6]]);
```

$$A := \begin{bmatrix} 23 & 123 & 7 \\ 22 & 17 & 18 \\ 1 & 2 & 6 \end{bmatrix}$$

```
> CharacteristicMatrix(A,tau);
```

$$\begin{bmatrix} -\tau + 23 & 123 & 7 \\ 22 & -\tau + 17 & 18 \\ 1 & 2 & -\tau + 6 \end{bmatrix}$$

- **CharacteristicPolynomial:** Esta función calcula el polinomio característico de la matriz A (es decir, (-1)^n*det(A-lambda*I) , donde I es la matriz identidad y n es la dimensión de A). La sintaxis es CharacteristicPolynomial(**A, lambda**). Ej:

```
> A:=Matrix([[23,123,7],[22,17,18],[1,2,6]]);
```

$$A := \begin{bmatrix} 23 & 123 & 7 \\ 22 & 17 & 18 \\ 1 & 2 & 6 \end{bmatrix}$$

```
> CharacteristicPolynomial(A,lambda);
```

$$\lambda^3 - 46\,\lambda^2 - 2118\,\lambda + 12315$$

- **RowSpace y ColumnSpace:** Estas funciones calculan, respectivamente, una base del subespacio de columnas y de filas de la matriz, que es pasada como argumento. Véase un ejemplo y la respuesta que da Maple:

```
> with(LinearAlgebra):
> A:=Matrix([[23,a,1-c],[2,4,6],[1,2,3]]);
```

$$A := \begin{bmatrix} 23 & a & 1-c \\ 2 & 4 & 6 \\ 1 & 2 & 3 \end{bmatrix}$$

```
> ColumnSpace(A);
```

$$\left[\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \frac{1}{2} \end{bmatrix}\right]$$

Ésta es una base con dos vectores. A partir de estos dos vectores podremos conseguir mediante combinaciones lineales los 3 vectores que forman las columnas de la matriz A. Por tanto, como es un subespacio bidimensional (en la base hay solo dos vectores) deducimos que las 3 columnas de la matriz no son linealmente independientes.

```
> RowSpace(A);
```

$$\left[\left[1, 0, \frac{-2+3\,a+2\,c}{-46+a}\right], \left[0, 1, -\frac{68+c}{-46+a}\right]\right]$$

Estos dos vectores son base de las filas de la matriz. Como no podía ser de otra manera, las 3 filas tampoco son linealmente independientes.

- **Determinant:** Esta función calcula el determinante de una matriz definida de forma numérica o simbólica. La sintaxis es Determinant(**A**), donde A es la matriz. Ej:

Podemos comprobar que la matriz anterior no tenía vectores linealmente independientes, ya que su determinante nos dará 0:

```
> Determinant(A);      #tener cuidado, poner Determinant con D mayúscula.
```

$$0$$

Otros ejemplos:

```
> B:=Matrix([[2,23,1],[2,4,6],[1,7,3]]);
```

$$B := \begin{bmatrix} 2 & 23 & 1 \\ 2 & 4 & 6 \\ 1 & 7 & 3 \end{bmatrix}$$

```
> Determinant(B);
```

-50

```
> C:= Matrix([[a^2-b,a,1-c],[b^3-c-a,1-2/b,a],[c,2,(2-a)/c]]);
```

$$C := \begin{bmatrix} a^2-b & a & 1-c \\ b^3-c-a & 1-\frac{2}{b} & a \\ c & 2 & \frac{2-a}{c} \end{bmatrix}$$

```
> Determinant(C);
```

$$-(2a^3bc-2ab^2c+4a^2+2ba-2c^2-4a^2b+2a^3b+2b^2-b^2a-2a^3+2c^3-2b^4c+2b^4c^2+2b^4a-b^4a^2+3bc^2-3bc^3+bca^2-2bac^2-c^2a^2b-4b)/(bc)$$

- **Eigenvalues:** Esta función calcula los valores propios de una matriz cuadrada, calculando las soluciones del problema **A . x = lambda*x** ; Para el caso generalizado la expresión es **A . x = lambda*C * x.** La sintaxis del comando es *Eigenvalues(**A**, **C**, **imp**, **o**, **outopts**)*, donde A es la matriz del problema (conviene tener en cuenta que cuando la matriz contiene elementos simbólicos y no es puramente numérica, puede desbordar la capacidad de cálculo de nuestra máquina, por esto se recomienda tener cuidado y evaluar bien el problema antes de ejecutarlo cuando se trabaja con elementos simbólicos. C(opcional) es la matriz para el caso generalizado, imp(opcional) es una variable boolean que nos dice si se van a devolver los valores como raíz de una ecuación (RootOf) o como radicales, o(opcional) es el objeto en el que queremos que se devuelvan los resultados, pudiendo ser 'Vector', 'Vector[row]', 'Vector[column]', o 'list' (en vez de o, se pone output='Vector' por ejemplo). Por último outpots(opcional) hace referencia a las opciones de construcción del objeto de salida. Veamos unos ejemplos:

```
> with(LinearAlgebra):
> A := Matrix([[1,1,1],[2,1,1],[0,0,1]]);
```

$$A := \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> v:=Eigenvalues(A, output='Vector');
```

$$v := \begin{bmatrix} 1 \\ 1+\sqrt{2} \\ 1-\sqrt{2} \end{bmatrix}$$

Podemos acceder ahora a los resultados fácilmente.

```
> v[1];v[2];
```

$$1$$

$$1+\sqrt{2}$$

```
> Eigenvalues(A,implicit ,output='Vector');
```

$$\begin{bmatrix} 1 \\ \text{RootOf}(\_Z^2-2\,\_Z-1,\ index=1) \\ \text{RootOf}(\_Z^2-2\,\_Z-1,\ index=2) \end{bmatrix}$$

El efecto de la opción 'implicit'. Index se refiere que el polinomio tiene 2 raíces y por tanto las numera.

- **Eigenvectors:** Esta función calcula los vectores propios de una matriz cuadrada, calculando las soluciones del problema **A . x = lambda*x.** Para el caso generalizado la expresión es **A . x = lambda*C*x.** La sintaxis del comando es *Eigenvalues(**A**, **C**, **imp**, **o**, **outopts**)*, donde A es la matriz del problema (conviene tener en cuenta que cuando la matriz contiene elementos simbólicos y no es puramente numérica, puede desbordar la capacidad de cálculo de nuestra máquina, por esto se recomienda tener cuidado y evaluar bien el problema antes de ejecutarlo cuando se trabaja con elementos simbólicos). C(opcional) es la matriz para el caso generalizado, imp(opcional) es una variable boolean que nos dice si se van a devolver los valores como raíz de una ecuación (RootOf) o como radicales, o(opcional) es el objeto en el que queremos que se devuelvan los resultados, pudiendo ser 'vectors' o 'list'. Por último outpts (opcional) hace referencia a las opciones de construcción del objeto de salida. Ej:

```
> with(LinearAlgebra):
A := Matrix([[-1,-3,-6],[3,5,6],[-3,-3,-4]]);
```

$$A := \begin{bmatrix} -1 & -3 & -6 \\ 3 & 5 & 6 \\ -3 & -3 & -4 \end{bmatrix}$$

```
> (v, e) := Eigenvectors(A);
```

$$v, e := \begin{bmatrix} 2 \\ 2 \\ -4 \end{bmatrix}, \begin{bmatrix} -2 & -1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{bmatrix}$$

Vemos que en esta forma nos da también los autovalores.

```
> A . e[1..-1,2] = v[2] . e[1..-1,2];  #Accedemos a los resultados.
```

$$\begin{bmatrix} -2 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \\ 0 \end{bmatrix}$$

```
> B := Matrix([[1,2,3],[2,4,6],[5,10,15]]);
```

$$B := \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 5 & 10 & 15 \end{bmatrix}$$

```
> Eigenvectors(B, output='list');
```

$$\left[\left[0,\ 2,\ \left\{\begin{bmatrix}-2\\1\\0\end{bmatrix},\ \begin{bmatrix}-3\\0\\1\end{bmatrix}\right\}\right],\left[20,\ 1,\ \left\{\begin{bmatrix}\frac{1}{2}\\1\\\frac{5}{2}\end{bmatrix}\right\}\right]\right]$$

Si le pedimos que la salida sea de output='list', entonces nos da una lista en que cada elemento es a su vez una lista. El primer elemento es el valor propio, el segundo su multiplicidad y el tercero son los vectores propios.

```
> B:=Matrix(3,3,1);
```

$$B := \begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$$

```
> Eigenvectors(B);Eigenvectors(B,output='list');
```

$$\begin{bmatrix}0\\0\\3\end{bmatrix},\begin{bmatrix}-1 & -1 & 1\\1 & 0 & 1\\0 & 1 & 1\end{bmatrix}$$

$$\left[\left[3,\ 1,\ \left\{\begin{bmatrix}1\\1\\1\end{bmatrix}\right\}\right],\left[0,\ 2,\ \left\{\begin{bmatrix}-1\\0\\1\end{bmatrix},\ \begin{bmatrix}-1\\1\\0\end{bmatrix}\right\}\right]\right]$$

Vemos que el autovalor 0 tiene multiplicidad 2 y que la primera columna de la matriz de autovectores corresponde con el primer autovalor, la segunda con el segundo, y así sucesivamente.

- **GaussianElimination:** Esta función realiza la ***triangularización*** de una matriz ***m*** por ***n*** con pivotamiento por filas. El resultado es una matriz triangular superior, Ej:

```
> with(LinearAlgebra):
A := Matrix([[-1,-3,-6],[3,5,6],[-3,-3,-4]]);
```

$$A := \begin{bmatrix}-1 & -3 & -6\\3 & 5 & 6\\-3 & -3 & -4\end{bmatrix}$$

```
> GaussianElimination(A);
```

$$\begin{bmatrix}-1 & -3 & -6\\0 & -4 & -12\\0 & 0 & -4\end{bmatrix}$$

- **HermitianTranspose y Transpose:** Calculan la hermítica y la traspuesta respectivamente. Ambas tienen la forma Transpose(A,ip,outopts). ip especifica si sobreescribimos en la matriz A o no. 'implace=true' hace que el resultado se escriba en A, y si es false o no le pasamos el argumento crea una nueva matriz. La variable outopts se utiliza como en el resto de funciones.

```
> with(LinearAlgebra):
A := <<1,5,w>|<2,6,x>|<3,7,y>|<4,8,z>>;
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ w & x & y & z \end{bmatrix}$$

```
> Transpose(A);
```

$$\begin{bmatrix} 1 & 5 & w \\ 2 & 6 & x \\ 3 & 7 & y \\ 4 & 8 & z \end{bmatrix}$$

```
> V := <a,b,c>;
```

$$V := \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

```
> HermitianTranspose(V, inplace=true): V;
```

$$[\overline{a}, \overline{b}, \overline{c}]$$

```
> HermitianTranspose(x);
```

$$\overline{x}$$

- **LinearSolve:** Esta función nos devuelve el vector x que satisface A.x = B. La sintaxis es LinearSolve(**A**, **B**), donde A y B(opcional) son las matrices del problema. El ejemplo más adelante en *Ejemplos*

- **Pívot:** Pone a cero el resto de los elementos de la columna del elemento especificado. Su forma es: Pivot(**A**, **i**, **j**). i,j es la posición del elemento a pivotar. Veamoslo más claro con un ejemplo:

```
> A := <<1,5,9,3>|<2,6,0,4>|<3,7,1,5>|<4,8,2,6>>;
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \end{bmatrix}$$

```
> Pivot(A, 1, 1);
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & -18 & -26 & -34 \\ 0 & -2 & -4 & -6 \end{bmatrix}$$

Hay unas funciones que no están en el paquete LinearAlgebra, sino que están dentro del subpaquete LinearAlgebra del paquete Student (las funciones hasta ahora explicadas también aparecen en esta última). Estas funciones son:

- **AddRow(A, i, j, s):** Sustituye la fila i de A por la (fila i)+ s*(fila j).
- **MultiplyRow(A, i, s):** Sustituye la fila i por (fila i)*s.
- **SwapRow(A, i, j):** Intercambia las filas i y j.

```
> with(Student[LinearAlgebra]):
> A := <<1,3,7>|<4,5,6>>;
```

$$A := \begin{bmatrix} 1 & 4 \\ 3 & 5 \\ 7 & 6 \end{bmatrix}$$

```
> B := AddRow(A, 2, 1, -3);
```

$$B := \begin{bmatrix} 1 & 4 \\ 0 & -7 \\ 7 & 6 \end{bmatrix}$$

```
> SwapRow(B, 1, 3);
```

$$\begin{bmatrix} 7 & 6 \\ 0 & -7 \\ 1 & 4 \end{bmatrix}$$

```
> MultiplyRow(A, 2, x);
```

$$\begin{bmatrix} 1 & 4 \\ 3x & 5x \\ 7 & 6 \end{bmatrix}$$

EJEMPLOS

Ejemplo 1

```
> with(LinearAlgebra):
M := <<1,1,1,4>|<1,1,-2,1>|<3,1,1,8>|<-1,1,-1,-1>|<0,1,1,0>>;
LinearSolve(M);
```

$$M := \begin{bmatrix} 1 & 1 & 3 & -1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -2 & 1 & -1 & 1 \\ 4 & 1 & 8 & -1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \frac{25}{6} \\ \frac{4}{3} \\ \frac{-5}{2} \\ -2 \end{bmatrix}$$

* si Ax=B, B es la última columna de M. Si no introducimos B, se supondrá que es una columna de ceros.

Si el problema que queremos resolver A=M, siendo Ax=0, entonces lo que habrá que hacer será introducir una columna de ceros:

```
> LinearSolve(M,free='s');  #especificamos cuál queremos que sea el
```

$$M := \begin{bmatrix} 1 & 1 & 3 & -1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & -2 & 1 & -1 & 1 & 0 \\ 4 & 1 & 8 & -1 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} -\frac{25}{12} s_4 \\ -\frac{2}{3} s_4 \\ \frac{5}{4} s_4 \\ s_4 \\ \frac{1}{2} s_4 \end{bmatrix}$$

```
parámetro mediante free=.
```

Ejemplo 2

```
> A := <<1,0,0>|<2,1,0>|<1,0,0>|<-1,-1,-3>>: b := <2,-1,-9>:
LinearSolve(A, b, method='subs', free='s');
```

$$\begin{bmatrix} 1 - s_1 \\ 2 \\ s_1 \\ 3 \end{bmatrix}$$

En este caso introducimos el vector de coeficientes independientes como variable

Mediante 'GenerateMatrix' podemos crear la matriz a partir de las ecuaciones**:**

```
> with(LinearAlgebra):
sys := [ 3*x[1]+2*x[2]+3*x[3]-2*x[4] = 1,
         x[1]+  x[2]+  x[3]         = 3,
         x[1]+2*x[2]+  x[3]-  x[4] = 2 ]:
var := [ x[1], x[2], x[3], x[4] ]:
(A, b) := GenerateMatrix( sys, var );
```

$$A, b := \begin{bmatrix} 3 & 2 & 3 & -2 \\ 1 & 1 & 1 & 0 \\ 1 & 2 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$$

También se puede realizar el proceso inverso mediante 'GenerateEquations' y las variables serán iguales, solo que en vez de el sistema habrá que poner la matriz ampliada (con los coeficientes independientes).

Ejemplo 3

Resolver el siguiente sistema de ecuaciones:

x-2y+3z-4t=4
y-z+t=-3
x+3y-3t=1
-7y+3z-t=1

```
> with(LinearAlgebra):
> sys:=[x-2*y+3*z-4*t=4,y-z+t=-3,x+3*y-3*t=1,-7*y+3*z-t=1];
```

$$sys := [x-2y+3z-4t=4,\ y-z+t=-3,\ x+3y-3t=1,\ -7y+3z-t=1]$$

```
> (A,b):= GenerateMatrix(sys,[x,y,z,t]);
```

$$A, b := \begin{bmatrix} 1 & -2 & 3 & -4 \\ 0 & 1 & -1 & 1 \\ 1 & 3 & 0 & -3 \\ 0 & -7 & 3 & -1 \end{bmatrix}, \begin{bmatrix} 4 \\ -3 \\ 1 \\ 1 \end{bmatrix}$$

```
> C:=Matrix(4,5,0);
```

$$C := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> C[1..4,1..4]:=A: C[1..4,5]:=b: C;
```

$$\begin{bmatrix} 1 & -2 & 3 & -4 & 4 \\ 0 & 1 & -1 & 1 & -3 \\ 1 & 3 & 0 & -3 & 1 \\ 0 & -7 & 3 & -1 & 1 \end{bmatrix}$$

```
> g:=LinearSolve(C);
```

$$g := \begin{bmatrix} -8 \\ 1 \\ 2 \\ -2 \end{bmatrix}$$

```
> H:=Matrix(4,5,0):H[1..4,1..4]:=IdentityMatrix(4): H[1..4,5]:=G:H;
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & -8 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix}$$

```
> GenerateEquations(H,[x,y,z,t]);
```

$$[x=-8,\ y=1,\ z=2,\ t=-2]$$

Ejemplo General

Vamos a calcular la inversa de una matriz utilizando el método de Gauss-Jordan.

```
> with(LinearAlgebra):
```

```
> A:=Matrix(<<1,2,1>|<2,2,3>|<3,2,1>|<1,0,0>|<0,1,0>|<0,0,1>>);
B:=A[1..3,1..3]:
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 1 & 0 \\ 1 & 3 & 1 & 0 & 0 & 1 \end{bmatrix}$$

```
> A:=GaussianElimination(A);
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & -2 & -4 & -2 & 1 & 0 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

Se puede resolver de dos modos distintos:

Modo 1:

```
> b:=A[2,1..6]-A[3,1..6]:
A[2,1..6]:=b:
A;
```

$$\begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & -2 & 0 & 0 & \frac{1}{2} & -1 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> c:=4*A[1,1..6]+3*A[3,1..6]:
A[1,1..6]:=c:
A;
```

$$\begin{bmatrix} 4 & 8 & 0 & -2 & \frac{3}{2} & 3 \\ 0 & -2 & 0 & 0 & \frac{1}{2} & -1 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> d:=A[1,1..6]+4*A[2,1..6]:
A[1,1..6]:=d:
A;
```

$$\begin{bmatrix} 4 & 0 & 0 & -2 & \frac{7}{2} & -1 \\ 0 & -2 & 0 & 0 & \frac{1}{2} & -1 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> A[1,1..6]:=A[1,1..6]/4:
A[2,1..6]:=A[2,1..6]/(-2):
A[3,1..6]:=A[3,1..6]/(-4):
A;
```

$$\begin{bmatrix} 1 & 0 & 0 & \frac{-1}{2} & \frac{7}{8} & \frac{-1}{4} \\ 0 & 1 & 0 & 0 & \frac{-1}{4} & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{2} & \frac{-1}{8} & \frac{-1}{4} \end{bmatrix}$$

La inversa:

```
> InvA:=A[1..3,4..6];
```

$$InvA := \begin{bmatrix} \frac{-1}{2} & \frac{7}{8} & \frac{-1}{4} \\ 0 & \frac{-1}{4} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{8} & \frac{-1}{4} \end{bmatrix}$$

Modo 2:

```
> with(Student[LinearAlgebra]):
> A:=Matrix(<<1,2,1>|<2,2,3>|<3,2,1>|<1,0,0>|<0,1,0>|<0,0,1>>);
B:=A[1..3,1..3]:
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 1 & 0 \\ 1 & 3 & 1 & 0 & 0 & 1 \end{bmatrix}$$

```
> A:=GaussianElimination(A);
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & -2 & -4 & -2 & 1 & 0 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> A:=AddRow(A,2,3,-1);
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & -2 & 0 & 0 & \frac{1}{2} & -1 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> A:=AddRow(A,1,3,3/4);
```

$$A := \begin{bmatrix} 1 & 2 & 0 & \frac{-1}{2} & \frac{3}{8} & \frac{3}{4} \\ 0 & -2 & 0 & 0 & \frac{1}{2} & -1 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> A:=AddRow(A,1,2,1);
```

$$A := \begin{bmatrix} 1 & 0 & 0 & \frac{-1}{2} & \frac{7}{8} & \frac{-1}{4} \\ 0 & -2 & 0 & 0 & \frac{1}{2} & -1 \\ 0 & 0 & -4 & -2 & \frac{1}{2} & 1 \end{bmatrix}$$

```
> A:=MultiplyRow(A,2,-1/2): A:=MultiplyRow(A,3,-1/4);
```

$$A := \begin{bmatrix} 1 & 0 & 0 & \frac{-1}{2} & \frac{7}{8} & \frac{-1}{4} \\ 0 & 1 & 0 & 0 & \frac{-1}{4} & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{2} & \frac{-1}{8} & \frac{-1}{4} \end{bmatrix}$$

```
> InvA:=A[1..3,4..6];
```

$$InvA := \begin{bmatrix} \frac{-1}{2} & \frac{7}{8} & \frac{-1}{4} \\ 0 & \frac{-1}{4} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{8} & \frac{-1}{4} \end{bmatrix}$$

Comprobémoslo:

```
> InvB:=evalm(B^(-1));
```

$$InvB := \begin{bmatrix} \frac{-1}{2} & \frac{7}{8} & \frac{-1}{4} \\ 0 & \frac{-1}{4} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{8} & \frac{-1}{4} \end{bmatrix}$$

Se observa que son iguales

5.5. ECUACIONES DIFERENCIALES. *(DEtools)*

El paquete DEtools contiene funciones que ayudan a trabajar con las ecuaciones diferenciales, aunque contiene multitud de secciones, aquí nos centraremos en algunas de las funciones. Si se quiere consultar qué funciones existen en DEtools, podemos teclear *?DEtools* en la hoja de trabajo. Las funciones son:

```
> with(DEtools);
```

[*DEnormal*, *DEplot*, *DEplot3d*, *DEplot_polygon*, *DFactor*, *DFactorLCLM*, *DFactorsols*, *Dchangevar*, *FunctionDecomposition*, *GCRD*, *LCLM*, *MeijerGsols*, *PDEchangecoords*, *RiemannPsols*, *Xchange*, *Xcommutator*, *Xgauge*, *Zeilberger*, *abelsol*, *adjoint*, *autonomous*, *bernoullisol*, *buildsol*, *buildsym*, *canoni*, *caseplot*, *casesplit*, *checkrank*, *chinisol*, *clairautsol*, *constcoeffsols*, *convertAlg*, *convertsys*,

dalembertsol, dcoeffs, de2diffop, dfieldplot, diff_table, diffop2de, dperiodic_sols, dpolyform, dsubs, eigenring, endomorphism_charpoly, equinv, eta_k, eulersols, exactsol, expsols, exterior_power, firint, firtest, formal_sol, gen_exp, generate_ic, genhomosol, gensys, hamilton_eqs, hypergeomsols, hyperode, indicialeq, infgen, initialdata, integrate_sols, intfactor, invariants, kovacicsols, leftdivision, liesol, line_int, linearsol, matrixDE, matrix_riccati, maxdimsystems, moser_reduce, muchange, mult, mutest, newton_polygon, normalG2, ode_int_y, ode_y1, odeadvisor, odepde, parametricsol, phaseportrait, poincare, polysols, power_equivalent, ratsols, redode, reduceOrder, reduce_order, regular_parts, regularsp, remove_RootOf, riccati_system, riccatisol, rifread, rifsimp, rightdivision, rtaylor, separablesol, singularities, solve_group, super_reduce, symgen, symmetric_power, symmetric_product, symtest, transinv, translate, untranslate, varparam, zoom]

A continuación se describen algunas funciones para ecuaciones diferenciales ordinarias (en adelante EDO).

- **intfactor:** Esta función busca un factor integrante para una EDO, de manera que si el factor integrante es por ejemplo mu, mu*EDO es una ecuación diferencial exacta. Por defecto intfactor busca un número de factores integrantes igual al orden de la EDO dada, y devuelve una respuesta tan pronto como todos los factores integrantes hayan sido hallados o cuando todos los esquemas hayan sido probados. La sintaxis es ***intfactor***(**EDO**, **y(x)**), donde EDO es la ecuación diferencial ordinaria que se desea resolver; y(x) (opcional) es la variable dependiente, necesaria cuando la ecuación diferencial contiene más de una función diferenciada.. Veamos unos ejemplos de EDOs de primer orden:

```
> with(DEtools):
> EDO := diff(y(x),x) = (y(x)^2-x^2)/(2*x*y(x));
```

$$EDO := \frac{d}{dx}y(x) = \frac{1}{2}\frac{y(x)^2 - x^2}{x\,y(x)}$$

```
> mu := intfactor(EDO);
```

$$\mu := \frac{y(x)}{x}, \frac{y(x)}{y(x)^2 + x^2}$$

Puesto que nos devuelve dos factores integrantes comprobaremos que utilizando ambos coinciden las soluciones.

```
> mu[1]*EDO;
```

$$\frac{y\,y'}{x} = \frac{y^2 - x^2}{2\,x^2}$$

```
> dsolve(%,y(x));
```

$$y = \sqrt{-x^2 + x\_C1}, y = -\sqrt{-x^2 + x\_C1}$$

```
> mu[2]*EDO;
```

$$\frac{y\,y'}{y^2 + x^2} = \frac{y^2 - x^2}{2\,(y^2 + x^2)\,x}$$

```
> dsolve(%,y(x));
```

$$y = \sqrt{-x^2 + x\_C1}, y = -\sqrt{-x^2 + x\_C1}$$

- **reduceOrder**: Esta función aplica el método de reducción de orden a una EDO, el método consiste en que conociendo una solución no trivial de la ecuación podemos convertir la EDO en una de orden inferior. La sintaxis es ***reduceOrder***(**EDO**, **dvar**, **partsol**), donde **dvar** es la variable dependiente de la ecuación y **partsol** es una solución parcial de la ecuación (o una lista de soluciones). Ej:

```
> with(DEtools);
> de := Diff(y(x),x$3) - 6*Diff(y(x),x$2) + 11*Diff(y(x),x) -
6*y(x);
```

$$de := \left(\frac{\partial^3}{\partial x^3} y\right) - 6\left(\frac{\partial^2}{\partial x^2} y\right) + 11\left(\frac{\partial}{\partial x} y\right) - 6\,y$$

```
> sol := exp(x);
```

$$sol := \mathbf{e}^{x}$$

```
> reduceOrder( de, y(x), sol);
```

Nos devuelve una ecuación de orden menor.

$$y'' - 3\,y' + 2\,y$$

```
> reduceOrder( de, y(x), sol, basis);
```

En este caso nos resuelve la ecuación.

$$\left[\mathbf{e}^{x}, \mathbf{e}^{(2\,x)}, \frac{1}{2}\,\mathbf{e}^{(3\,x)}\right]$$

- **polysols:** Permite obtener soluciones en el caso de una E.D.O lineal con coeficientes que sean funciones racionales. Si en los argumentos le ponemos output=solution , entonces nos dará la solución del problema. Si no nos dará las bases del problema (en una lista) y una solución particular.

```
> with(DEtools, polysols):
> odeH := diff(z(t),t$2) - 3/t*diff(z(t),t) + 3/t^2*z(t) = t^2;
```

$$odeH := \left(\frac{d^2}{dt^2} z(t)\right) - \frac{3\left(\frac{d}{dt} z(t)\right)}{t} + \frac{3\,z(t)}{t^2} = t^2$$

```
> polysols(odeH);
```

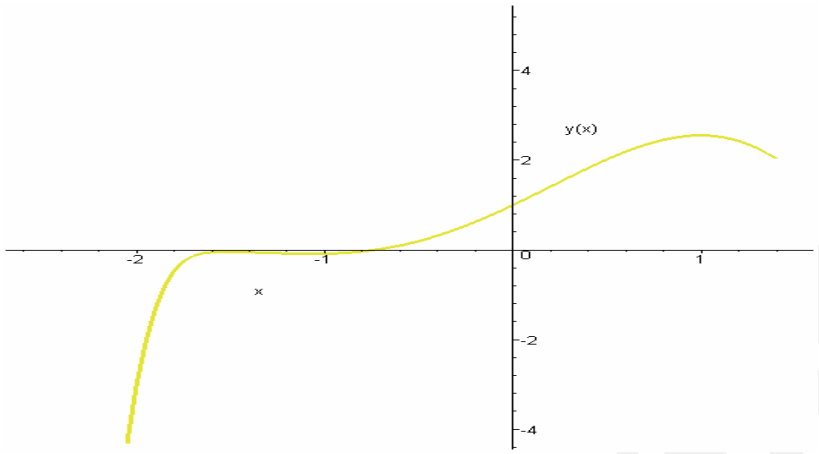
$$\left[[t, t^3], \frac{t^4}{3}\right]$$

```
> polysols(odeH,output=solution);
```

$$\left[z(t) = \_C1\,t + \_C2\,t^3 + \frac{1}{3}\,t^4\right]$$

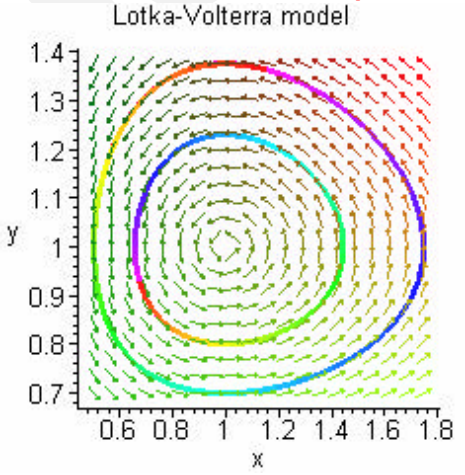
- **DEplot:** Esta función nos permite representar las soluciones de sistemas de ecuaciones diferenciales. Tiene multitud de maneras de ser llamada pasando distintos argumentos (para ver todas ?DEplot). Lo más fácil será ver algunos ejemplos:

```
> with(DEtools):
DEplot(cos(x)*diff(y(x),x$3)-diff(y(x),x$2)+Pi*diff(y(x),x)=y(x)-
x,y(x),
x=-2.5..1.4,[[y(0)=1,D(y)(0)=2,(D@@2)(y)(0)=1]],y=-
4..5,stepsize=.05);
```



Vemos que el primer argumento es la EDO. Luego hay que especificar cúal es la variable dependiente. Después el rango de la variable dependiente, seguido de las condiciones iniciales, el rango de la variable dependiente y finalmente las opciones que queramos (las opciones también están en el Help, ?DEplot).

```
> DEplot([diff(x(t),t)=x(t)*(1-y(t)),diff(y(t),t)=.3*y(t)*(x(t)-
1)],
[x(t),y(t)],t=-
7..7,[[x(0)=1.2,y(0)=1.2],[x(0)=1,y(0)=.7]],stepsize=.2,
title=`Lotka-Volterra model`,color=[.3*y(t)*(x(t)-1),x(t)*(1-
y(t)),.1],
linecolor=t/2,arrows=MEDIUM,method=rkf45);
```

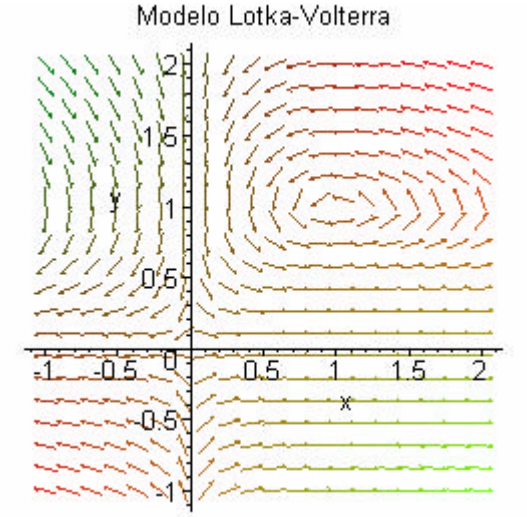


También se pueden representar sistemas de EDOs. En este caso, poniendo la opción arrows=MEDIUM, generamos el campo de direcciones (esto también se consigue con dfieldplot y con las condiciones iniciales generamos las 2 trayectorias. Vemos que los colores son funciones de las variables.

- **dfieldplot:** Nos muestra el campo de direcciones de un sistema de ecuaciones diferenciales. La sintaxis es dfieldplot(**deqns**, **vars**, **trange**, **yrange**, **xrange**, **options**), siendo *deqns* es la lista de EDOs, vars son las variables dependientes,

trange el rango de la variable independiente y xrange e yrange las de las dependientes. Hay que tener en cuenta que para sistemas no autónomos no se producirá un campo de direcciones. Ej:

```
> with(DEtools):
dfieldplot([diff(x(t),t)=x(t)*(1-y(t)), diff(y(t),t)=.3*y(t)*(x(t)-
1)],
[x(t),y(t)],t=-2..2, x=-1..2, y=-1..2, arrows=small,
title=`Modelo Lotka-Volterra`, color=[.3*y(t)*(x(t)-1),x(t)*(1-
y(t)),.1]);
```



Si cambiamos la primera EDO multiplicando por t el miembro de la derecha dejará de ser un sistema autónomo y ya no podremos dibujar el campo de direcciones.

```
> dfieldplot([diff(x(t),t)=t*x(t)*(1-y(t)),
diff(y(t),t)=.3*y(t)*(x(t)-1)],
[x(t),y(t)],t=-2..2, x=-1..2, y=-1..2, arrows=small,
title=`Modelo Lotka-Volterra`, color=[.3*y(t)*(x(t)-1),x(t)*(1-
y(t)),.1]);
Error, (in dfieldplot) Cannot produce plot non-autonomous DE(s)
require initial conditions.
```

- **pdsolve:** Resuelve las ecuaciones diferenciales en derivadas parciales, o sistemas de estas utilizando la separación de variables. En las soluciones aparecerán funciones arbitrarias.

```
> PDE := x*diff(f(x,y),y)-y*diff(f(x,y),x) = 0;
```

$$PDE := x\left(\frac{\partial}{\partial y}\mathrm{f}(x,y)\right)-y\left(\frac{\partial}{\partial x}\mathrm{f}(x,y)\right)=0$$

```
> pdsolve(PDE);
```

$$\mathrm{f}(x,y)=\_\mathrm{F1}(x^2+y^2)$$

```
> PDE1:=x^2*diff(f(x,y,z),x$2)+y*z*diff(f(x,y,z),z)=x+y;
```

$$PDE1 := x^2\left(\frac{\partial^2}{\partial x^2}\mathrm{f}(x,y,z)\right)+y\,z\left(\frac{\partial}{\partial z}\mathrm{f}(x,y,z)\right)=x+y$$

```
> pdsolve(PDE1);
```

$$(f(x,y,z) = \_F00(y)\,\_F3(z) + y\,\_c_3\,\_F00(y)\ln(x) - y\ln(x) + x\ln(x) - x + x\,\_F5(y) + \_F6(y))$$

$$\&\text{where}\left[\left\{\frac{d}{dz}\_F3(z) = \frac{\_c_3}{z}\right\},\ (\_F00(y), \_F5(y), \_F6(y),\ \textit{are arbitrary functions.})\right]$$

- **PDEplot:** Sirve para dibujar la solución de una ecuación diferencial en derivadas parciales de primer grado para unas condiciones iniciales dadas. Si hay más de 2 variables independientes, la representación se hará mediante animación, ya que no se puede dibujar en 3 dimensiones. La forma de la función es:

  PDEplot(**PDE**, **inits**, **srange**, **options**).

  - PDE es la ecuación diferencial en derivadas parciales de n variables
  - inits son las condiciones iniciales. Habrá que dar una curva en paramétricas. Es decir, habrá que dar n+1 expresiones (las n variables más la función que queremos representar) en función de n-1 variables. Si n=2, la función será una superficie y tendremos que dar como condición inicial una curva que dependerá de un parámetro.
  - srange será el rango de las variables que utilizaremos para describir la curva.
  - options (opcional) son las características de la gráfica. Para más información mirar en PDEplot options. Las opciones se pueden ir cambiando desde el menú de la gráfica.

Para utilizar esta función hace falta abrir el paquete:

```
> with(PDEtools);
```

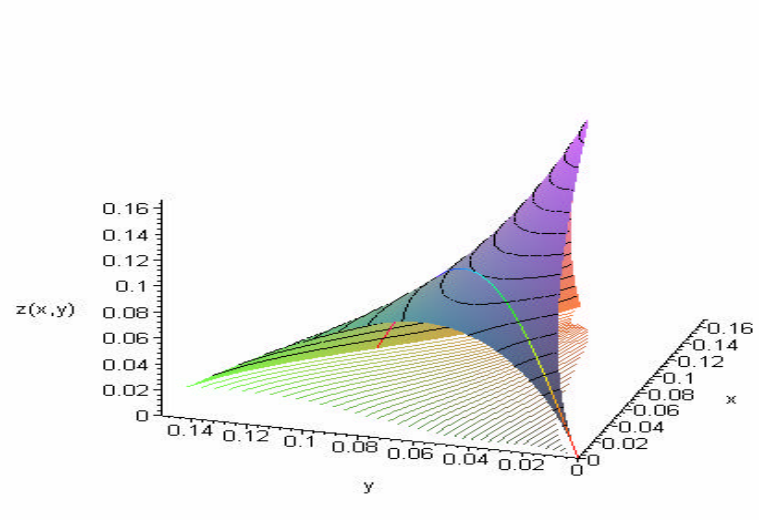
[*PDEplot*, *TWSolutions*, *build*, *casesplit*, *charstrip*, *dchange*, *dcoeffs*, *declare*, *difforder*, *dpolyform*, *dsubs*, *mapde*, *separability*, *splitstrip*, *splitsys*, *undeclare*]

Ejemplos:

```
> with(PDEtools);
> pde := (y^2+z(x,y)^2+x^2)*diff(z(x,y),x) - 2*x*y*diff(z(x,y),y)
- 2*z(x,y)*x = 0;
```

$$pde := (y^2 + z(x,y)^2 + x^2)\left(\frac{\partial}{\partial x}z(x,y)\right) - 2\,x\,y\left(\frac{\partial}{\partial y}z(x,y)\right) - 2\,z(x,y)\,x = 0$$

```
> PDEplot(pde, [t,t,sin(Pi*t/0.1)/10], t=0..0.1, numchar=40,
orientation=[-163,56], basechar=true, numsteps=[20,20],
stepsize=.15,
initcolour=cos(t)*t, animate=false, style=PATCHCONTOUR);
```



*NOTA: En las condiciones, la primera se refiere a x (x=t), la segunda a y, y la última a f(x,y).

Otro ejemplo:

```
> pde2 := sin(diff(u(x,y,z),y)) = u(x,y,z);
```

$$pde2 := \sin\left(\frac{\partial}{\partial y}\,u(x, y, z)\right) = u(x, y, z)$$

```
> ics := [cos(t)*sin(s), cos(s)*cos(t), cos(t), sin(t)] , [t=0..Pi,
s=0..Pi];
```

$$ics := [\cos(t)\sin(s), \cos(s)\cos(t), \cos(t), \sin(t)], [t = 0 .. \pi, s = 0 .. \pi]$$

```
> PDEplot(pde2, ics, animate=only, numsteps=[-5,6], stepsize=.1,
axes=NONE,
style=PATCHNOGRID, numchar=[16,16], orientation=[148,66],
lightmodel='light2');
```

Se puede observar que al ser n>2, nos sale una animación. Clicando sobre la gráfica empezará a moverse, aunque es difícil entender lo que representa.

- **diff_table:** Sirve para simplificar la notación de las ecuaciones diferenciales en derivadas parciales. Es una función nueva en Maple 9.5. Veremos como funciona mediante un ejemplo:

```
> U := DEtools[diff_table]( u(x,y,t) ):
  V := DEtools[diff_table]( v(x,y,t) ):
> e1 := U[y,t] + V[x,x] + U[x]*U[y] + U[]*U[x,y];
```

$$e1 := \left(\frac{\partial^2}{\partial y\,\partial t}\,u(x,y,t)\right)+\left(\frac{\partial^2}{\partial x^2}\,v(x,y,t)\right)+\left(\frac{\partial}{\partial x}\,u(x,y,t)\right)\left(\frac{\partial}{\partial y}\,u(x,y,t)\right) + u(x,y,t)\left(\frac{\partial^2}{\partial y\,\partial x}\,u(x,y,t)\right)$$

5.6. TRANSFORMADAS INTEGRALES. *(inttrans)*

La librería *inttrans* dispone de una colección de funciones destinadas al cálculo de transformadas integrales.

Para utilizar una función de *inttrans*, se puede definir esa única función mediante el comando ***with(inttrans, función)*** o definir a la vez todas las funciones de la librería con ***with(inttrans)***. Alternativamente, se puede llamar a la función directamente utilizando la notación ***inttrans[función]***. Esta notación es necesaria siempre que haya un conflicto entre el nombre de la función de la librería y el de otra función definida en la misma sesión.

Las funciones disponibles son:

```
> with(inttrans);
```

[*addtable*, *fourier*, *fouriercos*, *fouriersin*, *hankel*, *hilbert*, *invfourier*, *invhilbert*, *invlaplace*, *invmellin*, *laplace*, *mellin*, *savetable*]

Nos centraremos en las transformadas de Laplace y de Fourier y en la posibilidad de agregar transformadas de funciones a la tabla de transformadas del programa. Si el lector necesita utilizar alguna otra función, recordamos que basta con introducir ***?función*** para obtener ayuda sobre la misma.

5.6.1. Transformada de Laplace

5.6.1.1 Transformada directa de Laplace

Esta operación se realiza mediante el comando ***laplace*** utilizado con la siguiente sintaxis: ***laplace(****expr*,*t*,*s****)***. Así calculamos la transformada de Laplace de *expr* con respecto a *t* utilizando la siguiente definición (*t* será la variable de la función original y *s* la de la transformada):

$$F(s) = \int_0^{\infty} f(t)\,\mathbf{e}^{(-s\,t)}\,dt$$

Pueden ser transformados muchos tipos de expresiones, incluyendo aquellos que contengan funciones exponenciales, trigonométricas, funciones de Bessel y muchas otras.

```
> expr:=t^2+sin(t);
```

$$expr := t^2 + \sin(t)$$

```
> transf:=laplace(expr,t,s); #transformada de laplace de la función
```

$$transf := \frac{2}{s^3} + \frac{1}{s^2+1}$$

```
> laplace(t^5/3+exp(9*t)+cosh(7*t),t,s);
```

$$\frac{40}{s^6} + \frac{1}{s-9} + \frac{s}{s^2-49}$$

El comando ***laplace*** reconoce las derivadas y las integrales. Cuando transformemos expresiones como ***diff(y(t),t,s)*** se introducirán los valores correspondientes a *y(0)*, *D(y)(0)*, etc. Recordamos que *D(y)(0)* representa el valor de la primera derivada en el punto 0, *D(D(y))(0)*, el valor de la segunda derivada en el punto 0 y análogamente para las derivadas superiores. Veamos un ejemplo:

```
>  Laplace(diff(f(t),t$2),t,s)=laplace(diff(f(t),t$2),t,s);
#transformada de la derivada (fijarse en que en una Laplace con
mayustcula y en la otra con minuscula)
```

$$\text{Laplace}\left(\frac{d^2}{dt^2}\text{f}(t), t, s\right) = s^2\,\text{laplace}(\text{f}(t), t, s) - \text{D}(f)(0) - s\,\text{f}(0)$$

```
> f(0):=0; D(f)(0):=0; #si imponemos estas condiciones iniciales
```

$$\text{f}(0) := 0$$

$$\text{D}(f)(0) := 0$$

```
> laplace(diff(f(t),t$2),t,s);
```

$$s^2\,\text{laplace}(\text{f}(t), t, s) - \text{D}(f)(0) - s\,\text{f}(0)$$

```
>  eval(%);
```

$$s^2\,\text{laplace}(\text{f}(t), t, s)$$

```
> expr:=Int((1-exp(-x))/x,x=0..t);
```

$$expr := \int_0^t \frac{1 - \mathbf{e}^{(-x)}}{x}\,dx$$

```
> laplace(expr,t,s);
```

$$-\frac{\ln(s) - \ln(1+s)}{s}$$

Conviene recordar también que tanto la función ***laplace*** como la función que veremos en el apartado siguiente, ***invlaplace***, reconocen la función Delta de Dirac (representada por ***Dirac(t)***), y la función escalón de Heaviside (representada por ***Heaviside(t)***).

```
> laplace(Heaviside(t-a),t,s);
```

$$\text{laplace}(\text{Heaviside}(t-a), t, s)$$

```
> assume(a>0): laplace(Heaviside(t-a),t,s);  #hay que suponer que
a>0, ya que si no la solución no sirve
```

$$\frac{\mathbf{e}^{(-s\,a\sim)}}{s}$$

Como se verá más adelante, los usuarios podrán mediante ***addtable*** añadir sus propias funciones a la tabla interna que dispone el programa para calcularlas.

5.6.1.2 Transformada inversa de Laplace

El comando ***invlaplace*** nos permite calcular la transformada inversa de Laplace de una expresión. Su sintaxis es ***invlaplace(****expr,s,t****)*** y calculará la transformada inversa de Laplace de *expr* con respecto a *s*. Su funcionamiento es completamente análogo a la de la transformada directa. Veamos algunos ejemplos:

```
> invlaplace(s*exp(-4*s)/(s^2+6*s+10),s,t);
```

$$\text{Heaviside}(t-4)\,\mathbf{e}^{(-3\,t+12)}\,(\cos(t-4)-3\sin(t-4))$$

Planteemos ahora un problema típico de ecuaciones diferenciales resuelto mediante la transformada de Laplace:

```
> restart;

> with(inttrans):

> addtable(laplace,f(t),F(s),t,s); #mediante addtable (que ya se
verá más adelante) indicamos que la transformada de f(t) es F(s)
> laplace(f(t),t,s);
```

$$\text{F}(s)$$

```
> eqn:=diff(f(t),t$2)+f(t)=Heaviside(t-3); #ecuación
```

$$eqn := \left(\frac{d^2}{dt^2}\,\text{f}(t)\right) + \text{f}(t) = \text{Heaviside}(t-3)$$

```
> f(0):=0; D(f)(0):=1; #condiciones iniciales
```

$$\text{f}(0) := 0$$

$$\text{D}(f)(0) := 1$$

```
> laplace(eqn,t,s); #transformamos la ecuación
```

$$s^2\,\text{F}(s) - 1 + \text{F}(s) = \frac{\mathbf{e}^{(-3\,s)}}{s}$$

```
> sol:=solve(%,F(s)); #resolvemos
```

$$sol := \frac{s + \mathbf{e}^{(-3\,s)}}{s\,(s^2+1)}$$

```
> f(t):=invlaplace(sol,s,t); #aplicamos la transformada inversa
```

$$\text{f}(t) := \sin(t) + 2\,\text{Heaviside}(t-3)\sin\left(\frac{t}{2}-\frac{3}{2}\right)^2$$

5.6.2. Transformada de Fourier

5.6.2.1 Transformada directa de Fourier

El comando ***fourier*** nos permite calcular la transformada de Fourier de una expresión. Su sintaxis es ***fourier(****expr, t,w****)***. Así logramos calcular la transformada de Fourier de *expr* respecto a *t* (es decir, *t* es la variable de la función original y *w*, la de la transformada), mediante la siguiente definición:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, \mathbf{e}^{(-I\,\omega\,t)}\, dt$$

Expresiones compuestas por exponenciales complejas, polinomios, funciones trigonométricas y una gran variedad de funciones pueden ser transformadas.

Al igual que los comandos anteriores, la función ***fourier*** reconoce tanto derivadas como integrales, así como la función Delta de Dirac y la función escalón de Heaviside. Veamos algunos ejemplos:

```
> with (inttrans):
```

```
> assume(a>0):
```

```
> fourier(cos(a*t),t,w);
```

$$\pi\,(\mathrm{Dirac}(w + a\sim) + \mathrm{Dirac}(-w + a\sim))$$

```
> fourier(3/(a^2+t^2),t,w);
```

$$\frac{3\,\pi\,(\mathbf{e}^{(a\sim\, w)}\,\mathrm{Heaviside}(-w) + \mathbf{e}^{(-a\sim\, w)}\,\mathrm{Heaviside}(w))}{a\sim}$$

```
> fourier(diff(f(t),t$4),t,w); #transformada de la derivada cuarta
```

$$w^4\,\mathrm{fourier}(f(t), t, w)$$

```
> F:=int(g(tau)*h(t-tau),tau=-infinity..infinity);
```

$$F := \int_{-\infty}^{\infty} g(\tau)\, h(t-\tau)\, d\tau$$

```
> fourier(F,t,w); #teorema de la convolución
```

$$\mathrm{fourier}(g(t), t, w)\,\mathrm{fourier}(h(t), t, w)$$

Al igual que sucedía con la transformada de Laplace, podemos incluir en la tabla interna de transformadas nuestras propias funciones mediante el comando ***addtable*** que se verá en la sección posterior.

Maple dispone también de las funciones ***fouriersin*** y ***fouriercos*** que calculan las transformadas de Fourier utilizando las dos definiciones siguientes respectivamente:

$$F(s) := \frac{\sqrt{2}}{\sqrt{\pi}} \int_0^{\infty} f(t)\, \sin(s\,t)\, dt$$

$$F(s) := \frac{\sqrt{2}}{\sqrt{\pi}} \int_0^{\infty} f(t)\, \cos(s\,t)\, dt$$

Ambas funciones nos devuelven una F(s) únicamente definida para valores positivos del eje real. Su sintaxis y funcionamiento es por lo demás análogo al del comando ***fourier***. Veamos algún ejemplo:

```
> fouriercos(1/(t^2+1),t,s);
```

$$\frac{1}{2}\sqrt{2}\,\sqrt{\pi}\,\mathbf{e}^{(-s)}$$

```
> fouriersin(t^3/(t^2+1),t,s);
```

$$-\frac{1}{2}\sqrt{2}\,\sqrt{\pi}\,\mathrm{Dirac}(1,s)-\frac{1}{2}\sqrt{2}\,\sqrt{\pi}\,\mathbf{e}^{(-s)}$$

```
> fouriersin(%,s,t);
```

$$t-\frac{t}{t^2+1}$$

```
> simplify(%); #son transformadas autoinvertibles
```

$$\frac{t^3}{t^2+1}$$

5.6.2.2 Transformada inversa de Fourier

La función ***invfourier*** aplica la transformada inversa de Fourier a una expresión. Su sintaxis es ***invfourier(****expr*,*w*,*t****)*** calculando así la transformada inversa de Fourier de *expr* respecto a *t* siguiendo la siguiente definición:

$$\mathrm{f}(t)=\frac{1}{2}\left(\frac{1}{\pi}\int_{-\infty}^{\infty}\mathrm{F}(\omega)\,\mathbf{e}^{(\omega\,t\,I)}\,d\omega\right)$$

Su funcionamiento es completamente análogo al de la transformada directa. Veamos algunos ejemplos:

```
> fourier(t/(t^2+1),t,w);
```

$$\pi\,(\mathbf{e}^{w}\,\mathrm{Heaviside}(-w)-\mathbf{e}^{(-w)}\,\mathrm{Heaviside}(w))\,I$$

```
> invfourier(%,w,t);
```

$$\frac{t}{t^2+1}$$

```
> eqn:=diff(y(t),t$2)-y(t)=cos(t)*sin(t); #ecuación diferencial
```

$$eqn:=\left(\frac{d^2}{dt^2}\mathrm{y}(t)\right)-\mathrm{y}(t)=\cos(t)\sin(t)$$

```
> fourier(eqn,t,w);
```

$$-\mathrm{fourier}(\mathrm{y}(t),t,w)\,(1+w^2)=\frac{1}{2}\,I\,\pi\,(\mathrm{Dirac}(w+2)-\mathrm{Dirac}(w-2))$$

```
> solve(%,'fourier'(y(t),t,w)); #resuelve en función de la
transformada de fourier de y(t).
```

$$\frac{\frac{-1}{2}\,I\,\pi\,(\mathrm{Dirac}(w+2)-\mathrm{Dirac}(w-2))}{1+w^2}$$

```
> invfourier(%,w,t);
```

$$-\frac{1}{10}\sin(2\,t)$$

```
> subs(y(t)=%,eqn); #comprobamos la solución
```

$$\left(\frac{d^2}{dt^2}\left(-\frac{1}{10}\sin(2\,t)\right)\right)+\frac{1}{10}\sin(2\,t)=\cos(t)\sin(t)$$

```
> eval(%);
```

$$\frac{1}{2}\sin(2\,t)=\cos(t)\sin(t)$$

```
> combine(%,trig);
```

$$\frac{1}{2}\sin(2\,t)=\frac{1}{2}\sin(2\,t)$$

5.6.3. Función addtable

Como ya hemos visto en algunos ejemplos en esta sección, la función ***addtable*** nos permite añadir una entrada a la tabla de transformadas integrales de la que dispone el programa. Su sintaxis más sencilla es ***addtable(****tname, patt, expr, t,s, parameter****),*** donde *tname* es el nombre de la transformada donde vamos a añadir nuestra entrada, *patt* la entrada que queremos añadir, *expr* la transformada de *patt*, *t* la variable independiente en *patt*, *s* la variable independiente en *expr* y *parameter* la lista de parámetros en *patt* y *expr* (opcional). Una vez ejecutado este comando, cualquier llamada a la función *tname* con argumento *patt* devolverá como resultado *expr*. Veamos algunos ejemplos:

```
> with(inttrans):

> fourier(f(t),t,w);
```

$$\text{fourier}(f(t),\,t,\,w)$$

```
> addtable(fourier,f(t),F(w),t,w): #le indicamos que la trans. de
f(t) es F(w)
> fourier(f(t),t,w);
```

$$F(w)$$

```
> laplace(g(3*p+2),p,x);
```

$$\text{laplace}(g(3\,p+2),\,p,\,x)$$

```
> addtable(laplace,g(a*x+b),G(s+a)/(b-a),x,s,{a,b}); #añadiendo
como parámetros a y b
> laplace(g(3*p+2),p,x);
```

$$-G(x+3)$$

A veces puede ocurrir que estemos interesados en guardar la información introducida en las tablas para luego ser empleada en otras sesiones. En este caso tendremos que utilizar la función ***savetable*** que nos permite guardar la información de una tabla en particular en un archivo concreto. Una vez generado el archivo, no tendremos mas que leerlo al iniciar la siguiente sesión para recuperar la tabla deseada. Su sintaxis es ***savetable(****tabla, archivo_destino****).*** Veamos un ejemplo:

```
> restart;
```

```
> with(inttrans):
> addtable(laplace,f(t),F(s),t,s);
> laplace(f(t),t,s);
```

$$F(s)$$

```
> savetable(laplace,`Mi_tabla.m`);
```

Ahora utilizamos la tabla en otra sesión:

```
> restart;
> with(inttrans):
> read(`Mi_tabla.m`);
> laplace(f(t),t,s);
```

$$F(s)$$

5.7. FUNCIONES DE ESTADÍSTICA. *(stats)*

Para poder acceder a las funciones de esta librería es necesario cargarla previamente, en caso de que no esté cargada Maple mostrará a la salida lo mismo que se introduce a la entrada, sin haber realizado ningún cálculo.

Para cargar todas las funciones de esta librería usaremos el comando *with* de la siguiente manera:

```
> with(stats);
```

[*anova, describe, fit, importdata, random, statevalf, statplots, transform*]

Lo que acabamos de hacer es cargar todos los subpaquetes de los que dispone la librería *stats*, si no nos interesan todos los subpaquetes podemos limitarnos a cargar uno en concreto de la siguiente manera; *with(stats, subpaquete)*. Para acceder a una función de la librería se hace mediante *subpackage[function](args)*.

El primer subpaquete que aparece en la lista es *anova*, éste nos permite hacer varios análisis de varianza de un conjunto de datos estadísticos.

El subpaquete *describe* nos proporciona varias funciones estadísticas descriptivas para el análisis de unos datos estadísticos.

- **coefficientofvariation***:* El coeficiente de variación se define como la desviación estándar dividida entre la media, nos da una idea de la dispersión relativa de los datos. Partiendo de que los datos son una lista estadística, la manera de utilizar el comando es *describe[coefficientofvariation[Nconstraints]](datos);* donde *datos* es una lista estadística y *Nconstraints* vale 1 si se trata de una muestra y 0 si es toda la población, esta última está puesta por defecto a 0. Ej:

```
> with(stats);
```

[*anova, describe, fit, importdata, random, statevalf, statplots, transform*]

```
> datos1:=[7,8,9];
```

$$datos1 := [7, 8, 9]$$

```
> datos2:=[137,138,139];
```

$$datos2 := [137, 138, 139]$$

```
> describe[standarddeviation](datos1)=
  describe[standarddeviation](datos2);
```

$$\frac{\sqrt{6}}{3} = \frac{\sqrt{6}}{3}$$

Aunque tienen la misma desviación estándar la dispersión relativa no es la misma:

```
> describe[coefficientofvariation](datos1): evalf(%);
```

$$0.1020620726$$

```
> describe[coefficientofvariation](datos2): evalf(%);
```

$$0.005916641891$$

- **count**: Cuenta el número de observaciones que no se han perdido en el conjunto de datos. La sintaxis es *describe[count](datos).* Ej:

```
> datos1:=[18,5,Weight(5,4)];
```

$$datos1 := [18, 5, \text{Weight}(5, 4)]$$

*NOTA: Weight(5,4) representa que el número cinco está cuatro veces en la lista. Con Weight también se pueden introducir datos cercanos, por ejemplo si tenemos Weight(0.002..0.003,2), esto quiere decir que hay dos datos entre 0.002 y 0.003.

```
> describe[count](datos1); Tenemos 6 elementos en total.
```

$$6$$

```
> data2:=[Weight(5,4),missing, 2, Weight(11..12,5)];
```

$$data2 := [\text{Weight}(5, 4), missing, 2, \text{Weight}(11 .. 12, 5)]$$

```
> describe[count](data2);  #Tenemos 10 elementos conocidos
```

$$10$$

- **countmissing**: Cuenta el número de observaciones que se han perdido en el conjunto de datos. La sintaxis es *describe[countmissing](datos).* Ej:

```
> datos1:=[Weight(3,10),missing, 4, Weight(11..12,3)];
```

$$datos1 := [\text{Weight}(3, 10), missing, 4, \text{Weight}(11 .. 12, 3)]$$

```
> describe[countmissing](datos1);
```

$$1$$

- **covariance**: Nos da la covarianza entre dos listas estadísticas, las listas deben de tener el mismo número de observaciones, ya que cov(X,Y)=E(XY)-E(X)E(Y). Por tanto para conocer E(XY) la cantidad de observaciones en X y en Y deberán ser iguales. La llamada es de la forma *describe[covariance](datos1, datos2).* Ej:

```
> with(stats):
datos1:=[1,5,7,8];
```

$$datos1 := [1, 5, 7, 8]$$

```
> datos2:=[22,34,6,8,4,345];
```

$$datos2 := [22, 34, 6, 8, 4, 345]$$

```
> describe[covariance](datos1, datos2);
Error, (in stats/abort) [[describe[covariance], needs lists of
identical number of items, received, [1, 5, 7, 8], [22, 34, 6, 8,
4, 345]]]
```

Nos da el error porque las listas no tienen el mismo número de elementos, entonces:

```
> datos2:=[22,34,6,8];
```

$$datos2 := [22, 34, 6, 8]$$

```
> describe[covariance](datos1, datos2);
```

$$\frac{-139}{8}$$

- **decile**: Nos devuelve el decil correspondiente al número que le pasamos como argumento entre paréntesis, si está entre dos elementos de la lista se hace una interpolación. La sintáxis es *describe[decile[which]](data, gap)*, donde *which* es el decil que se quiere calcular y *gap* (opcional) es el hueco entre las distintas clases.

```
> with(stats,describe);
```

$$[describe]$$

```
> datos1:=[seq(i,i=1..20)];
```

$$datos1 := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]$$

```
> describe[decile[1]](datos1); Obtenemos el primer decil.
```

$$2$$

```
> datos2:=[seq(i,i=1..17)];
```

$$datos2 := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]$$

```
> describe[decile[4]](datos2); Obtenemos el cuarto decil.
```

$$\frac{34}{5}$$

Vemos que aquí hace una interpolación ya que no hay un dato que sea el 4/10 de la distribución ordenada

- **geometricmean**: Nos devuelve la media geométrica de un conjunto de datos. La sintaxis es *describe[geometricmean](data)*. Ej:

```
> datos:=[2,4,6,8];
```

$$datos := [2, 4, 6, 8]$$

```
> describe[geometricmean](datos); evalf(%);
```

$$384^{(1/4)}$$

$$4.426727679$$

- **harmonicmean**: Nos devuelve la media harmónica de un conjunto de datos. **1/H=(1/N)*sum(1/X)** siendo H la media harmonica, N el número de datos y X cada dato.

  La sintaxis es *describe[harmonicmean](data)*, Ej:

```
> datos:=[1,5,121,34,2345,34,6];
```

$$datos := [1, 5, 121, 34, 2345, 34, 6]$$

```
> describe[harmonicmean](datos);
```

$$\frac{40518786}{8301611}$$

- **kurtosis**: Devuelve el coeficiente de curtosis de un conjunto de datos. La sintaxis es *describe[kurtosis[Nconstraints]](data)*, donde *Nconstraints* vale 1 si se trata de una muestra y 0 si es toda la población, esta última está puesta por defecto a 0. Ej:

```
> datos1:=[Weight(2,2),Weight(3,20),Weight(4,2)];
```

$$datos1 := [\text{Weight}(2, 2), \text{Weight}(3, 20), \text{Weight}(4, 2)]$$

```
> describe[kurtosis](datos1);
```

$$6.$$

```
> datos2:=[Weight(2,2),Weight(3,3),Weight(4,2)];
```

$$datos2 := [\text{Weight}(2, 2), \text{Weight}(3, 3), \text{Weight}(4, 2)]$$

```
> describe[kurtosis](datos2);evalf(%);
```

$$\frac{7}{4}$$

$$1.750000000$$

- **linearcorrelation**: Computa el coeficiente de correlación lineal entre dos listas estadísticas. La sintaxis es *describe[linearcorrelation](data1, data2)*. Ej:

```
> datos1:=[-23,43,332];
```

$$datos1 := [-23, 43, 332]$$

```
> datos2:=[127,23,-3];
```

$$datos2 := [127, 23, -3]$$

```
> describe[linearcorrelation](datos1,datos2): evalf(%);
```

$$-0.7766957130$$

- **mean**: Calcula la media aritmética de una lista dada. La sintaxis es *describe[mean](data)*. Ej:

```
> describe[mean](datos);evalf(%);
```

$$\frac{29}{4}$$

$$7.250000000$$

- **meandeviation**: Calcula la desviación media de una lista dada (no es la desviación estándar, sino la media de los valores absolutos de la distancia respecto a la media). La sintaxis es la que sigue, *describe[meandeviation](datos)*. Ej:

```
> describe[meandeviation]([1,3,7]),
describe[meandeviation]([2,3,5]);
```

$$\frac{20}{9}, \frac{10}{9}$$

- **median**: Nos da la mediana de unos datos. La sintaxis es *describe[median](datos)* . Ej:

Cuando el número de observaciones es par se hace mediante interpolación.

```
> describe[median]([1,2,3,4]);
```

$$\frac{5}{2}$$

- **mode**: Nos da la moda de unos datos. La sintaxis es *describe[mode](datos)*, Ej:

```
> data2:=[6,Weight(12,7),Weight(6,7),5, missing];
```

$$data2 := [6, \text{Weight}(12, 7), \text{Weight}(6, 7), 5, missing]$$

```
> describe[mode](data2);
```

$$6$$

- **moment**: Calcula el momento de orden n respecto a cualquier origen, los argumentos se pasan de la siguiente manera: *describe[moment[which,origen, Nconstraint]](data)*, donde *which* significa el orden del momento; *origen* por defecto es el 0; *y Nconstraint* vale 1 si se trata de una muestra y 0 si es toda la población, esta última está puesta por defecto a 0. Ej:

```
> datos:=[1,3,7,11,13,12,17,20];
```

$$datos := [1, 3, 7, 11, 13, 12, 17, 20]$$

```
> describe[moment[3]](datos);    #Momento de orden 3 respecto al 0.
```

$$\frac{4635}{2}$$

```
> describe[moment[2,6]](datos); #Momento de orden 2 respecto al 6.
```

$$\frac{231}{4}$$

```
> describe[moment[3,mean,1]](datos); #Momento de orden 3 respecto a
la media para una población de muestra.
```

$$\frac{-171}{7}$$

- **percentile**: Nos devuelve el percentil requerido, en caso necesario será interpolado. La sintáxis es *describe[percentile[which]](data)* donde *which* es el percentil que se quiere calcular. Ej:

```
> data:=[seq(i/19+17*i, i=1..20)];describe[percentile[15]](data);
```

$$data := \left[\frac{324}{19}, \frac{648}{19}, \frac{972}{19}, \frac{1296}{19}, \frac{1620}{19}, \frac{1944}{19}, \frac{2268}{19}, \frac{2592}{19}, \frac{2916}{19}, \frac{3240}{19}, \frac{3564}{19}, \frac{3888}{19}, \frac{4212}{19}, \frac{4536}{19}, \frac{4860}{19}, \frac{5184}{19}, \frac{5508}{19}, \frac{5832}{19}, 324, \frac{6480}{19}\right]$$

$$\frac{972}{19}$$

Los percentiles requeridos son:

```
> mispercentiles:=[seq(describe[percentile[9*i]],i=1..6)];
```

$$mispercentiles := [describe_{percentile_9}, describe_{percentile_{18}}, describe_{percentile_{27}}, describe_{percentile_{36}}, describe_{percentile_{45}}, describe_{percentile_{54}}]$$

Los imprimimos en pantalla:

```
> mispercentiles(data);
```

$$\left[\frac{2916}{95}, \frac{5832}{95}, \frac{8748}{95}, \frac{11664}{95}, \frac{2916}{19}, \frac{17496}{95}\right]$$

- **quadraticmean**: Computa la media cuadrática de un conjunto de datos, la sintaxis es *describe[quadraticmean](datos)*. Recordar que la media cuadrática es la raíz cuadrada de la media de los datos u observaciones al cuadrado. Ej:

```
> datos:=[seq((i+i*2)/i^2,i=1..10)];
```

$$datos := \left[3, \frac{3}{2}, 1, \frac{3}{4}, \frac{3}{5}, \frac{1}{2}, \frac{3}{7}, \frac{3}{8}, \frac{1}{3}, \frac{3}{10}\right]$$

```
> describe[quadraticmean](datos); evalf(%);
```

$$\frac{\sqrt{3936658}}{1680}$$

$$1.181012683$$

- **quantile**: Nos devuelve el cuantil requerido, la sintaxis es *describe[quantile[which, offset]](data)*, donde *which* es el cuantil requerido como una fracción entre 1 y 0 y *offset* es una cantidad añadida a la posición calculada . La posición del cuantil se obtiene mediante la fórmula r*n + offset, siendo r el *which* y n el número total de datos Ej:

```
> datos:=[seq(i/9, i=1..20)];describe[quantile[1/7]](datos);
```

$$datos := \left[\frac{1}{9}, \frac{2}{9}, \frac{1}{3}, \frac{4}{9}, \frac{5}{9}, \frac{2}{3}, \frac{7}{9}, \frac{8}{9}, 1, \frac{10}{9}, \frac{11}{9}, \frac{4}{3}, \frac{13}{9}, \frac{14}{9}, \frac{5}{3}, \frac{16}{9}, \frac{17}{9}, 2, \frac{19}{9}, \frac{20}{9}\right]$$

$$\frac{20}{63}$$

```
> misquantiles:=[seq(describe[quantile[i/20]],i=3..11)]:
misquantiles(datos);
```

$$\left[\frac{1}{3}, \frac{4}{9}, \frac{5}{9}, \frac{2}{3}, \frac{7}{9}, \frac{8}{9}, 1, \frac{10}{9}, \frac{11}{9}\right]$$

- **quartile**: Nos devuelve el cuartil requerido, la sintaxis es *describe[quartile[which]](datos)*, donde *which* es el cuartil requerido. Ej:

Podemos hacer la misma operación con quantile:

```
> datos:=[10,20,30,40,50,60,70,80];
```

```
> describe[quartile[1]](datos)=describe[quantile[1/4]](datos);
```

$$datos := [10, 20, 30, 40, 50, 60, 70, 80]$$

$$20 = 20$$

- **range**: Esta función nos devuelve el valor máximo y el mínimo de las observaciones en la lista estadística. La sintaxis es *describe[range](data).* Ej:

```
> datos:=[seq(i/5,i=1..10)];
```

$$datos := \left[\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1, \frac{6}{5}, \frac{7}{5}, \frac{8}{5}, \frac{9}{5}, 2\right]$$

```
> describe[range](datos);
```

$$\frac{1}{5} .. 2$$

- **skewness**: Nos devuelve el coeficiente de asimetría o deformación. La sintaxis es *describe[skewness[Nconstraints]](datos)*, donde *Nconstraints* vale 1 si se trata de una muestra y 0 si es toda la población. Esta última está puesta por defecto a 0. Ej:

```
> describe[skewness]([1,6,7]);
```

$$-\frac{77}{961}\sqrt{62}$$

- **standarddeviation**: Devuelve la desviación estándar de un conjunto de datos. Sintaxis: *describe[standarddeviation[Nconstraints]](datos*), donde *Nconstraints* vale 1 si se trata de una muestra y 0 si es toda la población, esta última está puesta por defecto a 0. Ej:

```
> datos:=[1,4,3,234,234,334,2,-23,2,3,0];
```

$$datos := [1, 4, 3, 234, 234, 334, 2, -23, 2, 3, 0]$$

```
> describe[standarddeviation](datos);evalf(%);
```

$$\frac{2\sqrt{451901}}{11}$$

$$122.2246948$$

- **sumdata**: Computa la suma de distintas potencias de los datos dados respecto a cualquier origen, es decir la r-ésima potencia respecto a un origen S es sum((X-S)^r). La sintaxis es *describe[sumdata[which, origin, Nconstraint]](datos)*, donde *which* es el valor de la potencia que se quiere calcular (1 por defecto), *origin* es el origen (0 por defecto), y *Nconstraints* vale 1 si se trata de una muestra y 0 si es toda la población, esta última está puesta por defecto a 0. Ej:

```
> datos:=[3,19,23,21,4,14,2,7,8,5,10];
```

$$datos := [3, 19, 23, 21, 4, 14, 2, 7, 8, 5, 10]$$

```
> describe[sumdata[2,mean]](datos);evalf(%);
```

$$\frac{6278}{11}$$

$$570.7272727$$

- **variance**: Nos da la varianza de una lista estadística, la sintaxis es *describe[variance[Nconstraints]](datos)*, donde *Nconstraints* vale 1 si se trata

de una muestra y 0 si es toda la población, esta última está puesta por defecto a 0. Ej:

```
> datos:=[3,19,23,21,4,14,2,7,8,5,10];
```

$$datos := [3, 19, 23, 21, 4, 14, 2, 7, 8, 5, 10]$$

```
> describe[variance](datos)=describe[standarddeviation](datos)^2;
```

$$\frac{6278}{121} = \frac{6278}{121}$$

El siguiente **subpaquete** es ***fit***, ofrece herramientas para el ajuste o aproximación de curvas a unos datos estadísticos. Estas funciones no pueden trabajar con datos no numéricos, en caso de hacerlo los comandos quedarán sin evaluar. Las funciones que contiene *fit* son:

- **leastsquare**: Aproxima una curva a los datos estadísticos usando el método de los mínimos cuadrados. Las observaciones perdidas no se tienen en cuenta en los cálculos. La sintaxis es *fit[leastsquare[vars, eqn, parms]](datos)*, donde *vars* son la lista de variables, correspondiendo en orden a las de las listas estadísticas; *eqn* (opcional) es la ecuación por la que se quiere aproximar y que por defecto es una ecuación lineal, con la última variable en var como variable dependiente. Por último, *parms* (opcional), son un conjunto de parámetros que serán sustituidos. Ejemplo:

```
> with(stats):
fit[leastsquare[[x,y,z]]]([[1,2,3,5],[2,4,6,8],[3,5,7,10]]); # los
datos que están entre corchetes son X, Y, Z respectivamente.
```

$$z = 1 + x + \frac{y}{2}$$

z como variable dependiente.

```
> fit[leastsquare[[x,y,z]]] ([[1,2,3,5,5,5], [2,4,6,8,8,8],
[3,5,7,10,15,15]]);
```

$$z = 1 + \frac{13}{3}x - \frac{7}{6}y$$

```
>fit[leastsquare[[x,y,z]]]([[1,2,3,5,5], [2,4,6,8,8],
[3,5,7,10,Weight(15,2)]]);
```

$$z = 1 + \frac{13}{3}x - \frac{7}{6}y$$

Aproximación a una curva cuadrática.

```
> Xvalores:=[1,2,3,4];
```

$$Xvalores := [1, 2, 3, 4]$$

```
> Yvalores:=[0,6,14,24];
```

$$Yvalores := [0, 6, 14, 24]$$

```
> eq_fit:= fit[leastsquare[[x,y], y=a*x^2+b*x+c,
{a,b,c}]]([Xvalores, Yvalores]);  #en este caso {a,b,c} son los
parámetros que serán sustituidos.
```

$$eq\_fit := y = x^2 + 3x - 4$$

- **leastmediansquare**: Nos aproxima o ajusta una serie de listas estadísticas a una curva mediante el método de los mínimos medianos cuadrados. La ecuación que

se aproxima debe ser lineal en los parámetros que buscamos. Las observaciones perdidas no se tienen en cuenta en los cálculos. La sintaxis es *fit[leastmediansquare[vars]](data)*, donde *vars* son la lista de variables, correspondiendo en orden a las de las listas estadísticas. Ej:

En primer lugar ponemos los datos de manera que podamos trabajar con ellos.

```
> Data:=convert(linalg[transpose]([[1,3],[2,4],[3,5],[1,2]]),
listlist);
```

$$data := [[1, 2, 3, 1], [3, 4, 5, 2]]$$

```
> fit[leastmediansquare[[x,y]]](data);
```

$$y = 2 + x$$

El siguiente **subpaquete** que nos encontramos en stats es ***random***. Aquí encontraremos herramientas para generar números aleatorios siguiendo una distribución determinada. Primero se genera una distribución uniforme de números aleatorios y después mediante distintos filtros se transforman en una distribución concreta de números aleatorios. La función para poder hacer esto es random, la sintaxis es random[distribution](quantity, uniform, method) o stats[random,distribution](quantity, uniform, method) si no hemos abierto la librería stats, donde distribution es la distribución de acuerdo a la cual se quieren generar los números aleatorios. Las distribuciones posibles son:

- Distribuciones discretas:

  binomiald[n,p] discreteuniform[a,b]

  empirical[list_prob] hypergeometric[N1, N2, n]

  negativebinomial[n,p] poisson[mu]

- Distribuciones continuas:

  beta[nu1, nu2] cauchy[a, b] chisquare[nu]

  exponential[alpha, a] fratio[nu1, nu2] gamma[a, b]

  laplaced[a, b] logistic[a, b] lognormal[mu, sigma]

  normald[mu, sigma] studentst[nu] uniform[a, b]

  weibull[a, b]

El parámetro *quantity* es la cantidad de números que se desea generar, *uniform(opcional)* es para generar los números usando flujos independientes, y *method* es el método empleado, que puede ser auto, inverse o builtin. Ejemplos:

1. Distribución normal.

```
> stats[random, normald](20);
```

-0.1049905652, 1.996496081, -1.257880271, -0.05641157088, 1.113504099, 0.3691334664, -0.03153626578, 0.6190037193, 1.743790472, -1.119097459, -0.4917291616, -0.4647978628, 0.4533976549, -0.6373819594, -0.9470399048, 1.597348970, 0.4529719537, -1.526885022, 0.3519454534, 0.6335404689

2. Distribución Chi cuadrado con 3 grados de libertad.

```
> stats[random,chisquare[3]](20);
```

5.138542272, 1.282240289, 1.067665866, 0.3978314268, 0.6652482030, 0.2302201352, 4.876179684, 5.017065798, 6.645459908, 1.923756411, 4.534898778, 0.5545206484, 4.263350708, 1.503923388, 4.787752364, 1.299340718, 0.4840973072, 4.144630036, 7.608969618, 4.664898018

3. Distribución Poisson con lambda igual a 3.

```
> stats[random,poisson[3]](20);
```

1, 5, 2, 5, 1, 5, 5, 5, 5, 3, 6, 2, 5, 0, 1, 3, 3, 2, 2, 4

El **subpaquete *statevalf*** permite hacer cálculos numéricos de funciones estadísticas, las funciones que tenemos para el caso continuo son *cdf* (función de ditribución), *icdf* (función de distribución inversa) y *pdf* (función de densidad), análogamente para el caso discreto tenemos *dcdf* (función de distribución), *idcdf* (función de distribución inversa) y *pf* (función de masa). La sintaxis es *statevalf[function, distribution](args)*, donde *function* es una de las arriba citadas, *distribution* es la distribución con la que se quiere trabajar, y *args* el valor de la variable de la función (por ejemplo en la función de densidad f(x), args sería x).

Ejemplos:

```
> statevalf[pdf,normald](3);
```

0.004431848412

```
> statevalf[icdf,normald[7,2]](0.39);
```

6.441361931

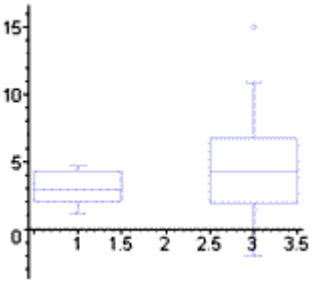
```
> statevalf[cdf,normald](2);
```

0.9772498681

El subpaquete statplot sirve para hacer representaciones gráficas de listas de datos:

- ***boxplot***: Representa los valores de los datos en y. El eje de los x no representa mucho, ya que se puede cambiar con width=w (la anchura del boxplot) y shift=s (donde queremos que se centre). Veamos un ejemplo en el que dibujamos dos boxplots:

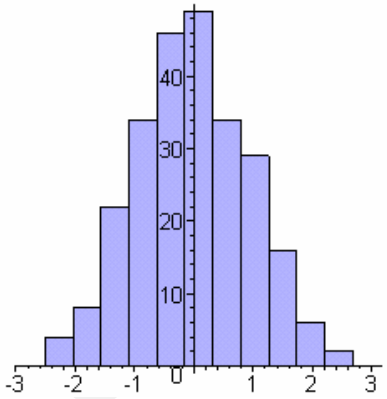
```
> with(stats[statplots]);
data1:=[2.93, 2.58, 2.85, 4.26, 2.94, 4.33, 1.71, 4.42, 3.59,
  4.35, 2.07, 1.16, 2.36, 1.16, 4.72]:
data2:=[2.46, 4.34, .182, 3.22, 5.37, 15, 3.11, -1.99, -.865,
  2.56, 10.6, 10.9, 6.56, 7.22, 4.84]:
boxplot(data1, data2, shift=1);
```



La raya horizontal de la mitad representa la mediana, mientras las dos siguientes representan el primer y tercer cuartil. Las dos rayas que se extienden tienen como longitud máxima 3/2 del rango intercuartil (longitud entre el primer y tercer cuartil), aunque nunca superará el rango de los datos. Finalmente, los datos que se encuentren fuera se representan como datos.

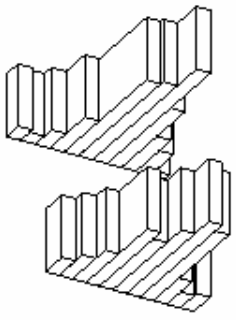
- ***histogram***: Representa el histograma de la lista de datos. Se puede especificar el número de barras mediante numbars=x. Por defecto el area de todas las barras será igual y la suma de todas igual a 1. Si especificamos la característica area=a, entonces el ancho de todas las barras será igual y la suma de todas las areas sera igual a 'a '(con area nos referimos al valor que toma y, y por tanto la suma será la suma de los valores de cada barra). Si ponemos area=count, el área total será igual al número total de datos.

```
> histogram([random[normald](250)],area=count);
```



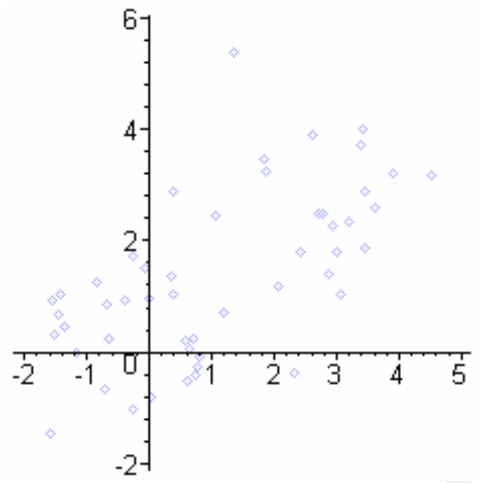
También se pueden representar dos variables aleatorias a la vez dibujando el histograma en tres dimensiones:

```
> data1:=[random[normald](200)]:
> data2:=[random[normald[1,1]](200)]:
> histogram(data1,data2,area=count);
```



- ***scatterplot***: Sirve para representar en un plano los datos. Se puede representar una variable en una dimensión, o dos variables en dos dimensiones.

```
> data1:=[random[normald](30), random[normald[3,1]](20)]:
data2:=[random[normald](30), random[normald[3,1]](20)]:
scatterplot(data1, data2);
```



5.7.1. Ejercicios globales

Ejercicio 1

Determinar las características de tendencia central y de dispersión más importantes de la siguiente muestra de 50 edades de ejecutivos.

38	50	35	46	63	69	54	62	68	40
48	44	55	43	42	59	54	57	47	46
42	60	43	64	49	36	59	42	60	38
61	56	51	50	66	63	57	51	38	45
62	37	50	44	48	69	64	56	53	52

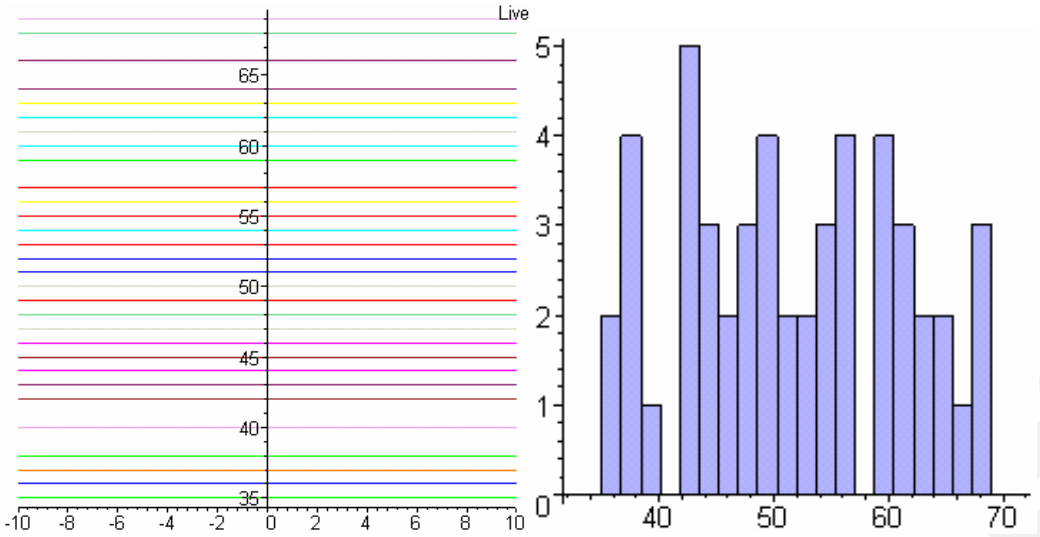
```
> with(stats[describe]):
```

Lo primero es crear una lista con todos los datos.

```
> datos:=[38, 50, 35, 46, 63, 69, 54, 62, 68, 40,
> 48, 44, 55, 43, 42, 59, 54, 57, 47, 46,
> 42, 60, 43, 64, 49, 36, 59, 42, 60, 38,
> 61, 56, 51, 50, 66, 63, 57, 51, 38, 45,
> 62, 37, 50, 44, 48, 69, 64, 56, 53, 52];
```

datos := [38, 50, 35, 46, 63, 69, 54, 62, 68, 40, 48, 44, 55, 43, 42, 59, 54, 57, 47, 46, 42, 60, 43, 64, 49, 36, 59, 42, 60, 38, 61, 56, 51, 50, 66, 63, 57, 51, 38, 45, 62, 37, 50, 44, 48, 69, 64, 56, 53, 52]

```
> smartplot(datos); histogram(datos,area=count,numbars=20);
```



Calculamos ahora las características de tendencia central y de dispersión

Valores extremos

```
> range(datos);
```

$$35 .. 69$$

Mediana

```
> median(datos);
```

$$51$$

Media

```
> mean(datos);evalf(%);
```

$$\frac{1293}{25}$$

$$51.72000000$$

Varianza

```
> variance(datos):evalf(%);
```

$$89.12160000$$

Desviación estándar

```
> standarddeviation[1](datos):evalf(%);
```

$$9.536268042$$

Si calculamos este parámetro mediante la definición

```
> mean(datos)=(sumdata[1](datos))/count(datos);
```

$$\frac{1293}{25} = \frac{1293}{25}$$

```
> variance(datos)=(sumdata[2,mean](datos))/count(datos);
```

$$\frac{55701}{625} = \frac{55701}{625}$$

Podemos calcular los cuartiles por ejemplo

```
> quartiles= seq(quartile[i](datos),i=1..3);
```

$$quartiles = \left(\frac{87}{2}, 51, \frac{119}{2} \right)$$

Coeficientes de asimetría y curtosis

```
> kurtosis(datos):evalf(%);
```

1.956784408

Es bastante grande porque como vemos en el histograma los datos no están muy centrados en la media.

```
> skewness(datos):evalf(%);
```

0.05900558828

Es bastante pequeño,lo que indica que la distribución es bastante simétrica, que por otro lado se ve que lo es en el histograma.

Ejercicio 2

Vamos a realizar el siguiente ejercicio de dos formas, utilizando las funciones por un lado, y generando una muestra de datos de esa distribución y analizándola por otro. El problema:

Si los errores aleatorios que se cometen al realizar unas pesadas en una balanza siguen una distribución Normal, de media 0 y desviación típica 2 decigramos, calcular:

a-) ¿Cual será el error máximo que se comete el 95% de las veces?

b-) Si se realizan 50 pesadas, ¿con qué probabilidad en menos del 10% de ellas, se supera este máximo? (en este apartado haremos una sola muestra, es decir los 50 datos de las 50 pesadas)

a-) Modo1:

```
> with(statevalf);
```

[*cdf*, *dcdf*, *icdf*, *idcdf*, *pdf*, *pf*]

```
> er:=icdf[normald[0,2]](0.975); #en decigramos
```

$er := 3.919927969$

Modo 2:

```
> datos:=[random[normald[0,2]](1000)]:
```

```
> describe[quantile[975/1000]](datos);
```

3.859800824

Vemos que se parecen bastante.

b-) Modo 1:

```
> p:=dcdf[binomiald[50,0.05]](4);
```

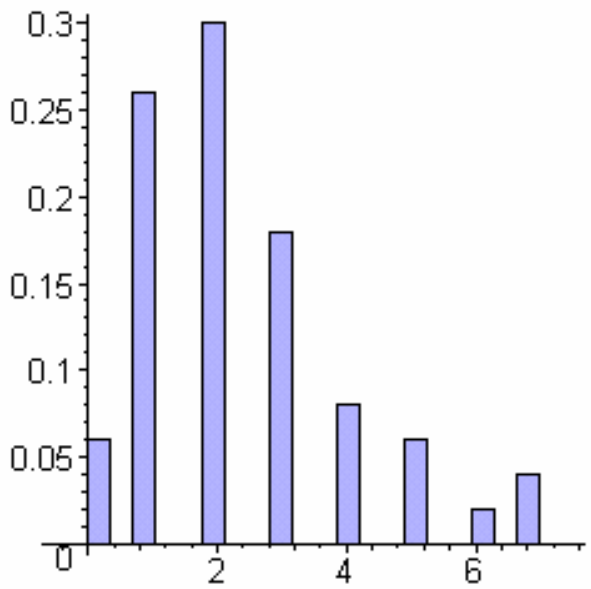
$$p := 0.8963831899$$

Modo 2:

```
> datos1:=[random[binomiald[50,0.05]](50)];
```

datos1 := [2, 1, 7, 1, 3, 2, 2, 1, 4, 4, 2, 1, 5, 1, 5, 2, 0, 1, 1, 2, 4, 4, 2, 6, 3, 3, 1, 2, 1, 5, 2, 2, 0, 2, 1, 1, 3, 1, 7, 2, 3, 3, 2, 0, 3, 1, 2, 2, 3, 3]

```
> histogram(datos1,numbars=20,area=1);
```



Más o menos p(errores<5)=1-p(x=5,6,7)=0.88 (vemos que coincide bastante con el valor teórico).

5.8. MATLAB

Desde Maple podemos utilizar algunas funciones de Matlab, supuesto que Matlab esté instalado en el equipo. Para ello hay que abrir el paquete Matlab y automáticamente se abrirá la ventana de comandos del programa:

```
>.with(Matlab);
```

[*chol*, *closelink*, *defined*, *det*, *dimensions*, *eig*, *evalM*, *fft*, *getvar*, *inv*, *lu*, *ode15s*, *ode45*, *openlink*, *qr*, *setvar*, *size*, *square*, *transpose*]

Para cerrar Matlab se utiliza la funcion

```
> closelink();
```

Estas son las funciones que se pueden utilizar. Algunas de ellas las veremos a continuación.

Primero veamos como se trabaja con las variables. Matlab trabaja con matrices, y aunque sea una sola constante, en Matlab será una matriz 1×1. Para utilizar en Maple una matriz creada en Matlab, habrá que utilizarla entre comillas dobles, pero esta matriz no podrá utilizarse para funciones de Maple, sino para las funciones de Matlab mencionadas arriba. Sucede lo mismo en el sentido contrario, ya que no se puede utilizar directamente en Matlab una matriz creada en Maple. Para poder utilizarlas habrá que utilizar unas funciones que veremos más adelante.

Matlab y Maple utilizan memoria diferente, por tanto se podrá llamar M a dos matrices diferentes, una creada en Maple y la otra en Matlab.

- **getvar**: sirve para extraer una variable de Matlab a Maple y así poder utilizarla en este último. Ej:

Primero creamos una matriz en Matlab:

```
» A=[1 1 1; 2 2 2;3 3 3]

A =
     1     1     1
     2     2     2
     3     3     3
```

```
> with(Matlab);

> with(LinearAlgebra):  #abro esta libreria para crear una matriz
en Maple.
> getvar("A");
```

$$\begin{bmatrix} 1. & 1. & 1. \\ 2. & 2. & 2. \\ 3. & 3. & 3. \end{bmatrix}$$

```
> evalm(getvar("A")&*IdentityMatrix(3));
```

$$\begin{bmatrix} 1. & 1. & 1. \\ 2. & 2. & 2. \\ 3. & 3. & 3. \end{bmatrix}$$

- **setvar**: sirve para copiar una variable de Maple a una variable de Matlab, o una de Matlab a una de Matlab. Lo que no se puede hacer mediante esta función es copiar de una variable de Matlab a una de Maple, aunque esto se puede hacer utilizando getvar: **`M:=getvar("A");`**. Cuando copiamos una variable, hay que tener en cuenta que cada programa utilizará su propia memoria para la variable, por tanto habrá esa variable por duplicado. Veamos unos ejemplos:

```
>  with(Matlab):
   maple_matrix_a:=[35,623,22,115];
   setvar("matlab_matrix_a", maple_matrix_a);
```

$$maple\_matrix\_a := [35, 623, 22, 115]$$

Si ahora llamamos a la variable en Matlab :
» matlab_matrix_a

```
matlab_matrix_a =

    35
   623
    22
   115
```

```
> setvar("matlab_matrix_b", "matlab_matrix_a");
```

Ahora tendremos en Matlab dos variables con el mismo contenido.

- **evalM**: Sirve para evaluar una expresión creada mediante la sintaxis de Matlab. Veamos cómo se utiliza mediante un ejemplo:

```
> maplematrix_a:=Matrix([[3,3,2],[4,5,2],[6,2,4]]);
```

$$maplematrix\_a := \begin{bmatrix} 3 & 3 & 2 \\ 4 & 5 & 2 \\ 6 & 2 & 4 \end{bmatrix}$$

```
> maplematrix_b:=Matrix([[3,2,5],[1,8,2],[7,3,4]]);
```

$$maplematrix\_b := \begin{bmatrix} 3 & 2 & 5 \\ 1 & 8 & 2 \\ 7 & 3 & 4 \end{bmatrix}$$

```
> setvar("maplematrix_a",maplematrix_a);
setvar("maplematrix_b",maplematrix_b);
> evalM("c=maplematrix_a*maplematrix_b");
> getvar("c");
```

$$\begin{bmatrix} 26. & 36. & 29. \\ 31. & 54. & 38. \\ 48. & 40. & 50. \end{bmatrix}$$

5.8.1. Otras funciones

Como vemos al abrir la librería Matlab, podemos utilizar algunas funciones de Matlab, de modo que las funciones las realizará Matlab y no Maple. Así, podemos calcular el determinante con ***det***, la inversa con ***inv***, ver las dimensiones de una matriz con ***dimensions*** o ***size***, calcular la transpuesta con ***transpose***, etc. A todas estas funciones se les puede pasar como argumento tanto una matriz creada en Matlab como en Maple.

La diferencia entre *dimensions* y *size* es que la primera utiliza Maple para calcular la dimensión y en la segunda en cambio Matlab. Por eso para matrices de Maple será mejor utilizar *dimensions*, y para las de Matlab *size*, ya que hará el cálculo más rápido.

Veamos unos ejemplos:

```
> with(Matlab):
> with(LinearAlgebra):
> A:=Matrix([[1,1,2],[1,0,1],[2,2,3]]);
```

$$A := \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 2 & 3 \end{bmatrix}$$

```
> det(A); Determinant(A);
```

$$1.$$

$$1$$

```
> setvar("B",A);
> evalM("B(1,1)=3"):
```

```
> getvar("B");
```

$$\begin{bmatrix} 3. & 1. & 2. \\ 1. & 0. & 1. \\ 2. & 2. & 3. \end{bmatrix}$$

```
> det("B");
```

$$-3.$$

```
> inv(A);inv("B");
```

$$\begin{bmatrix} -2. & 1. & 1. \\ -1. & -1. & 1. \\ 2. & 0. & -1. \end{bmatrix}$$

$$\begin{bmatrix} 0.666666666666666518 & -0.333333333333333370 & -0.333333333333333259 \\ 0.333333333333333093 & -1.666666666666666630 & 0.333333333333333426 \\ -0.666666666666666518 & 1.333333333333333326 & 0.333333333333333259 \end{bmatrix}$$

```
> transpose("B");
```

$$\begin{bmatrix} 3. & 1. & 2. \\ 1. & 0. & 2. \\ 2. & 1. & 3. \end{bmatrix}$$

```
> size("B");
```

$$[3, 3]$$

```
> dimensions(A);
```

$$[3, 3]$$

También se pueden obtener los **valores y vectores propios** mediante la función ***eig.*** Su forma es **eig(A,C)**, siendo el problema A . x = lambda*C * x . Tanto A como C pueden ser matrices creadas en Maple o Matlab. Si queremos que nos devuelva los vectores propios también, habrá que introducir como argumento 'eigenvectors'= 'true'. En este último caso nos devolverá primero una matriz cuyas columnas serán los vectores propios y seguido nos devolverá otra matriz cuyos elementos de la diagonal serán los valores propios. Veamos un ejemplo:

```
> maplematrix_a:=Matrix([[1,2,3],[3,4,5],[6,7,8]]);
```

$$maplematrix\_a := \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

```
> setvar("matlabmatrix_a",maplematrix_a);
> eig(maplematrix_a);
```

$$\begin{bmatrix} 14.0663729752107667 \\ -1.06637297521077668 \\ -0.379901471867316940 \quad 10^{-15} \end{bmatrix}$$

```
> eig("matlabmatrix_a",'eigenvectors'='true');
> eig("matlabmatrix_a",'eigenvectors'='true');
```

$$\begin{bmatrix} -0.265647760732583504 & -0.744428627863832571 & 0.408248290463863518 \\ -0.491207120532433317 & -0.190701223873281478 & -0.816496580927725924 \\ -0.829546160232207730 & 0.639889882112543052 & 0.408248290463862574 \end{bmatrix}$$

$$\begin{bmatrix} 14.0663729752107667 & 0. & 0. \\ 0. & -1.06637297521077668 & 0. \\ 0. & 0. & -0.379901471867316940\ 10^{-15} \end{bmatrix}$$

```
> Eigenvectors("matlabmatrix_a");
```

$$\begin{bmatrix} \frac{13}{2}+\frac{\sqrt{229}}{2} \\ \frac{13}{2}-\frac{\sqrt{229}}{2} \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{9\left(\frac{35}{2}+\frac{3\sqrt{229}}{2}\right)}{\left(\frac{67}{2}+\frac{7\sqrt{229}}{2}\right)\left(\frac{11}{2}+\frac{\sqrt{229}}{2}\right)} & \frac{9\left(\frac{35}{2}-\frac{3\sqrt{229}}{2}\right)}{\left(\frac{67}{2}-\frac{7\sqrt{229}}{2}\right)\left(\frac{11}{2}-\frac{\sqrt{229}}{2}\right)} & 1 \\ \frac{3\left(\frac{19}{2}+\frac{\sqrt{229}}{2}\right)}{\frac{67}{2}+\frac{7\sqrt{229}}{2}} & \frac{3\left(\frac{19}{2}-\frac{\sqrt{229}}{2}\right)}{\frac{67}{2}-\frac{7\sqrt{229}}{2}} & -2 \\ 1 & 1 & 1 \end{bmatrix}$$

*NOTA: Matlab calcula los datos numéricamente, por eso muchas veces no salen valores exactos. Por ejemplo el tercer autovalor es 0, pero Matlab no lo ha conseguido exactamente (en Maple en cambio sí).

También podemos calcular la **Transformada discreta de Fourier** mediante la función ***dft***. Como argumento le pasamos un vector (de Maple o de Matlab) y el nos devolverá otro vector de la misma dimensión. Si queremos que solo nos calcule la transformada de una parte de los elementos del vector, le pasaremos como argumento el nº de elementos que queremos utilizar. Veamos unos ejemplos:

```
> v := <1, 2, 3, 4, 3, 2, 1, 2, 3, 4>; #también se puede utilizar
un vector fila.
```

$$v := \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 3 \\ 2 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

```
> fft(v);
```

$$\begin{bmatrix} 25. + 0.\,I \\ 0.618033988749894902 - 1.90211303259030706\,I \\ -3.61803398874989490 + 4.97979656976556040\,I \\ -1.61803398874989490 + 1.17557050458494626\,I \\ -1.38196601125010510 + 0.449027976579585484\,I \\ -3. + 0.\,I \\ -1.38196601125010510 - 0.449027976579585484\,I \\ -1.61803398874989490 - 1.17557050458494626\,I \\ -3.61803398874989490 - 4.97979656976556040\,I \\ 0.618033988749894902 + 1.90211303259030706\,I \end{bmatrix}$$

Utilizando solo los dos primeros elementos:

```
> fft(v,2);
```

$$\begin{bmatrix} 3. \\ -1. \end{bmatrix}$$

También se pueden hacer descomposiciones de matrices, tales como la descomposición de Cholesky mediante la función ***chol***, la descomposición LU mediante la función ***lu*** y la QR mediante ***qr***. Veamos cómo funcionan con ejemplos (para más información buscar en el help: Mathematics→Numerical Computations→Matlab):

```
> with(Matlab):
> maplematrix_a:=Matrix([[3,1,3,5],[1,6,4,2],[6,7,8,1],[3,3,7,3]]);
```

$$maplematrix\_a := \begin{bmatrix} 3 & 1 & 3 & 5 \\ 1 & 6 & 4 & 2 \\ 6 & 7 & 8 & 1 \\ 3 & 3 & 7 & 3 \end{bmatrix}$$

```
> (L, U) := lu(maplematrix_a);
```

$$L, U := \begin{bmatrix} 0.500000000000000000 & -0.517241379310344750 & 0.115789473684210484 \\ 0.166666666666666658 & 1. & 0. \\ 1. & 0. & 0. \\ 0.500000000000000000 & -0.103448275862068950 & 1. \end{bmatrix}, \begin{bmatrix} 6. & 7. & 8. & 1. \\ 0. & 4.83333333333333392 & 2.66666666666666696 & 1.83333333333333326 \\ 0. & 0. & 3.27586206896551736 & 2.68965517241379314 \\ 0. & 0. & 0. & 5.13684210526315788 \end{bmatrix}$$

```
> maplematrix_b:=Matrix([[1,0],[0,3]]);
```

$$maplematrix\_b := \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}$$

```
> C:=chol(maplematrix_b);
```

$$C := \begin{bmatrix} 1. & 0. \\ 0. & 1.73205080756887720 \end{bmatrix}$$

```
> (Q,R):=qr(maplematrix_a);
```

$$Q, R :=$$
[-0.404519917477945468 , 0.418121005003545432 , -0.120768607347027060 , -0.804334137667873206]
[-0.134839972492648424 , -0.903141370807658106 , 0.0315048540905287778 , -0.406400406400609482]
[-0.809039834955890602 , -0.0836242010007090254 , -0.399061485146697980 , 0.423333756667301608]
[-0.404519917477945302 , 0.0501745206004254873 , 0.908389959610246600 , 0.0931334264668064182],
[-7.41619848709566209 , -8.09039834955890668 , -11.0568777443971715 ,

```
-4.31487911976475047        ]
[ 0. , -5.43557306504609006        , -2.67597443202269014        , 0.351221644202978078        ]
[ 0. , 0. , 2.92995143041917760        , 1.78527506512996404        ]
[ 0. , 0. , 0. , -4.13173746507286488        ]
```

5.9. COMPARTIR DATOS CON OTROS PROGRAMAS

Puede suceder que necesitemos utilizar datos que se encuentran en un archivo de otro programa. Por ejemplo, si queremos analizar estadísticamente un archivo en el que están los datos de un experimento (puede ser una cantidad muy grande para volver a escribir los datos en Maple).

Leer otros archivos puede tener dos objetivos. Uno es utilizar los datos almacenados en el archivo. El otro caso puede ser debido a que los comandos que queremos utilizar en Maple están en un archivo de texto. Ahora profundizaremos en ambos casos y también veremos como exportar datos que hayamos calculado en Maple a otro archivo.

5.9.1. Leer datos de un archivo

Lo que haremos será importar los datos de un archivo en modo de matriz. Utilizaremos la función ***ImportMatrix*(“filename”, delimiter=string)** cuando se trate de archivos con la extensión *.txt. En *filename* habrá que poner la ubicación del archivo además del nombre (en el ejemplo veremos cómo se escribe). En el sitio de *string* pondremos cuál será el carácter que delimite dos datos en el archivo. Si dos datos están separados por un espacio pondremos “ “ y si es una tabulación “\t” (la tabulación está por defecto). Cuando encuentre el delimitador saltará de columna en la matriz en que recogerá los datos. Cuando encuentre un salto de línea entonces saltará de fila en la matriz. Veamos un ejemplo:

Imaginemos que el archivo se llama “prueba.txt” y que se encuentra en una carpeta “Datos” en el disco C(usaremos esta carpeta para todos los ejemplos siguientes). Los datos en el archivo prueba.txt son:

```
0    1     1
1   .54   .84
2  -.41   .90
3  -.98   .14
4  -.65  -.75
5   .28  -.95
6   .96  -.27
```

En Maple:

```
> L:=ImportMatrix("c:\\Datos\\prueba.txt",delimiter=" ");
```

$$L := \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0.54 & 0.84 \\ 2 & -0.41 & 0.90 \\ 3 & -0.98 & 0.14 \\ 4 & -0.65 & -0.75 \\ 5 & 0.28 & -0.95 \\ 6 & 0.96 & -0.27 \end{bmatrix}$$

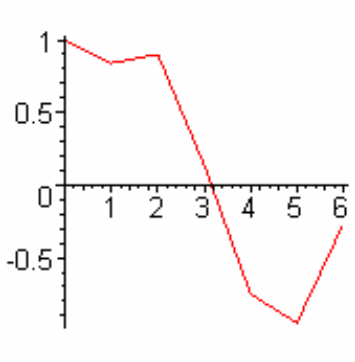
*NOTA: En el nombre del archivo hay que respetar las mayúsculas y minúsculas.

Vamos a dibujar ahora la gráfica de los puntos de la tercera columna respecto a la primera:

```
> convert(L[[1..-1],[1,3]],listlist);
```

$$[[0, 1], [1, 0.84], [2, 0.90], [3, 0.14], [4, -0.75], [5, -0.95], [6, -0.27]]$$

```
> plot(%);
```



Cuando queremos importar los datos desde Excel, primero habrá que guardar la hoja como tipo "Texto(delimitado por tabulaciones)(*.txt)" (en Archivo-Guardar Como). Entonces ya podremos utilizar ImportMatrix con el delimiter="\t" (no hace falta ponerlo, ya que es el valor por defecto). Veamos un ejemplo en el que tenemos en Excel 1000 valores aleatorios entre 0 y 1. El archivo se llama "prueba1.txt"

1	0,07442726
2	0,37153682
3	0,82365953
4	0,61700563
5	0,9145103
6	0,24780605
7	0,53141599

Vamos a calcular algunos datos estadísticos:

```
> L:=ImportMatrix("c:\\Datos\\prueba1.txt");
```

$$L := \begin{bmatrix} 1000 \text{ x } 2 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

Para ver si tenemos datos dentro de la matriz:

```
> L[100,2];
```

$$0.289227191$$

```
> lista:=convert(L[1..-1,2],list):
> stats[describe,mean](lista);
```

$$0.4861480568$$

```
> stats[describe,median](lista);
```

$$0.4718874265$$

```
> stats[describe,standarddeviation](lista);
```

0.2861087066

*NOTA: Si tenemos configurado Windows con la notación española, en Excel para los decimales utilizará la coma, y entonces en Maple no entenderá los números (1,34 lo cogerá como 2 números, 1 y 34). Por eso habrá que entrar en el panel de control, y en la configuración regional y de idioma habrá que cambiar esta opción.

Maple tampoco podrá importar matrices desde Word, por tanto aquí también tendremos que guardar con la extensión *.txt, y el delimitador será el espacio (" ").

5.9.2. Leer comandos desde un archivo

Se utiliza el comando ***read* "filename".** Filename se escribirá igual que en el apartado anterior. Veamos un ejemplo:

Escribimos en el bloc de notas lo siguiente:

```
with(LinearAlgebra):
L:=Matrix(3,3,1);
A:=IdentityMatrix(3);
evalm(L&*A);
```

```
> read "c:\\Datos\\prueba4.txt";
```

$$L := \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Si queremos que en Maple también aparezca el código, antes de introducir la sentencia 'read' habrá que introducir:

```
> interface(echo=2):
> read "c:\\Datos\\prueba4.txt";
> with(LinearAlgebra):
> L:=Matrix(3,3,1);
```

$$L := \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
> A:=IdentityMatrix(3);
```

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> evalm(L&*A);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

*NOTA: Como con la función ImportMatrix, no se puede leer un documento de Word. Por tanto, si el código estuviese en Word, habría que guardarlo con la extensión *.txt.

5.9.3. Escribir datos en un archivo

Mediante la función ***ExportMatrix*(“filename”,data)** podemos exportar una matriz a un archivo, siendo *filename* el lugar donde queremos guardar la matriz, y *data* la matriz. Si los datos que tenemos no están en una matriz podremos convertirla en matriz mediante mecanismos ya vistos. La función devuelve el número de bytes que se han escrito. En cada archivo se puede escribir una sola matriz, es decir, si mandamos que se escriban dos matrices en el mismo fichero se observa que la segunda matriz se sobreescribirá sobre la anterior. Si el archivo no existe, lo crea. Veamos un ejemplo:

```
> L:=LinearAlgebra[RandomMatrix](5);
```

```
> ExportMatrix("c:\\Datos\\matrixdata.txt",L);
```

$$L := \begin{bmatrix} -66 & -65 & 86 & 20 & -90 & 30 \\ 55 & 5 & -7 & -21 & 62 \\ 68 & 66 & 16 & -56 & -79 \\ 26 & -36 & -34 & -8 & -71 \\ 13 & -41 & -62 & -50 & 28 \end{bmatrix}$$

```
> A:=LinearAlgebra[IdentityMatrix](5);
```

$$A := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
> ExportMatrix("c:\\Datos\\matrixdata.txt",A);
```

50

Si abrimos ahora “matrixdata.txt” veremos que ha escrito la matriz A:

```
1       0       0       0       0
0       1       0       0       0
0       0       1       0       0
0       0       0       1       0
0       0       0       0       1
```

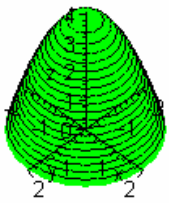
5.10. EJEMPLOS GENERALES

Ejemplo 1

Calcular el centroide de la región W acotada, limitada por el paraboloide z=4-x^2-y^2 y el plano z=0.

Utilizaremos coordenadas cilíndricas. Z=4-r^2. Vamos a dibujarlo utilizando la representación paramétrica, y le llamamos t=r y u=theta:

```
> with(plots):
> cylinderplot([t,u,4-t^2],u=0..2*Pi,t=0..2,color=green);
```



Vamos a calcular cuál es el jacobiano en cilíndricas:

```
> Jacobian([r*cos(theta), r*sin(theta),z], [r,theta,z],
output=determinant);
```

$$\cos(\theta)^2 r + \sin(\theta)^2 r$$

Vemos que el jacobiano es 'r'.

Ahora calculamos el volumen:

```
> V:=MultiInt(r,theta=0..2*Pi,z=0..(4-
r^2),r=0..2,output=integral)=MultiInt(r,theta=0..2*Pi,z=0..(4-
r^2),r=0..2);
```

$$V := \int_0^2 \int_0^{4-r^2} \int_0^{2\pi} r \, d\theta \, dz \, dr = 8\,\pi$$

* NOTA: Hay que tener cuidado de no poner el rango de 'r' antes del rango de 'z', ya que si no el resultado quedará en función de 'r'.

Ahora calculamos el momento estático respecto al plano XY:

```
> Mxy:=MultiInt(r*z,theta=0..2*Pi,z=0..(4-
r^2),r=0..2,output=integral)=MultiInt(r*z,theta=0..2*Pi,z=0..(4-
r^2),r=0..2);
```

$$Mxy := \int_0^2 \int_0^{4-r^2} \int_0^{2\pi} r\,z \, d\theta \, dz \, dr = \frac{32\,\pi}{3}$$

Dividiendo M_{xy}/V obtenemos la coordenada z del centroide:

$z_G = 4/3$

Ejemplo 2

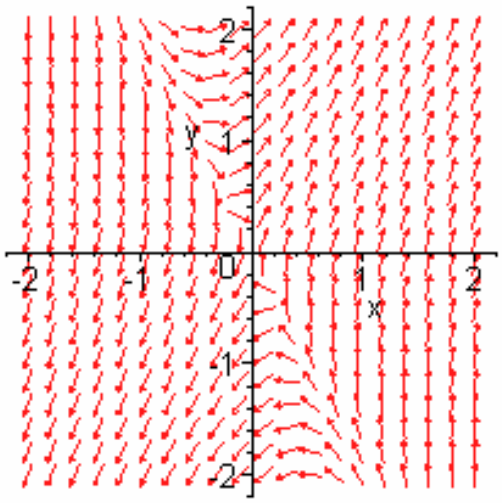
Resolver el siguiente sistema autónomo gráficamente y analíticamente:

x'=x+y
y'=4x+y

Primero dibujaremos la representación en el plano de fases:

```
> with(DEtools):

>dfieldplot([diff(x(t),t)=x(t)+y(t),diff(y(t),t)=4*x(t)+y(t)],[x(t)
,y(t)],t=-infinity..infinity,x=-2..2,y=-2..2,arrows=medium);
```



Ahora calcularemos la solución analíticamente:

```
> with(LinearAlgebra):
> A:=Matrix([[1,1],[4,1]]);
```

$$A := \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix}$$

```
> (Val,Vec):=Eigenvectors(A);
```

$$Val, Vec := \begin{bmatrix} 3 \\ -1 \end{bmatrix}, \begin{bmatrix} \frac{1}{2} & \frac{-1}{2} \\ 1 & 1 \end{bmatrix}$$

```
>Sol:=C1*exp(Val[1]*t)*Vec[1..2,1]+C2*exp(Val[2]*t)*Vec[1..2,2];
```

$$Sol := \begin{bmatrix} \frac{1}{2} C1\, \mathbf{e}^{(3t)} - \frac{1}{2} C2\, \mathbf{e}^{(-t)} \\ C1\, \mathbf{e}^{(3t)} + C2\, \mathbf{e}^{(-t)} \end{bmatrix}$$

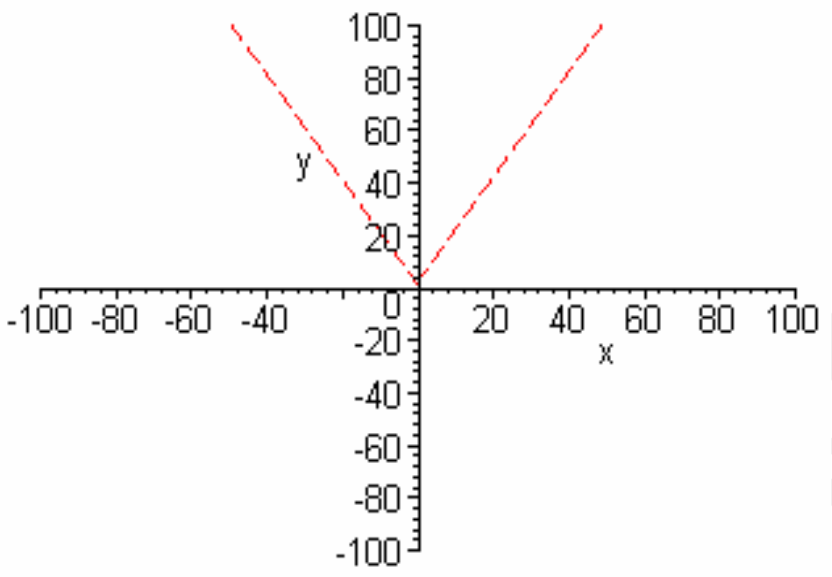
Veamos un caso particular dibujado:

```
> C1:=1;C2:=1;
```

$$C1 := 1$$

$$C2 := 1$$

```
> plot([Sol[1],Sol[2],t=-200..200],x=-100..100,y=-100..100);
```



6- INTRODUCCIÓN A LA PROGRAMACIÓN CON MAPLE 9.5

Lo que se pretende en este apartado es que el lector sea capaz de empezar a programar con una dedicación mínima de tiempo. Lo que aquí se explica es el lenguaje en sí, no algoritmia. Como se ha mencionado anteriormente, esta sección es una introducción a la programación por lo que lo que aquí se expone es muy básico, como se puede observar en los ejemplos. Estos tienen la misión de acelerar el proceso de aprendizaje.

Se recomienda que el lector domine la sintaxis de las sentencias básicas de programación antes de pasar al apartado de procedimientos.

6.1. SENTENCIAS BÁSICAS DE PROGRAMACIÓN

6.1.1. La sentencia *if*

Se trata de la sentencia de condición y selección más utilizada. La forma más general de utilizarla es la siguiente:

```
if (expresión de condición 1) then
        <sentencia1>;
elif (expresión de condición 2) then
        <sentencia2> ;
else
        <sentencia3>;
end if;
```

Los comandos ***elif*** indican otra condición para que se cumpla *sentencia2*, se puede utilizar tantas veces como se desee. ***else*** ejecutará la *sentencia3* en caso de que no se cumpla ninguna de las expresiones de condición. Una sentencia tipo ***if*** siempre terminará con un ***end if***. Si no se desean utilizar ***elif*** ni ***else***, basta con no incluirlos en la sentencia ***if***, su uso no es necesario. Por otra parte, si se desea se puede utilizar la sentenecia ***else if*** en lugar de ***elif***, en cuyo caso ha de concluirse con ***end if***.

Se debe tener en cuenta que la expresión de condición tiene que devolver siempre uno de los tres valores booleanos, es decir true, false o FAIL. En caso contrario el programa mandará un mensaje de error. Véase el siguiente ejemplo y pruébense distintos valores de a:

```
> a:=1;
> if (a>1) then;
>    print("a es mayor que uno");
> elif (a<1) then;
>    print("a es menor que uno");
> else
>    print("a es igual a uno");
> end if;
```

6.1.2. El bucle *for*

Esta sentencia nos servirá para realizar repeticiones. El bucle termina con **end do**. Forma más general:

for <**var**> from <**inicio**> by <**paso**> to <**final**> while (**expresión de condición**)
do <**sentencias**> end do;

Las cláusulas for **<var>,** from **<inicio>,** by **<paso>,** to **<final>**, while **<expresión de condición>,** son opcionales. Si no se les atribuye ningún valor, tomarán por defecto los mostrados en la siguiente tabla:

Cláusula	Valor por defecto
for	Variable auxiliar
from	1
by	1
to	infinito
while	true

Por ser opcionales podemos simplificar esta estructura de diversas formas y la sintaxis se muestra a continuación:

for <**var**> from <**expresión**> by <expresion> to <expresion> do
<**sentencias**>**;**
end do;

Ejemplo

```
> for i from 0 to 5 do
>    i^2;
> end do;
```

0

1

4

9

16

25

Se recomienda probar diferentes opciones (como cambiar “to 5” por “while i<5”) y observar los resultados.

Otra opción interesante del bucle for es que la expresión puede ser un vector, y *var* puede ser cada elemento del mismo. Por ejemplo:

```
> restart;
> L:=[seq(1/i,i=1..4)];
```

$$L := \left[1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right]$$

```
> for i in L do;
> evalf(exp(i));
```

```
> end do;
```

2.718281828

1.648721271

1.395612425

1.284025417

Como se puede observar, en este caso, la variable *k* representa cada elemento del vector *L*, y en caso de que se cumpla la condición impuesta, se imprimirán en pantalla. El comando seq se verá en el siguiente apartado.

6.1.3. El bucle *while*

Es utilizado para ejecutar repetidamente una secuencia de sentencias mientras una condición se cumpla. La estructura es la siguiente:

while (expresión de condición) do
<sentencias>;
end do;

La expresión de condición tiene que devolver siempre un valor booleano ***true***, ***false*** o **FAIL**. Ejemplo:

```
> restart;
> x:=256;
> while (x>1) do;
>    x:= x/4;
> end do;
```

$x := 64$

$x := 16$

$x := 4$

$x := 1$

Se recomienda al lector que teclee estos comandos o variantes de los mismos en Maple y observar los resultados.

6.1.4. La sentencia *break*

Una vez ejecutada esta sentencia, su resultado es la parada y salida directa durante el tiempo de ejecución de una estructura tipo ***for/while/do***. Es decir, con esta sentencia se logra salir de un bucle determinado antes de que lo termine. Después de salir, el programa continuará en la siguiente línea después del bucle o repetición en la que hemos introducido el ***break*****.** Ejemplo:

```
> restart;
> for i from 0 to 20 do;
>   if (i=6) then;
>       break;
>    end if;
```

```
>    i^2;
> end do;
```

0

1

4

9

16

25

6.1.5. La sentencia *next*

Cuando Maple ejecuta una sentencia ***next*** dentro de un bucle ***for/while/do***, salta directamente a la siguiente iteración sin realizar ninguna sentencia, pero sin salir del bucle, a diferencia del ***break***. Ejemplo:

```
> for i from 0 to 3 do;
   if (i=2) then;
       next;
    end if;
    i^2;
end do;
```

0

1

9

Compárese este ejemplo con el anterior.

6.1.6. Comandos para realizar repeticiones

Antes de empezar a describir uno por uno los comandos, que se juzgan más interesantes, tiene que quedar claro que son diferentes a los bucles explicados anteriormente. Algunos procesos que requieren repeticiones son tan comunes que Maple dispone comandos especiales para ellos. No es el usuario quien establece lo que se ha de realizar dentro de cada repetición, sino que viene definiido por el propio comando. Puede parecer un poco confuso, pero los ejemplos ayudarán al lector.

6.1.6.1 Comando Map

El comando ***map*** aplica una función o procedimiento a todos los elementos que pertenecen a un objeto (lista, vector, matriz, etc.). La forma más simple es:

map(función, objeto);

Un ejemplo sencillo de aplicación sería el siguiente:

```
> L:=[-1,2,-3,-4,5];
  q:=map(abs, L);
  map(x->x^2,L);
```

$$L := [-1, 2, -3, -4, 5]$$

$$q := [1, 2, 3, 4, 5]$$

$$[1, 4, 9, 16, 25]$$

Otro ejemplo:

```
> L:= [seq(Pi/i,i=1..5)];
```

$$L := \left[\pi, \frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{5}\right]$$

```
> q:= map(sin,L);
```

$$q := \left[0, 1, \frac{\sqrt{3}}{2}, \frac{\sqrt{2}}{2}, \sin\left(\frac{\pi}{5}\right)\right]$$

Este comando se considera perteneciente a los comandos de control de flujo, debido a que realiza un bucle recorriendo cada elemento del objeto. Por supuesto, este comando tiene muchas más utilidades que aquí no se mostrarán.

- ***Select, remove y selectremove:***

  El comando ***select*** devuelve los operandos que evaluados de forma booleana devuelven ***true***. Ejemplo:

```
> L:= [1,2,3,4,5,6,7,8];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8]$$

```
> Primos:= select(isprime,L);
```

$$Primos := [2, 3, 5, 7]$$

De esta manera obtenemos los números primos de la lista L.

El comando ***remove*** devuelve los operandos que evaluados de forma booleana devuelven ***false***. Ejemplo:

```
> L:= [1,2,3,4,5,6,7,8];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8]$$

```
> No_Primos:= remove(isprime,L);
```

$$No\_Primos := [1, 4, 6, 8]$$

Ahora hemos obtenido los números no primos de la lista L.

Y el comando ***selectremove*** devuelve dos objetos, uno con los mismos operandos recibidos con el comando ***select*** y el otro con los del comando ***remove***.

- ***Los comandos*** seq; add y sum; mul y product***:***

  Estos comandos forman secuencias, sumas y multiplicaciones respectivamente. Las estructuras son las siguientes:

seq(f, i= a..b);

Ejemplo de seq:

```
> L:= [seq(i^2,i=0..4)];
```

$$L := [0, 1, 4, 9, 16]$$

```
> L[3];
```

$$4$$

De esta forma formamos una lista de 5 elementos y para acceder a uno de ellos utilizamos la notación indicada. Ahora planteamos un ejemplo más interesante que dibuja una función senoidal mediante rectas (aproximado):

```
> p:=[seq(k,k=0..12)];
> y :=[seq(sin(k),k=0..12)];
```

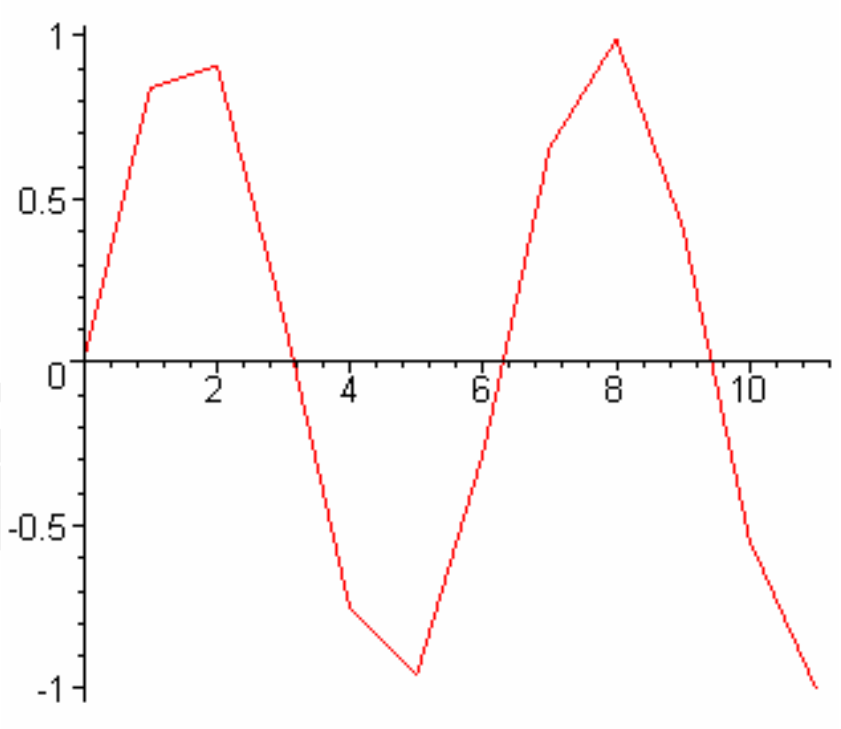
$$p := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

$$y := [0, \sin(1), \sin(2), \sin(3), \sin(4), \sin(5), \sin(6), \sin(7), \sin(8), \sin(9), \sin(10), \sin(11), \sin(12)]$$

```
> s :=[seq([p[k],y[k]],k=1..12)];  #secuencia de binomios (puntos)
```

$$s := [[0, 0], [1, \sin(1)], [2, \sin(2)], [3, \sin(3)], [4, \sin(4)], [5, \sin(5)], [6, \sin(6)], [7, \sin(7)], [8, \sin(8)], [9, \sin(9)], [10, \sin(10)], [11, \sin(11)]]$$

```
> plot(s);
```



Pruébese a escribir ahora

```
> p[0];
```

Esta es la causa por la cual en la lista s el contador k empieza desde 1 y no desde 0.

Para sumar:

add(f, i= a..b); sum(f, i= a..N);

En un principio estos dos comandos pueden parecer iguales pero no lo son, existen importantes diferencias:

La primera es que a y b han de ser constantes numéricas y en cambio N puede no tener ningún valor asignado.

La segunda es el modo en que operan: add suma término a término y en cambio sum lo hace de manera simbólica lo que al programar puede hacer que nuestros algoritmos no funcionen o por el contrario sean más eficientes (menor tiempo de ejecución).

Ejemplos

```
> add(i,i=0..5);
```

$$15$$

```
> assume(abs(r)<1);
> sum(r^(n-1),n=1..N);   #suma simbólica de la serie geométrica
```

$$\frac{r\sim^{(N+1)}}{r\sim\,(r\sim-1)}-\frac{1}{r\sim-1}$$

Para multiplicar:

mul(f, i= a..b); product(f, i= a..N);

El concepto y estructura de estos dos comandos para multiplicar es el mismo que el de add y sum, sólo que en vez de sumar se multiplica: mul es el equivalente a add y product lo es a sum. Se omiten ejemplos por similitud con los anteriores.

6.2. PROCEDIMIENTOS CON MAPLE

Un procedimiento es un conjunto de sentencias agrupadas y enfocadas a realizar una tarea específica. Es lo que por ejemplo en C++ se llamaría función o subrutina. Definición de procedimiento

La estructura general de un procedimiento de Maple es la siguiente:

Nombre_del_procedimiento := proc(*P*)
local *L*; (declaración de variables locales)
global *G*; (declaración de variables globales)
options *O*; (opciones)
description *D*; (descripción)
cuerpo del procedimiento
end proc;

- No son necesarias todas estas especificaciones pero este orden es inalterable.
- La letra P representa a los parámetros que se le pasan al procedimiento.

Generalmente se debe nombrar el procedimiento utilizando el nombre con el que se le va a invocar. Para ejecutarlo o invocarlo se utiliza la llamada al procedimiento, que tiene esta forma:

Nombre_del_procedimiento(A);

El nombre del procedimiento que vamos a invocar es *Nombre_del_procedimiento* y la letra A representa los parámetros que se van a utilizar en dicho procedimiento. Lo que

devuelve es lo que aparece en la sentencia anterior a end proc (ver return). Ejemplo sencillo de procedimiento para calcular el cubo de un número cualquiera:

```
> Cubo_Proc := proc(a)
>    a^3;
> end proc;
```

Cubo_Proc := **proc**(*a*) *a*^3 **end proc**

```
> Cubo_Proc(4);     # invocamos el procedimiento
```

64

Obsérvese que no hemos especificado tipos de variables, opciones y descripción. La estructura mostrada arriba es completamente general.

6.2.1. Componentes de un procedimiento

6.2.1.1 Parámetros

Se puede escribir un procedimiento que sólo funcione con un tipo determinado de parámetros. En este caso es interesante indicarlo en la descripción del procedimiento de forma que si se intenta pasar otro tipo de parámetros, Maple envíe un mensaje de error informativo. La declaración sería de la forma:

`parameter :: tipo`

donde `parameter` es el nombre del parámetro y *tipo*, el tipo que aceptará.

Ejemplo

```
> N::integer
```

Cuando se llama al procedimiento, antes de ejecutar el cuerpo, Maple examina los tipos de los parámetros actuales y solamente si todo es correcto, se ejecuta el resto.

La llamada a un procedimiento se realiza de igual forma que la de una función

```
> F(A);
```

Como ejemplo vamos a realizar un procedimiento que calcule el factorial de un número entero:

```
> Factorial_Proc := proc(N::posint)    # N es un entero positivo
>    mul(i,i=1..N);      # ¿Por qué se puede escribir N?
> end proc;
```

Factorial_Proc := **proc**(*N*::*posint*) mul(*i*, *i* = 1 .. *N*) **end proc**

```
> Factorial_Proc(5);      # se invoca de la misma manera
```

120

6.2.1.2 Variables locales y variables globales

En un procedimiento pueden existir tanto variables locales como globales. Fuera de éste las variables siempre serán globales. Existen dos diferencias principales entre variables locales y variables globales:

Dentro de un procedimiento se puede cambiar el valor de una variable local, sin afectar a una variable global con el mismo nombre y/o a una variable local de otro procedimiento.

Se recomienda declarar el carácter de las variables explícitamente. Por defecto Maple declara las variables como locales:

- Si aparece a la izquierda de una sentencia de asignación.

  `A:=` ó `A[i]:=`

- Si aparece como la variable índice de un bucle `for`, o en un comando `seq`, `add`, sum, mul o product.

Si no se cumple ninguno de estos dos puntos, la variable se convertirá en global.

La otra diferencia entre las variables locales y globales es el nivel de evaluación. Durante la ejecución de un procedimiento, las variables locales se evalúan sólo un nivel, mientras que las globales lo hacen totalmente.

Ejemplo

```
> f:=x+y:
> x:=z^2:
> z:=y^3+1:
```

Todas las variables son globales, así que se evaluará totalmente, es decir, se ejecutarán todas las asignaciones realizadas para dar la expresión de f.

```
> f;
```

$$(y^3+1)^2+y$$

Se puede controlar el nivel de evaluación utilizando el comando `eval`.

```
> eval(f,1);    (Sólo ejecuta la primera asignación)
```

$$x+y$$

```
> eval(f,2);    (Sólo ejecuta la primera y la segunda asignación)
```

$$z^2+y$$

```
> eval(f,3);    (Ejecuta las tres asignaciones)
```

$$(y^3+1)^2+y$$

Así se puede conseguir que una variable local dentro de un procedimiento se evalúe totalmente, aunque no afecta demasiado al comportamiento del programa.

```
> F:=proc()
>  local x, y, z;
>  x:= y^2;  y:= z^2;  z:=3;
>  eval(x);
> end:
> F();
```

Sin la llamada a `eval` el resultado hubiese sido y^2

NOTA: Para obtener resultados numéricos debe tenerse en cuenta el tipo de variables que se utiliza. Es importante distinguir cuándo se está trabajando con números reales y cuándo con enteros. Cuando se trabaja con números reales, Maple realiza todas las operaciones necesarias para llegar al resultado numérico aproximado, que depende del número de cifras significativas que se estén empleando. Cuando se trabaja con números enteros, las operaciones son lentas y a menudo hacen que el programa se bloquee. Esto se debe a que Maple opera simbólicamente, manejando todas las expresiones exactamente, sin sustituir valores ni realizar operaciones numéricas que no sean exactas. Esto hace que la cantidad de memoria que maneja el programa en estos cálculos sea mucho mayor que si se sustituyen las expresiones por valores numéricos y se opera con ellos directamente, como sucede cuando se opera con números reales.

Ejemplo

```
> sin(3/4);

> sin(3./4.);
```

Nota importante: Evalf vs evalhf: De cara a aumentar la velocidad de ejecución Maple ofrece la posibilidad de utilizar el hardware para realizar cálculos. Dependiendo de la capacidad del ordenador se pueden ejecutar operaciones a velocidades muy altas. Esto tiene el inconveniente de que no se puede determinar el número de cifras significativas de la salida, ya que no depende de Maple sino de la capacidad del procesador. Para operar de este modo se utiliza el comando evalhf en lugar de evalf. Normalmente es más que suficiente la precisión que ofrece el hardware para realizar cálculos, por lo que se recomienda vivamente utilizar el comando evalhf cuando se hayan de realizar un número significativo de operaciones.

6.2.1.3 Options

Las opciones de un procedimiento deben aparecer inmediatamente después de la declaración de variables. Un procedimiento puede tener una o varias opciones que ofrece Maple:

`Options` *`O1, O2, ..., On`*

Opciones remember y system

Comenzamos con un ejemplo: considérese la secuencia de Fibonacci donde cada número es la suma de los dos anteriores (siendo los dos primeros 0 y 1: 0, 1, 1, 2, 3, 5, 8, 13...). Este sería el procedimiento que calcula el número enésimo de la secuencia:

```
> fibonacci := proc(n::nonnegint)  #entero no negativo
>    if(n<2) then;
>       n;
>    else
>       fibonacci(n-1)+fibonacci(n-2);
>    end if;
```

```
> end proc;
```

fibonacci := **proc**(*n*::*nonnegint*)
if $n < 2$ **then** *n* **else** fibonacci($n-1$) + fibonacci($n-2$) **end if**
end proc

```
> fibonacci(5);
```

5

Se observa que si escribimos fibonacci(25) tarda un tiempo considerable en calcularlo, y con 50 tarda más de 30 minutos. Esto es así porque recalcula todos los sumandos una y otra vez. Para evitar esto se utiliza la opción remember que almacena los valores calculados en el procedimiento en una remember table y si se requieren otra vez, los toma de esta. Mismo ejemplo con dicha opción:

```
> fibonacci := proc(n::nonnegint)  #entero no negativo
>    option remember,system;
>    if(n<2) then;
>       n;
>    else
>       fibonacci(n-1)+fibonacci(n-2);
>    end if;
> end proc;
```

fibonacci := **proc**(*n*::*nonnegint*)
option *remember*, *system*;
if $n < 2$ **then** *n* **else** fibonacci($n-1$) + fibonacci($n-2$) **end if**
end proc

```
> fibonacci(50);
```

12586269025

La opción `system` permite a Maple borrar resultados anteriores de una remember table. Estas dos opciones se utilizan conjuntamente.

Opción Copyright

Maple considera cualquier opción que comienza con la palabra Copyright como una opción Copyright. Maple no imprime el cuerpo de estos procesos (a menos que se especifique lo contrario con el comando interface(verboseproc=2);). Ejemplo;

```
> f:=proc(expr::anything, x::name)
> option `Copyright 1684 by G.W. Leibniz`;
> Diff(expr,x);
> end;
```

6.2.1.4 El campo de descripción

Es la última claúsula de un procedimiento y debe aparecer justo antes del cuerpo. No tiene ningún efecto en la ejecución del procedimiento, su único objetivo es informar. Maple lo imprime aunque no se imprima el procedimiento debido a la opción Copyright.

Ejemplo

```
> f := proc(x)
>   option `Copyright Tecnun`;
>   description "calcular el cuadrado de x";
>   x^2;
> end proc;
```

f := **proc**(*x*) **descriptio** "calcular el cuadrado de x" ... **end proc**

6.2.2. Valor de retorno

Cuando se invoca un procedimiento, el valor que Maple devuelve es normalmente el valor de la última sentencia del cuerpo del proceso. Pueden existir otros tres tipos de valor de retorno en un procedimiento:

1. A través de un parámetro. (no recomendable y por ello no se explica)
2. A través de un return explícito.
3. Mediante un return de error.

6.2.2.1 Return explícito

Un *return explícito* ocurre cuando se llama al comando `RETURN` , que tiene la siguiente sintaxis:

`RETURN (`*`secuencia`*`);`

Este comando causa una respuesta inmediata del procedimiento, que es el valor de *secuencia*. Se vuelve al punto desde donde se ha invocado el procedimiento

Por ejemplo, el siguiente procedimiento determina la primera posición *i* del valor *x* en una lista de valores *L*. Si *x* no aparece en la lista *L*, el procedimiento devuelve un 0.

```
> f:=proc(x::anything, L::list)
> local i;
> for i to nops(L) do
> if x=L[i] then RETURN (i) fi;
> od;
> 0;
> end:

> posicion := proc(x::anything, L::list)
>    local i;
>    for i to nops(L) do;
>       if (x=L[i]) then;
>          RETURN (i);
>       else 0;
>       end if;
>    end do;
> end proc;
```

La comando `nops` calcula el número de operandos de una expresión.

6.2.2.2 Return de error

Un *return de error* ocurre cuando se llama al comando `ERROR` , que tiene la siguiente sintaxis:

```
error " texto "
```

Normalmente causa la salida del procedimiento a la sesión de Maple, donde se imprime un mensaje de error.

```
Error, (in nombre_del_procedimiento), secuencia
```

Texto es el argumento del comando `ERROR` y _ nombre_del_procedimiento el nombre del procedimiento donde se ha producido el error. Si el procedimiento no tiene nombre el mensaje será:

```
Error, (in unknown), secuencia
```

Ejemplo de aplicación

```
> cociente := proc(x, y)
>   if (y=0) then;
>     error "no se puede dividir por cero";
>   else
>     x/y;
>   end if;
> end proc;
```

cociente := **proc** (x, y) **if** y = 0 **then error** "no se puede dividir por cero" **else** x/y **end if end proc**

```
> cociente(3,0);
Error, (in cociente) no se puede dividir por cero
```

6.2.3. Guardar y recuperar procedimientos

Mientras se está desarrollando un procedimiento se puede salvar el trabajo grabando la hoja de Maple entera. Una vez que se está satisfecho con cómo funciona el procedimiento se puede guardar en un archivo ***.m** (sólo el procedimiento). Estos archivos forman parte del formato interno de Maple, lo hace que se pueda trabajar con ellos de manera más eficiente. Para grabarlos con esta extensión se utiliza el comando `save` y si lo que se quiere es recuperarlos, `read`. Ejemplo:

```
> Cubo_Proc := proc(a)
>    a^3;
> end proc;
> save Cubo_Proc, "Cubo_Proc.m";
> read "Cubo_Proc.m";
```

6.2.4. Procedimientos que devuelven procedimientos

Algunos de los comandos básicos de Maple devuelven procedimientos. Por ejemplo, ***rand*** devuelve un procedimiento en el cual se generan números enteros en un rango determinado.

Ejemplo

```
> f:= rand(4..7);
> seq(f(),i=1..20);
```

5, 6, 5, 7, 4, 6, 5, 4, 5, 5, 7, 7, 5, 4, 6, 5, 4, 5, 7, 5

La función ***dsolve*** con la opción *type=numeric* devuelve un procedimiento que estima numéricamente una ecuación diferencial.

Esta sección tratará sobre como pasar de un procedimiento externo a otro interno.

A continuación se realizará un ejemplo de procedimiento que devuelve un procedimiento según la función introducida. El método de iteraciones de Newton consiste en lo siguiente (no se explicará el método en sí, sólo cómo se programa):

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

El código para generar el procedimiento deberá aplicar la ecuación a la función introducida y de la siguiente manera:

```
> MakeIteration:=proc(expr::algebraic,x::name)
>     local iteration;
>     iteration:=x-expr/diff(expr,x);
>     unapply(iteration,x);
> end proc:
```

Para probarlo ponemos como valor inicial 2 y pedimos cuatro datos. El procedimiento devuelto se llamará Newton:

```
> expr:=x-2*sqrt(x);
```

$$expr := x - 2\sqrt{x}$$

```
> Newton:=MakeIteration(expr,x);
```

$$Newton := x \rightarrow x - \frac{x - 2\sqrt{x}}{1 - \frac{1}{\sqrt{x}}}$$

```
> x0:=2.0;
```

$$x0 := 2.0$$

```
> to 4 do x0:=Newton(x0); end do;
```

$$x0 := 4.828427124$$

$$x0 := 4.032533198$$

$$x0 := 4.000065353$$

$$x0 := 4.000000000$$

♦ ***El operador*** Shift:

Consideramos el problema de programar un procedimiento que utiliza una función como parámetro, y devuelve otra, como por ejemplo *g(x)=f(x+1)*, se puede escribir dicho procedimiento de la siguiente manera:

```
> shift := (f::procedure) -> (x->f(x+1)):
> shift(sin);
```

$$x \to \sin(x+1)$$

El ejemplo anterior trata de cómo funciona el operador ***shift*** con funciones de una sola variable, si se utilizan más Maple devolverá un mensaje de error. Para ello existe otra manera de programarlo, mediante la palabra ***args***, que es la secuencia actual de parámetros excepto el primero de ellos. Por ejemplo:

```
> h := (x,y) -> x*y;
```

$$h := (x, y) \to x\,y$$

```
> shift := (f::procedure) -> (x->f(x+1, args [2..-1])):
> hh := shift(h);
```

$$hh := x \to h(x+1, \mathrm{args}_{2\,..\,-1})$$

```
> hh(x,y);
```

$$(x+1)\,y$$

♦ ***Entrada interactiva de datos:***

Normalmente, los datos que se introducen en los procedimientos son parámetros. Algunas veces, en cambio, se necesita reclamar directamente los datos pertinentes al usuario del procedimiento. Los dos comandos principales para realizar esta tarea son ***readline*** y ***readstat.***

El comando ***readline*** lee cadenas de caracteres del teclado, su uso es muy sencillo y se muestra a continuación un ejemplo:

```
> s := readline(terminal);
> Maple;
```

Se pueden desarrollar pequeños programas como el que se muestra :

```
> DeterminarSigno := proc (a::algebraic)
>  local s;
>  printf("¿Es el signo de %a positivo? Responda si o no: ",a);
>  s := readline(terminal);
>  evalb( s="si" or s="s");
end proc:
> DeterminarSigno(u-1);
```

El comando ***readstat*** es similar al anterior, a diferencia de que este último lee expresiones y no variables tipo *string*. Su sintaxis se puede observar en el siguiente ejemplo:

```
> restart:
> readstat("Introduzca grado: ");
```

Introduzca grado: `n-1;`

$$n-1$$

Otra diferencia entre ***readline*** y ***readstat***, es que mientras que la primera tan sólo puede captar una línea, la segunda permite escribir una expresión a lo largo de varias líneas. Además, el comando ***readstat*** se re-ejecuta en caso de error.

NOTA: Si se desea pasar de una cadena de caracteres a una expresión, se puede utilizar el comando ***parse***:

```
> s:="a*x^2+1";
```

$$s := \text{"a*x^2+1"}$$

```
> y:=parse(s);
```

$$y := a\,x^2 + 1$$

6.2.5. Ejemplo de programación con procedimientos

Como ejemplo aclaratorio de la programación con procedimientos, se creará un procedimiento al cual se le pasarán los valores de los coeficientes de una ecuación cúbica y el programa devolverá un estudio detallado de las raíces de dicha ecuación además de obtener sus extremos relativos y clasificarlos debidamente en función de su condición de máximo o mínimo.

Para la realización de dicho programa se utilizarán las instrucciones y comandos introducidos en el presente manual, así como la potente herramienta de derivación de la que dispone Maple 9.5.

Véase a continuación el código del procedimiento, así como dos ejemplos prácticos de cómo funciona acompañados de una representación gráfica de la ecuación polinómica a la cual representan.

```
> raices := proc(a,b,c,d)
>    local pol,dpol,ddpol,raiz,maxx,minx,ext,num,i,plotinf,plotsup,
ddpolexpr;
>    pol :=  a*x^3 + b*x^2 + c*x + d;
>    dpol := diff(pol,x); ddpol := diff(dpol,x);
>    raiz := [fsolve(pol=0,x)];
>    print(`Las raices del polinomio son`,`raiz`);
>    ext := [fsolve(dpol=0,x)];
>    num := nops(ext);
>    ddpolexpr := unapply(ddpol,x);
>    i := 1;
>    if (num>0) then
>       while i<=num do
>          if (ddpolexpr(ext[i])>0) then
>             minx := ext[i];
>             print (`El polinomio tiene un mínimo en` ,`minx`);
>          elif (ddpolexpr(ext[i])<0) then
```

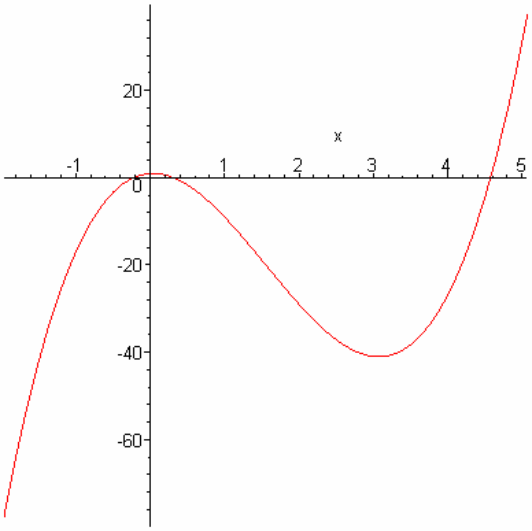
```
>            maxx := ext[i];
>            print (`El polinomio tiene un máximo en`,`maxx`);
>         end if;
>      i := i + 1;
>      end do;
>      plotinf:=ext[1]-2; plotsup:=ext[num]+2;
>      plot(pol,x=plotinf..plotsup);
>   else
>      print(`El polinomio no tiene ni máximo ni mínimo`);
>      plotinf:=raiz[1]-2; plotsup:=raiz[1]+2;
>      plot(pol,x=plotinf..plotsup);
>   end if;
> end proc;

> raices(3,-14,1,1);
```

Las raices del polinomio son, [-0.2291013282, 0.3178192610, 4.577948734]

El polinomio tiene un máximo en, 0.03613396320

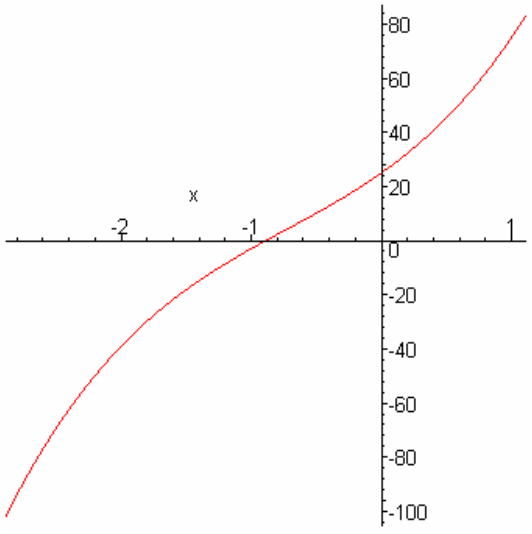
El polinomio tiene un mínimo en, 3.074977148




```
> raices(5.,11.,34., 25.);
```

Las raices del polinomio son, [-0.8872715475]

El polinomio no tiene ni máximo ni mínimo



Se observa fácilmente que en el primero de los casos la ecuación tiene tres raíces, así como dos extremos relativos; la segunda ecuación cúbica, en cambio, tiene una raíz de multiplicidad tres y carece de extremo relativo alguno.

6.3. PROGRAMACIÓN CON MÓDULOS (PROGRAMACIÓN AVANZADA)

Los procedimientos permiten asociar una secuencia de comandos con un simple comando. Igualmente, los módulos nos permiten asociar procedimientos y datos externos al Maple. Se podría definir un módulo como un procedimiento que devuelve algunas de sus variables locales.

Este apartado tratará de describir dichos módulos, éstos son un tipo de expresión de Maple (como números, ecuaciones y procedimientos), que permiten crear algoritmos genéricos, paquetes, o utilizar sentencias del tipo Pascal. El uso de módulos satisface cuatro conceptos principales de la ingeniería de software:

- **Encapsulación:** Garantiza que una abstracción es utilizada de acuerdo a un interface especificado. Se pueden escribir sistemas de software que se pueden transportar y reutilizar en otros programas o aplicaciones. Esto hace el código más sencillo de mantener y entender.
- **Paquetes:** Son el vehículo para poder agrupar procedimientos de Maple que tienen relación con un mismo problema.
- **Modelado de objetos:** Los objetos son fácilmente representados utilizando módulos. En la ingeniería de software o programación orientada a objetos (POO), un objeto es definido como algo que tiene un estado y un comportamiento. Se puede programar con objetos mandándoles mensajes, los cuales responden desarrollando un servicio.
- **Programación genérica:** Acepta objetos que poseen unas propiedades específicas o comportamientos.

Un pequeño ejemplo para comprender la estructura básica de un módulo es el siguiente (obsérvese que la estructura es muy parecida a un procedimiento):

```
> TempGenerator := module()
>    description "Generador de símbolos temporales";
>    export gentemp;            # ¡Diferencia fundamental!
>    local count;
>    count:=0;
>    gentemp := proc()          # procedimiento dentro del módulo
>         count := 1+count;
>         (T||count)            #Concatena caracteres
>         end proc;
> end module;
```

Nótese que no hay return explícito

Módulos vs Procedimientos*:*

La principal diferencia en el ejemplo propuesto es que si en lugar de hacerlo con un módulo, lo hiciéramos con procedimientos no deberíamos usar una declaración de variable tipo *export*, esto significa que es utilizable fuera de la estructura donde fue creada. Las variables tipo export son unas variables locales pero accecibles desde el exterior del módulo. La sintaxis para acceder a este tipo de variables es diferente. Por ejemplo, para acceder a la variable *gentemp* exportada, se haría así:

```
> TempGenerator:-gentemp();
```

T1

Para crear las exportaciones se utiliza la sentencia ***use*** de la siguiente manera:

```
> use TempGenerator in
>    gentemp();
>    gentemp();
>    gentemp();
> end use;
```

T1

T2

T3

Como se puede observar la sentencia use permite acceder a las variables export directamente, sin tener que escribir el nombre del módulo.

6.3.1. Sintaxis y semántica de los módulos

La sintaxis de un módulo es muy similar a la de un procedimiento. La estructura más general tiene el siguiente aspecto:

```
Nombre_del_módulo:= module()  ¡No hay argumentos explícitos!
    local L;
    export E;
    global G;
    options O;
    description D;
    Cuerpo del módulo (puede incluir procedimientos...);
end module;
```

6.3.1.1 Parámetros de los módulos

A diferencia de los procedimientos los módulos no tienen parámetros explícitos porque no se invocan con argumentos.

La definición de parámetros en los módulos es similar a los de los procedimientos, en cambio, todos los módulos poseen un parámetro llamado ***thismodule***. A pesar del cuerpo del módulo, este parámetro especial evalúa el módulo que se utiliza.

Todas las definiciones de módulos llevan implícitas las definiciones de los parámetros procname, args y nargs. Las definiciones sobre el módulo no pueden dar referencia a éstos parámetros implícitos. La diferencia entre thismodule y procname es

que procname evalúa el nombre del procedimiento mientras que thismodule evalúa la expresión contenida en un módulo. Esto es debido a que los procedimientos suelen ejecutarse por medio de su nombre, que es conocido, mientras que en el caso de los módulos, no es necesario.

6.3.1.2 Declaraciones

La sección de declaraciones de un módulo debe aparecer inmediatamente después del paréntesis. Todas las sentencias en la sección de declaraciones son opcionales. La mayoría de las declaraciones de los módulos son similares a las de los procedimientos.

Un ejemplo de declaración podría ser la sentencia description, que da información sobre la tarea que realiza el módulo. Se utiliza de la siguiente manera:

```
> Hello := module()
>      description "Mi primer módulo";
>      export say;
>      say := proc()
>         print("HELLO WORLD")
>      end proc;
> end module:
> eval(Hello);
```

```
module() export say; description "Mi primer módulo"; end module
```

La declaración ***export*** será descrita más adelante. Se recomienda declarar todas las variables utilizadas en el módulo para evitar errores. Tipos de variables:

Las variables sobre las cuales se hace una referencia sin una definición dentro del módulo, se declararán como *globales*. Siguiendo a la palabra clave global va una cadena de caracteres o uno o más símbolos. En algunos casos concretos es recomendable utilizar variables globales para prevenirse de las normas sobre las variables locales de los módulos, que son más restrictivas.

Para definir variables *locales*, se utiliza la declaración local. Su formato es el mismo que para los procedimientos. Las variables locales no son visibles fuera de la definición del módulo en el que se utilizan, son privadas. Este tipo de variables son usualmente de corto uso, su vida dura el tiempo de ejecución del módulo en cuestión.

Las variables locales permiten ser declaradas como exportadas mediante la palabra *export* seguida del nombre de la variable a exportar. Nunca se puede declarar una varible tipo export implícitamente. La principal diferencia entre estas variables y las locales, es que a éstas podemos acceder después de que sean creadas. Para acceder a una exportación de un módulo se utiliza el operador :-, un ejemplo ligado al anterior citado sería el siguiente:

```
> Hello:-say();
```

6.3.1.3 Opciones de los módulos

Como en los procedimientos, la definición de los módulos puede contener opciones. En los módulos, las opciones disponibles son diferentes a las de los procedimientos. Tan sólo la opción trace y Copyright (descrita en el apartado de procedimientos) son

comunes a ambas estructuras. Las siguientes opciones tienen un significado predefinido en los módulos: load, unload, package, y record.

Las opciones load y unload: La opción de inicialización de un módulo es load=nombre_procedimiento, donde nombre_procediemiento es el nombre de un procedimiento en la declaración local o exportada del módulo. Si se utiliza esta opción, entonces el procedimiento es llamado cuando se lee el módulo en el que está contenido. La opción unload=nombrep, especifica el nombre del procedimiento local o exportado que se ejecuta cuando el módulo es destruido. Estas dos opciones pueden tener relación con el constructor/destructor de clases de C++.

La opción package: Los módulos con esta opción representan un paquete de Maple. La exportación de un módulo con esta opción es automáticamente protegido.

La opción record: Esta opción se utiliza para identificar grabaciones. Las grabaciones son producidas por el constructor Record y son representadas utilizando módulos.

6.4. ENTRADA Y SALIDA DE DATOS

A pesar de que Maple es un lenguaje orientado a la manipulación matemática, nos permite también operar con datos provenientes de una fuente externa, operar con datos provenientes de una fuente externa al Maple, o bien exportar datos calculados a otros programas, incluso introducir y mostrar datos directamente con un usuario. El software de Maple incluye muchos comandos de entrada y salida para facilitar estas labores.

6.4.1. Ejemplo introductorio

Esta sección pretende mostrar cómo se puede utilizar la librería de entrada y salida de Maple. En concreto, muestra cómo escribir datos numéricos en un archivo, y cómo leer los datos de un archivo externo. Considérense los siguientes datos:

```
> A := [[0,0],
>       [1, .345632347],
>       [2, .173094562],
>       [3, .026756885],
>       [4, .986547784],
>       [5, 1.00000000]]:
```

En este conjunto de datos, la agrupación se ha hecho por parejas xy, donde x representa un número entero mientras que y representa números reales.

Si estos datos se desean utilizar en cualquier otro programa será necesario guardarlos en un archivo externo, esto se puede hacer utilizando la librería I/O de la siguiente manera:

```
> for xy in A do fprintf("Data", "%d %e\n", xy[1], xy[2]) end do:
> fclose("Data");
```

El archivo Data ha sido guardado en el directorio actual de trabajo. Para determinar el directorio se puede utilizar el comando currentdir(). Se pueden visualizar los datos con cualquier editor de texto, como el notepad.

El comando fprintf guarda cada par de números en el archivo. Este comando requiere dos o más argumentos, el primero indica el archivo en el que se van a guardar los datos, el segundo las propiedades de los mismos, y los siguientes son los datos que se van a escribir.

En el ejemplo anterior hemos guardado el conjunto A en el archivo Data. Si este archivo ya existía anteriormente, automáticamente Maple lo sustituirá por el archivo nuevo. Existe la opción de añadir datos a un archivo ya existente, esta tarea se realiza mediante el comando fopen, que será descrito más adelante.

El formato de los caracteres guardados en el archivo se manipula mediante el segundo argumento del comando fprintf, en el ejemplo que hemos propuesto utilizamos "%d %e\n", esto quiere decir que el primer dato de cada par es un entero (%d), y el segundo está en notación científica de tipo Fortran (%e), un espacio separa el primer y segundo dato. La "\n" al final del argumento indica un salto a la línea siguiente. Por defecto, los datos son exportados con una precisión de 6 cifras significativas, pero se puede cambiar utilizando las opciones del formato %e.

Cuando se ha terminado de escribir el archivo con los datos, existe una instrucción llamada fclose que sirve para cerrarlos, si no se utiliza dicha instrucción, Maple lo cerrará automáticamente al salir del programa.

La manera más sencilla, por otra parte de crear nuestros archivos de datos es mediante el comando writedata, ya que realiza las tareas de abrir, escribir y cerrar con un solo comando. Pero a diferencia de las instrucciones anteriores, no es tan manipulable el formato de los datos.

Por último, para leer los datos provenientes de un archivo externo, en el ejemplo citado antes, se haría de la siguiente manera:

```
> do
>    xy := fscanf("Data","%d %e");
>    if xy=0 then break end if;
>    A := [op(A), xy];
> end do;
```

$$xy := [0, 0.]$$

$$A := [[0, 0.]]$$

$$xy := [1, 0.3456323]$$

$$A := [[0, 0.], [1, 0.3456323]]$$

$$xy := [2, 0.1730946]$$

$$A := [[0, 0.], [1, 0.3456323], [2, 0.1730946]]$$

$$xy := [3, 0.02675689]$$

$$A := [[0, 0.], [1, 0.3456323], [2, 0.1730946], [3, 0.02675689]]$$

$$xy := [4, 0.9865478]$$

$$A := [[0, 0.], [1, 0.3456323], [2, 0.1730946], [3, 0.02675689], [4, 0.9865478]]$$

$$xy := [5, 1.000000]$$

$$A := [[0, 0.], [1, 0.3456323], [2, 0.1730946], [3, 0.02675689], [4, 0.9865478], [5, 1.000000]]$$

$$xy := [\ ]$$

$$A := [[0, 0.], [1, 0.3456323], [2, 0.1730946], [3, 0.02675689], [4, 0.9865478], [5, 1.000000], [\ ]]$$

$$xy := 0$$

Como podemos observar, los datos se extraen secuencialmente hasta completar el conjunto de datos A. Se ha impuesto una condición para que cuando termine de leer los datos, el bucle finalice su ejecución. Existe también otro comando para leer los datos de manera más fácil, mediante readdata, veamos cómo funciona:

```
> A := readdata("Data", [integer, float]);
```

$$A := [[0, 0.], [1, 0.3456323], [2, 0.1730946], [3, 0.02675689], [4, 0.9865478], [5, 1.000000]]$$

6.4.2. Tipos y modos de archivos

La librería I/O de Maple permite guardar los archivos como STREAM o como RAW, y opera con ambos indistintamente. Generalmente se suele utilizar el modo STREAM dado que utiliza un buffer en la memoria y guarda más rápidamente la información. Los de tipo RAW no utilizan buffer y son útiles cuando se quiere examinar de manera precisa cuánto ocupa un archivo, y cómo responde el sistema operativo ante dicho archivo.

Asimismo, Maple se refiere a los archivos que maneja de dos maneras: por nombre o por su descripción.

Por nombre: Referirse a un archivo por su nombre es el más sencillo de los dos métodos. En primera instancia Maple abre el archivo de datos, esté en modo READ o WRITE, y aunque sea tipo BINARY o TEXT, de acuerdo con la operación que se vaya a realizar. La desventaja de éste método frente al otro es que en éste no se puede manipular carácter a carácter el archivo de datos.

Por descripción: Las ventajas de éste método es una mayor flexibilidad a la hora de manipular el archivo (se puede especificar si es BINARY o TEXT, así como en qué modo lo vamos a leer), incrementando la eficiencia cuando se desean hacer numerosas manipulaciones, además de poder trabajar con archivos tipo RAW.

6.4.3. Comandos de manipulación de archivos

Antes de leer o escribir un archivo, se debe abrir. Cuando nos referimos a archivos mediante el nombre, se hace automáticamente al realizar cualquier operación sobre el archivo. Cuando se utiliza el método *descriptor*, se debe especificar el archivo antes de abrirlo.

Los dos comandos para abrir archivos son open y fopen. El comando fopen se encarga de abrir archivos tipo STREAM, mientras que open los de tipo RAW.

La sintaxis de fopen es la siguiente:

fopen(**Nombre, Modo, Tipo**)

Donde Nombre indica el archivo al que nos referimos, Modo puede ser READ, WRITE o APPEND. El argumento Tipo es opcional, y especifica si un archivo es TEXT o BINARY. Si se trata de abrir un archivo para leer que no existe, fopen devolverá un error, mientras que si se trata de escribir en uno que no existe, éste se creará. Si se especifica el modo APPEND, los datos se irán añadiendo al archivo, sin empezar desde cero.

La sintaxis de open es:

open(**Nombre, Modo**)

Y los argumentos significan lo mismo que en el comando anterior.

El comando complementario al de abrir es cerrar, para ello se utilizan los comandos fclose y close, ambos de manera equivalente:

fclose(**Identificador)**
close(**Identificador**)

Donde Identificador puede ser el nombre del archivo o su descriptor. Un ejemplo de aplicación podría ser el siguiente:

```
> f := fopen("testFile.txt", WRITE):
> writeline(f, "Esto es una prueba"):
> fclose(f);
> writeline(f,"Esto es otra prueba"):
Error, (in fprintf) file descriptor not in use
```

♦ ***Determinación de la posición y ajustes:***

El concepto de abrir un archivo está muy relacionado con el de posición en el mismo. Esto es la posición a partir de la cual se va a leer o escribir un archivo, de hecho, cualquier escritura y lectura de datos avanza según el número de bytes escritos o leídos. Se puede determinar la posición actual de un archivo utilizando el comando filepos, cuya sintaxis es:

filepos(**Identificador, Posición**)

El argumento Identificador puede ser el nombre del archivo o el descriptor, si se abre un archivo que no estaba abierto, por defecto se hará en el modo READ de tipo BINARY. El argumento Posición es opcional, si no se especifica, Maple devolverá la posición actual. Si suministramos este argumento, Maple sitúa la actual posición en la especificada. La posición puede ser un número entero o infinity, que indica que es el final del archivo. Un pequeño ejemplo ligado al fichero Datos utilizado anteriormente es:

```
> filepos("Datos", infinity);
```

- ***Detectar el final de un archivo:***

El comando feof determina dónde está situado el final de un archivo. Sólo se puede utilizar con archivos tipo STREAM, o archivos abiertos explícitamente mediante el comando fopen. Se realiza de la siguiente manera:

feof(**Identificador**)

Identificador, como en los comandos anteriores, puede ser un descriptor o nombre del archivo. El comando feof devuelve true si se ha utilizado un comando del tipo readline, readbytes o fscanf para ver la longitud del archivo, sino devolverá false. Esto significa que si un archivo está compuesto por 20 bytes, y se leen mediante el comando readbytes, entonces el comando feof devuelve false.

- ***Determinar el "Status" de un archivo:***

El comando iostatus devuelve la información detallada sobre los archivos actualmente en uso. Su sintaxis es muy simple, basta con poner:

```
> iostatus()
```

Al ejecutar esta sentencia se devuelve una lista que contiene los siguientes elementos:

1. iostatus()[1]: El número de archivos que la librería I/O de Maple está utilizando.
2. iostatus()[2]: El número de comandos read anidados (es decir, cuando read lee un archivo, que en su interior contiene la sentencia read).
3. iostatus()[3]: El salto más alto que el sistema operativo impone en iostatus()[1] + iostatus()[2].
4. iostatus()[n]: (n>3) Devuelve una lista con información sobre los archivos que están actualmente en uso.

- ***Borrar archivos:***

Muchos archivos son utilizados de manera temporal, dado que no se suelen utilizar en sesiones futuras de Maple, se borran. Para realizar esta operación se utiliza el comando fremove así:

fremove(**Identificador**)

El argumento Identificador puede ser tanto el nombre como el descriptor de un archivo.

6.4.4. Comandos de entrada

El comando de entrada más simple es readline. Los caracteres de una línea de archivo son leídos, y se devuelven como string de Maple. Si se lee una línea en la que no hay ningún dato, entonces devolverá 0. Su sintaxis es la siguiente:

readline(**Identificador**)

Donde identificador puede ser un descriptor o nombre de archivo. Para hacerla compatible con el resto de versiones de Maple, se puede omitir el Identificador, en este caso Maple utilizará el default, identificador por defecto de Maple, que está protegido. Un ejemplo de procedimiento que utiliza este comando podría ser el siguiente:

```
> ShowFile := proc(fileName::string)
>    local line;
>    do
>      line := readline(fileName);
>      if line = 0 then break end if;
>      printf("%s\n", line);
>    end do;
> end proc:
> ShowFile("testFile.txt"); #archivo creado en ejemplo anterior.
```

♦ ***Lectura arbitraria de bytes de un archivo:***

Para leer los bytes en general de un archivo se utiliza el comando readbytes. Si el archivo que se va a leer está vacío, se devolverá 0, indicando que se ha llegado al final de dicho archivo. La sintaxis es la siguiente:

readbytes(**Identificador, longitud, TEXT**)

Identificador es el nombre o descriptor de un archivo, longitud, que se puede omitir, especifica cuantos bytes se desean leer, en el caso de su omisión se leerá un solo byte. El parámetro opcional TEXT indica que el resultado se va a devolver como string, aunque sea una lista de enteros.

Se puede especificar la longitud como infinity, de manera que se lea el archivo entero.

Si se ejecuta readbytes con el nombre de un archivo, y dicho archivo no está abierto, se abrirá en modo READ. Si se especifica TEXT, el archivo se abrirá como texto, y si no, como fichero tipo BINARY. Cuando readbytes devuelve 0, indica el final del archivo.

♦ ***Entrada formateada:***

Los comandos fscanf y scanf leen de un archivo, los datos con un formato específico. Su sintaxis es la siguiente:

fscanf(**Identificador, formato**)
scanf(**formato**)

Identificador es el nombre o descriptor del archivo que se va a leer. Fscanf y scanf son similares, la segunda toma como identificador default.

El formato es un string de Maple que consiste en una serie de especificaciones de conversión, de cómo son separados los caracteres. El argumento formato tiene la siguiente sintaxis:

%[*][**width**][modifiers] code

El símbolo “%” comienza las especificaciones de conversión.

El “*” opcional significa que Maple va a escanear un objeto, pero no lo devuelve como parte de un resultado.

El width es también opcional, e indica el número máximo de caracteres que se van a escanear del archivo.

Los modifiers opcionales son utilizados para indicar el tipo de valor que se va a retornar, pueden ser de muchas maneras, como se muestra en la tabla:

L ó l	Estas letras son incluidas para hacer compatible esta función con la función scanf en lenguaje C, indica que se va a devolver un "long int" o "long long". En Maple no tiene utilidad.
Z ó zc	Indica que se va a escanear un número complejo
d	Se devuelve un entero de Maple con signo
o	Devuelve un entero en base 8 (octal). El entero es convertido a decimal y devuelto a Maple como entero.
x	Se leen datos hexadecimales, se pasan a decimales, y son devueltos a Maple como enteros.
y	Se leen datos con 16 caracteres hexadecimales (formato IEEE de coma flotante), y se pasan a Maple como tipo float
e, f ó g	Los valores pueden ser enteros con signo o con decimales y se devuelven como valores de coma flotante de Maple.
he, hf ó hg	Sirven en general para leer arrays de varias dimensiones.
hx	Los datos leídos deberán ser arrays de una o dos dimensiones o números de coma flotante en formato IEEE (16 caracteres por número)
s	Se devuelven los datos leídos como string de Maple.
a	Se devuelve una expresión de Maple no evaluada.
m	Los datos deben de estar guardados en un fichero .m de Maple, y se devuelven las expresiones contenidas en el mismo.
c	Este código devuelve los caracteres como strings de Maple.
M	Las secuencias de datos que se leen han de ser elementos de tipo XML, pueden ser datos de cualquier tipo.
n	El número total de caracteres escaneados es devuelto a Maple como número entero.

- ***Leer datos de una tabla:***

El comando readdata lee archivos tipo TEXT que contienen tablas de datos. Para tablas simples es más conveniente este comando que realizar un procedimiento utilizando fscanf. La sintaxis que se utiliza es:

readdata(**Identificador, tipoDatos, numColumnas**)

El identificador es el nombre o descriptor del archivo en cuestión, mientras que tipoDatos puede ser integer o sino float. En el caso de que no se especifique, por defecto se optará por tipo float. El argumento numColumnas indica el número de columnas de datos que el archivo contiene, si no se especifica se tomará por defecto el valor 1.

Si Maple lee una única columna, readdata devolverá una lista de valores leídos. Pero si se lee más de una columna, readdata devolverá en este caso una lista de listas de valores.

6.4.5. Comandos de salida

Antes de adentrarse en los comandos de salida, es recomendable saber cómo configurar los parámetros de salida utilizando el comando interface. Dicho comando no se puede

interpretar como una instrucción de salida, sino como un mecanismo que facilita la comunicación entre el Maple y el usuario. Para establecer los parámetros, se llama al comando interface de la siguiente manera:

interface(**variable=expresión**)

El argumento variable especifica el parámetro que se desea cambiar, y el argumento expresión especifica el valor que el parámetro va a obtener. Para saber el estado de los parámetros, utilice la siguiente instrucción:

interface(**Variable**)

El argumento Variable especifica el parámetro sobre el que se desea obtener la información.

- ***Escribir strings de Maple en un archivo:***

Para realizar esta operación se utiliza en comando writeline. Cada string aparece en una línea separada. Su sintaxis es la siguiente:

writeline(**Identificador, stringSequence**)

El *Identificador* es el nombre o descriptor del archivo, y *stringSequence* es la secuencia de caracteres que writeline debe escribir. Si se omite el segundo argumento, writeline dejará una línea en blanco.

- ***Escribir bytes en un archivo:***

Para escribir uno o más caracteres individuales o bytes se utiliza el comando writebytes. Se pueden especificar los bytes tanto como string o como una lista de enteros. La utilización de este comando se hará de la siguiente manera:

writebytes(**Identificador, bytes**)

El *Identificador* es el nombre del archivo o descriptor. El argumento *bytes* especifica los bytes que se van a escribir. Si queremos por ejemplo copiar los datos de un archivo a otro, se podría hacer mediante un procedimiento de la siguiente manera:

```
> CopiarArchivo := proc(ArchivoFuente::string, ArchivoDestino::string)
>    writebytes(ArchivoDestino, readbytes(ArchivoFuente, infinity));
> end proc:
```

- ***Salida formateada:***

Los comandos fprintf y printf escriben objetos en un archivo, utilizando un formato específico. Su sintaxis es la siguiente:

fprintf(**Identificador, formato, expressionSequence**)
printf(formato, expressionSequence)

El *Identificador* es el nombre o descriptor del archivo en el que se va a escribir. Los comandos fprintf y printf son equivalentes a diferencia de que el segundo tiene como Identificador el default. El formato especifica cómo se van a escribir los datos de la secuencia de expresiones. Ésta última consiste en una secuencia de especificaciones formateadas. La sintaxis del formato es como sigue:

%[flags][width][.precision][modifiers] code

El símbolo "%" indica que empiezan las especificaciones de formato. Los flags que pueden acompañar a este símbolo son los siguientes:

+ : Los valores de salida irán precedidos del símbolo "+" o "-" según su signo.

- : La salida tendrá justificación izquierda o derecha.

En blanco: Los números en la salida se mostrarán con su correspondiente signo "-" cuando sean negativos, los positivos no llevarán el signo "+" delante.

0: A la salida se le añadirá un cero entre el signo y el primer dígito. Si se especifica como flag un "-", entonces el "0" es ignorado (no se puede utilizar ambos simultáneamente).

{}: Las llaves encierran las opciones detalladas para escribir una rtable, para más información: ?rtable_printf.

El **width** es opcional e indica el número mínimo de caracteres de salida de un campo. Los modificadores son opcionales e indican el tipo de valores que serán grabados en el archivo. Hay multitud de opciones que se pueden consultar en la ayuda del programa.

♦ ***Creación de tablas de datos:***

El comando writedata escribe datos de forma tabulada en archivos de tipo TEXT. En muchos casos, es más conveniente esta solución que la de escribir un procedimiento utilizando un bucle y sentencias fprintf. La llamada a este comando se realiza de la siguiente manera:

writedata(**Identificador, datos, TipoDatos, defaultProc**)

El *Identificador* es, como siempre, el nombre o descriptor del archivo en el que vamos a escribir. El argumento *datos* debe ser un vector, una matriz, una lista, o una lista de listas. El argumento *TipoDatos* es opcional, y especifica el tipo de datos que se van a escribir, como pueden ser floats, strings o integers. El último argumento *defaultProc* es opcional y especifica el procedimiento al que writedata va a llamar si hay algún valor que no es del tipo declarado en TipoDatos. Un buen procedimiento por defecto para salvar los datos que no corresponden a un tipo establecido podría ser el siguiente:

```
> DefaultProc := proc(f,x) fprintf(f,"%a",x) end proc:
```

Un ejemplo de entrada de datos en un archivo podría ser el siguiente. Se tiene una matriz de orden deseado (en este caso 5), y se guarda en un archivo. Nota: La matriz es la matriz de Hilbert procedente del paquete linalg de Maple:

```
> writedata("MatrizHilbert.txt", linalg[hilbert](5)):
```

Y si abrimos el archivo MatrizHilbert.txt deberíamos encontrarnos algo como lo que sigue:

1	.5	.3333333333	.25	.2
.5	.3333333333	.25	.2	.1666666667
.3333333333	.25	.2	.1666666667	.1428571429
.25	.2	.1666666667	.1428571429	.125
.2	.1666666667	.1428571429	.125	.1111111111

7- DEBUGGER

Al programar se suelen cometer errores difíciles de localizar mediante una inspección visual. Maple proporciona un *debugger* (debug en inglés significa buscar y eliminar errores) para ayudar a encontrarlos. Permite parar la ejecución de un procedimiento, comprobar o modificar el valor de las variables locales y globales y continuar hasta el final sentencia a sentencia, bloque a bloque o en una sola orden. Hay que decir el el debugger de Maple es algo más confuso que el de otros lenguajes de programación, como pueden ser VisualBasic o C++.

En la versión 9.5 de Maple se puede trabajar de dos maneras distintas (ver capítulo 2) y en el modo de trabajo Standard Math se dispone de un debugger interactivo que se describe en el apartado 'otros comandos'.

7.1. SENTENCIAS DE UN PROCEDIMIENTO

El comando `showstat` muestra las sentencias de un procedimiento numeradas. El número de sentencia puede ser útil más adelante para determinar dónde debe parar el *debugger* la ejecución del procedimiento.

Este comando se puede utilizar de varias formas:

a) `showstat (procedimiento);`

procedimiento es el nombre del procedimiento que se va a analizar. Con esta llamada se mostrará el procedimiento completo y todas las sentencias numeradas.

```
> f := proc(x) if x < 2 then print(x); print(x^2) fi; print(-x);
x^3 end:
> showstat(f);
```

```
f: = proc(x)
   1  if x < 2 then
   2   print(x);
   3   print(x^2)
       fi;
   4  print(-x);
   5  x^3
   end
```

b) Si sólo se desea ver una sentencia o un grupo de sentencias se puede utilizar el comando showstat de la forma:

```
showstat (procedimiento, numero);
showstat (procedimiento, rango);
```

En estos casos las sentencias que no aparecen se indican mediante "...". El nombre del procedimiento, sus parámetros y sus variables se muestran siempre.

```
> showstat(f,3..4);
  f: = proc(x)
        ...
     3  print(x^2)
         fi;
     4  print(-x);
        ...
  end
```

c) También se puede llamar al comando `showstat` desde dentro del *debugger*, es decir, con el *debugger* funcionando. En este caso se deberá escribir:

```
showstat procedimiento
showstat procedimiento numero_o_rango
```

Notese que no hacen falta ni paréntesis, ni comas, ni el carácter de terminación ";".

7.2. BREAKPOINTS

Para llamar al *debugger* se debe comenzar la ejecución del procedimiento y pararlo antes de llegar a la sentencia a partir de la que se quiera analizar. La forma más sencilla de hacer esto es introducir un *breakpoint* en el proceso, para lo que se utiliza el comando `stopat`.

```
stopat (nombreProc, numSentencia, condicion);
```

nombreProc es el nombre del procedimiento en el que se va a introducir el breakpoint y numSentencia el número de la sentencia del procedimiento anterior a la que se quiere situar el breakpoint. Si se omite numSentencia el breakpoint se sitúa antes de la primera sentencia del procedimiento (la ejecución se parará en cuanto se llame al procedimiento y aparecerá el prompt del debugger).

El argumento condicion es opcional y especifica una condición que se debe cumplir para que se pare la ejecución.

El comando showstat indica dónde hay un breakpoint con condicion mediante el símbolo "?". Si no se debe cumplir ninguna condición utiliza "*".

También se pueden definir breakpoints desde el debugger:

```
stopat nombreProc numSentencia condicion
```

Para eliminar *breakpoints* se utiliza el comando `unstopat`:

```
unstopat (nombreProc, numSentencia);
```

`nombreProc` es el nombre del procedimiento del que se va a eliminar el *breakpoint* y *`numSentencia`* el número de la sentencia del procedimiento donde está. Si se omite *`numSentencia`* entonces se borran todos los *breakpoints* del procedimiento. Si se realiza esta operación desde el *debugger*, la sintaxis será:

```
unstopat nombreProc numSentencia
```

7.3. WATCHPOINTS

Los *watchpoints* vigilan variables locales y globales y llaman al *debugger* si éstas cambian de valor. Son una buena alternativa a los *breakpoints* cuando lo que se desea es controlar la ejecución a partir de lo que sucede, en lugar de controlarla a partir del número de sentencia en el qué se está.

Un *watchpoint* se pude generar utilizando el comando `stopwhen`:

```
stopwhen (nombreVarGlobal);
stopwhen (nombreProc, nombreVar);
```

La primera forma indica que se llamará al *debugger* en cuanto la variable global *`nombreVarGlobal`* cambie de valor, mientras que con la segunda expresión solamente si el cambio en la variable se produce dentro del procedimiento *`nombreProc`*.

También se pueden colocar *watchpoints* desde el *debugger*:

```
stopwhen nombreVarGlobal
stopwhen [nombreProc nombreVar]
```

7.3.1. Watchpoints de error

Los *watchpoints de error* se generan utilizando el comando `stoperror`:

```
stoperror ("mensajeError");
```

Cuando ocurre un error del tipo *`mensajeError`*, se para la ejecución, se llama al *debugger*, y muestra la sentencia en la que ha ocurrido el error.

Si en el lugar correspondiente a *`mensajeError`* se escribe `all` la ejecución parará cuando se lance cualquier mensaje de error.

Los errores detectados mediante `traperror` no generan mensajes de error, así que `stoperror` no los detectará. Se debe utilizar la sintaxis específica:

```
stoperror (traperror);
```

Si la llamada se hace desde el *debuggger*:

```
stoperror mensajeError
```

Para eliminar *watchpoints de error* se utiliza el comando `unstoperror` con los mismos argumentos que `stoperror`. Si no se especifica ningún argumento, `unstoperror` borrará todos los *watchpoints de error.*

Los mensajes de error que entiende el comando `stoperror` son:

- ‘ interrupted’
- ‘time expired’
- ‘assertion failed’
- ‘invalid arguments’

Los siguientes errores se consideran críticos y no pueden ser detectados por el debugger:

- ‘out of memory’
- ‘stack overflow’
- ‘object too large’

7.4. OTROS COMANDOS

Existen otros comandos que ayudan a controlar la ejecución cuando se está en modo *debugg*:

➲ `next`: ejecuta la siguiente sentencia y se para, pero no entra dentro de sentencias anidadas.

➲ `step`: se introduce dentro de una sentencia anidada.

➲ `outfrom`: finaliza la ejecución en el nivel de anidamiento en el que se esté.

➲ `cont`: continua la ejecución hasta que termina normalmente o hasta que se encuentra un *breakpoint*.

➲ `list`: imprime las cinco sentencias anteriores, la actual y la siguiente para tener una idea rápida de dónde se ha parado el proceso.

➲ `showstop`: muestra una lista de los *breakpoints*, *watchpoints* y *watchpoints de error*.

➲ `quit`: hace salir del *debuggger*.

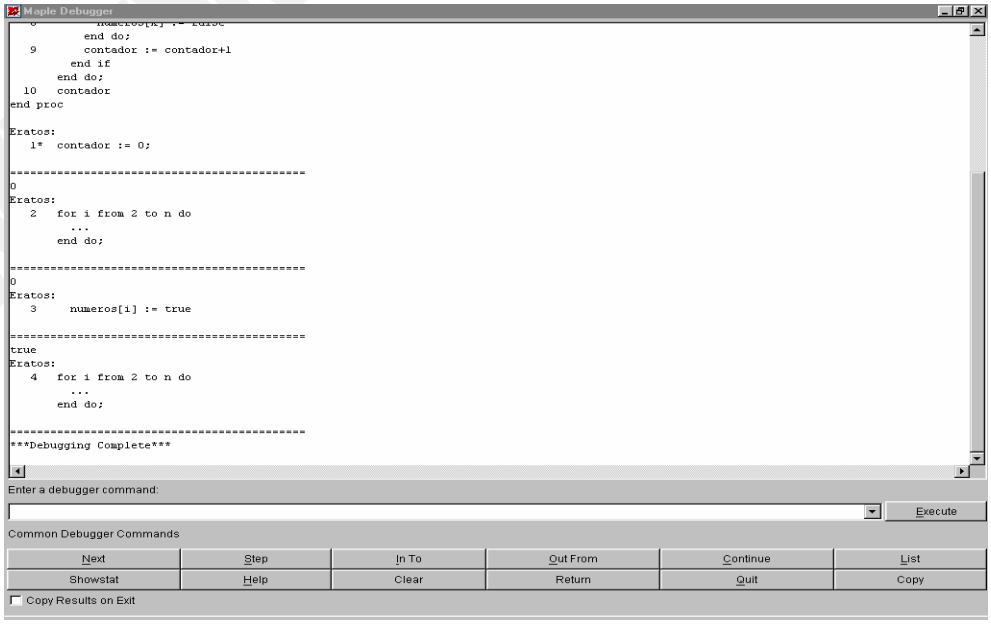
Lo descrito hasta aquí vale para Standard Math y para el Classical Worksheet. Ahora veremos una opción especial que dispone Standard Math (en la última versión de Maple): el debugger interactivo.

La esencia es la misma que el debugger normal, no hay nuevos comandos y tampoco aporta grandes ventajas. No obstante, es más agradable a la vista, puesto que dispone de ventana propia y permite que el usuario se centre más en el proceso de detección de errores.

Este debugger se inicia automáticamente al detectar:

- Un breakpoint
- Un watchpoint
- Un mensaje de error

O también se puede iniciar pulsando  durante una computación en curso. Este es el aspecto de dicho debugger:



7.5. EJEMPLO COMPLETO

Ahora mostraremos lo visto anteriormente con un ejemplo que dado un entero n, devuelve el número de números primos menores que n (incluído). En este ejemplo existe un error que no hemos sido capaces de encontrar, y por ello vamos a hacerlo con el debugger. Introduciremos los comandos sin muchas explicaciones, ya que han sido vistos anteriormente.

```
> Eratos := proc(n::integer)
>    local i, k, numeros, contador, doble;
>    contador:=0;
>    for i from 2 to n do
>       numeros[i]:=true
>    end do;
>    for i from 2 to n do
>       if numeros[i] then
>          doble := 2*i;
>          for k from doble by i to n do
>             numeros[k]=false;
>          end do;
>          contador := contador + 1
>       end if;
>    end do;
>    contador;
> end proc;
```

Si intoducimos Eratos(13) debería devolver 6 y no es así.

Primero numeramos las sentencias del procedimiento:

```
> showstat(Eratos);

Eratos := proc(n::integer)
local i, k, numeros, contador, doble;
   1   contador := 0;
   2   for i from 2 to n do
   3     numeros[i] := true
       end do;
   4   for i from 2 to n do
   5     if numeros[i] then
   6       doble := 2*i;
   7       for k from doble by i to n do
   8         numeros[k] = false
           end do;
   9       contador := contador+1
         end if
       end do;
  10   contador
end proc
```

Ahora introducimos un breakpoint al inicio del procedimiento:

```
> stopat(Eratos);                [Eratos]
```

Ejecutamos el procedimiento para n=13:

```
> Eratos(13);

Eratos:
   1*  contador := 0;
```

DBG>

Hemos entrado en el debugger (DBG). El asterisco indica que existe un breakpoint justo antes de esta sentencia. Ahora podemos acceder a los parámetros, variables locales, globales,etc. Por ejemplo:

DBG> **n**

```
13
Eratos:
   1*  contador := 0;
```

Nos ha devuelto tres cosas: el valor de n, el nombre del procedimiento y la sentencia (numerada) en la que ha parado.

Quitamos el breakpoint, reiniciamos, introducimos a continuació un *watchpoint* para la variable contador y volvemos a ejecutar:

DBG> **unstopat(Eratos);**

```
[]
Eratos:
   1   contador := 0;
```

DBG> **stopwhen([Eratos,contador]);**

[[*Eratos*, *contador*]]

```
> Eratos(13);

contador := 0
Eratos:
   2   for i from 2 to n do
         ...
       end do;
```

Ya hemos visto como funciona un poco el debugger pero todavía no hemos encontrado el error. Salimos del debugger y quitamos el watchpoint:

DBG> **quit**

```
Warning, computation interrupted
> unstopwhen();
```

[ ]

Ahora vamos a introducir un breakpoint en la 6ª sentencia y ejecutamos:

```
> stopat(Eratos,6);
```

[*Eratos*]

```
> Eratos(13);

true
Eratos:
   6*      doble := 2*i;
```

DBG>

Vamos paso a paso con el comando step:

DBG> **step**

```
4
Eratos:
   7       for k from doble by i to n do
             ...
           end do;
```

DBG> **step**

```
4
Eratos:
   8         numeros[k] = false
```

DBG> **step**

```
true = false
Eratos:
   8         numeros[k] = false
```

Aquí se observa que se hace true = false, no se asigna un valor. Se ha confundido una ecuación con una asignación. En la sentencia numero 8 tenemos que cambiar

```
> numeros[k]=false;
```

por

```
> numeros[k]:=false;
```

Con esto hemos corregido el procedimiento.

8- MAPLETS

Los Maplets son aplicaciones o interfaces gráficos que se pueden diseñar y programar con Maple y generalmente son ejecutados desde una sesión del programa. Permiten al usuario interactuar y utilizar los paquetes de Maple a través de botones y ventanas como en los programas que utilizamos habitualmente. Por ejemplo, podríamos hacer una calculadora de integrales que permitiera pedir funciones al usuario, límite inferior y superior para integrar, dibujar la gráfica de la función, etc.

Para empezar a programar un Maplet, primero hay que cargar el paquete:

```
> restart;
> with(Maplets[Elements]):
```

También hay que darle un nombre al Maplet (con los elementos entre corchetes) y para mostrarlo, se emplea habitualmente la función *Display* (en animaciones con Plotter será diferente):

```
> convplet:=Maplet([
   [Elemento11(...),Elemento12(...)],
   [Elemento21(...)]
   ... ...
   ]):
 > Maplets[Display](convplet);
```

En este ejemplo, “convplet” es el nombre del Maplet.

8.1. ELEMENTOS DE LOS MAPLETS

Los Elementos son los componentes de los Maplets. A través de ellos realizamos acciones que pueden estar vinculadas a cálculos, representación de funciones y también a obtener resultados por pantalla o cajas de texto, etc. El código de los elementos se ha de escribir entre corchetes.

Los Elementos pueden tener nombres de referencia; que se emplean para distinguirlos de los demás Elementos, en la definición de una acción de otro Elemento que deba modificar alguna propiedad del Elemento.

Se emplean las comas para distinguir los sucesivos elementos de un Maplet que vayan entre corchetes así como para separar las propiedades de cada Elemento.

8.1.1. Elementos del cuerpo de la ventana

Son los elementos visuales de la ventana del Maplet.

- Button:

  Los botones son elementos que ejecutan determinada acción (Action) al ser presionados. Tienen propiedades que se pueden modificar, tales como color, fuente, tamaño, si es visible o no etc.

```
> [Button[B1]("Caption",Shutdown(),width=89,foreground=blue)]
```

B1 es el nombre de referencia del botón. *Caption* es el texto que lleva impreso el botón y ha de ir entre comillas dobles. *Shutdown* (como en el ejemplo) es la acción que se ejecuta al presionar y puede ser otra. También se puede definir la anchura y el color del texto, entre otras opciones.

♦ Checkbox:

Permiten hacer selecciones no excluyentes, es decir, se puede tener seleccionada simultáneamente más de una opción (a diferencia de los *Radio Button*).

Al igual que los botones, se han de escribir entre corchetes y también tienen multitud de características que se pueden modificar. *Value* determina si la opción está seleccionada (true) o no (false) por defecto.

```
> [CheckBox[ch1](caption=Rojo,foreground=red,value=true,
onchange=SetOption('B1'('background') = 'red'))]
```

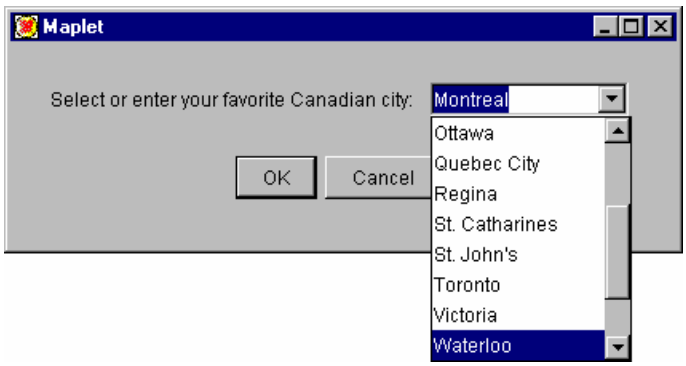
En este caso, *ch1* es el nombre de referencia de este *checkbox*.

La opción *onchange* establece la acción que se llevará a cabo al accionar la *checkbox*. En este caso, se cambia el color de fondo del elemento "B1" (por ejemplo, un botón).

Nota: *caption=Rojo* equivale a escribir "Rojo"; igual que antes, es el texto que lleva el elemento.

♦ ComboBox:

Es una lista desplegable predefinida en la que se puede escribir el nombre que se busca. Su aspecto es el siguiente:



Se ha de escribir, entre corchetes. He aquí un ejemplo aclaratorio:

```
> [ComboBox[C1]("nombre",sort(["Ronaldinho","Iraola","Van Basten"],
lexorder))]
```

La propiedad importante es *sort([ ])* donde se escriben las opciones de la lista entre comillas y después del corchete se establece el criterio de ordenación (*lexorder* lo hace alfabéticamente).

Devuelve el nombre seleccionado de la lista o escrito en la caja de texto que lleva incorporada.

Otros elementos muy similares son los *DropDownBox* que son *ComboBox* que no permiten escribir una opción que no sea de la lista. Su sintaxis es idéntica.

♦ **Label:**

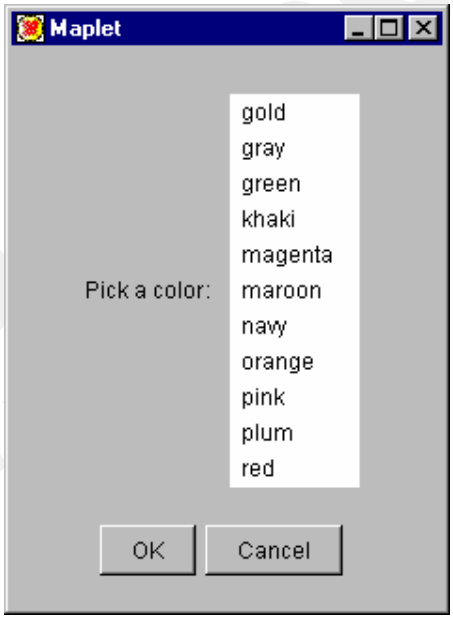
Son etiquetas que pueden contener texto o imágenes.

```
> [Label[L1]("Introduzca nombre",'font'=Font("courier",14))]
```

Es posible definir la fuente y el tamaño del texto como se indica.

♦ **ListBox:**

Son listas predefinidas en las que se puede hacer una selección múltiple utililzando las teclas MAYUS y CONTROL.




```
> [ListBox[lb1]("nombre",sort(["alpargata","tocino","berenjena"],
lexorder))]
```

Devuelve una lista separada por comas que contiene la selección del usuario. El modo de introducir los elementos de la lista es igual que en las *ComboBox*.

♦ **MathMLEditor:**

Sirve para crear y pasar ecuaciones a Maple u otros elementos del Maplet. Se pueden introducir los datos por teclado o con la ayuda de las paletas que se visualizan apretando el botón derecho del ratón sobre la ventana del MathMLEditor. Se utiliza la sentencia *MathML[Import]* para obtener lo escrito en la ventana del editor.

```
> with(Maplets[Elements]):
```

```
maplet := Maplet([
   [BoxCell("Enter an expression")],
   [MathMLEditor('reference'='ME1')],
   [Button("Done", Shutdown([ME1]))]
]):
result:=Maplets[Display](maplet);
MathML[Import](result[1]);
```

♦ **MathMLViewer:**

Muestra expresiones del tipo MathML mediante su propiedad *Value* y la función *MathML[Export]*( ).

```
> [MathMLViewer('value' = MathML[Export](int(sin(x^2), x)))]
```

♦ **Plotter:**

Permite representar funciones en 2D o 3D y animaciones. Para estas últimas, las propiedades más importantes son *play*, *stop* (que hay que escribir entre comillas graves, *`stop`*, para diferenciarla del "stop", keyword o palabra reservada), *pause* y *continuous*, que pueden tener valor *true* o *false* (por defecto, *false*). Generalmente estas propiedades se modifican debido a la acción sobre otros elementos del Maplet como botones o *checkbox*, para lo cual se emplea el nombre de referencia del elemento Plotter.

Para poder animar una función, hay que definirla como animable (mejor fuera o antes que el Maplet):

```
> p:=plots[animate](plot,[a*x,x=-10..10],a=0..100,frames=20):
```

Luego el Plotter dentro del Maplet:

```
> [Plotter[P](p,continuous=false)]
```

La opción *continuous=false* hace que la animación se detenga cuando *a* alcance su valor final. Por otro lado, los botones habrán de escribirse de la siguiente manera para que actúen sobre el *Plotter*:

```
> [Button("PLAY",SetOption(P('play')=true))],
[Button("STOP",SetOption(P('`stop`')=true))],
[Button("Exit",Shutdown())]
```

Nota: observar cómo se ha de diferenciar *stop* entre comillas graves.

Para mostrar este Maplet, la llamada ha de ser:

```
> Maplets:-Display(convplet);
```

♦ **RadioButton:**

Los *RadioButton* son *CheckBox* exclusivas, es decir, no se puede seleccionar más de una opción perteneciente al mismo grupo. Entonces, escribiremos la sintaxis de los

RadioButton, similar a la de los *CheckBox* salvo en esta propiedad y añadiremos al Maplet, fuera de los corchetes "principales", un elemento de grupo de botones *BottonGroup* definiendo su nombre de referencia:

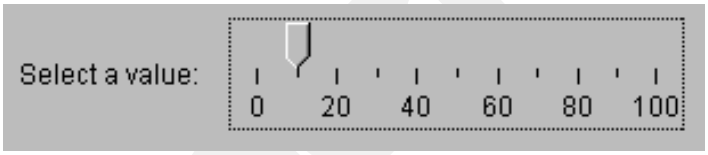
```
> with(Maplets[Elements]):
maplet := Maplet([
  [RadioButton['RB1']("1", 'value'=true, 'group'='BG1'),
RadioButton['RB2']("2", 'value'=false, 'group'='BG1')],
  ], ButtonGroup['BG1']()):
Maplets[Display](maplet);
```

- **Slider:**

Permite escoger un valor entero de un rango. En su definición se dan el *lower* y *upper*, los valores mayor y menor. También se puede decidir si es vertical u horizontal.

```
> [Slider['SL1'](0..100, 10, 'showticks', 'majorticks'=20,
'minorticks'=10, 'snapticks'=false)]
```

Majorticks son las marcas con número y *minorticks* sin.



- **Table:**

Define una tabla sobre la ventana del Maplet. Ha de ser incluido como elemento de un *BoxCell* para que aparezca el encabezado que, en cualquier caso, se ha de incluir siempre. En el ejemplo se observa cómo se introducen los datos (por filas) y que primero se escribe el encabezado (A-B) entre corchetes y después los datos.

```
> maplet := Maplet([
   BoxCell(Table([A,B], [[1, 2], [3, 4]]), 'as_needed'),
   Button("OK", Shutdown())
]):
Maplets[Display](maplet);
```

- **Cajas de texto (TextBox):**

Son cajas en las que se puede escribir (input) o mostrar (output) información. Su propiedad *editable* determina si se puede modificar su valor (si es =*true*) o impedir que se escriba en ella. Las cajas de texto tienen menús pop-up que se activan haciendo clic con el botón derecho. Conviene darle más de una linea de tamaño si se va a escribir tanto o más (*height*= y *width*= o bien *h..w*).

```
> TextBox['IB1'](width=7,height=70)] ó TextBox['IB1'](7..70)]
```

- **TextField:**

Es un campo de entrada o salida de datos de una sola línea, dependiendo si su opción 'editable' está en true o false.

- **ToggleButton:**

Son como los *RadioButton* pero en estos se puede añadir un nombre dentro del botón:

```
> [ToggleButton['TB1']("This is a toggle button", 'value'=true,
'group'='BG1')]
```

8.1.2. Elementos de diseño

Los elementos introducidos en el código se van situando de arriba abajo y de izquierda a derecha en la ventana del Maplet. Si queremos darle determinado aspecto a la distribución de botones, *checkbox* etc. dentro de la ventana, hay que emplear los elementos de diseño, *BoxLayout* y *GridLayout*. Los elementos del cuerpo de la ventana (botones, plotter, *radiobutton* etc.) van dentro de estos. Explicaremos únicamente *BoxLayout*:

- **BoxLayout:**

Controla la posición horizontal (elemento *BoxRow*) o vertical (elemento *BoxColumn*) de los elementos. Para que, en horizontal, dos elementos permanezcan pegados (juntos) al cambiar el tamaño de la ventana, se utiliza la opción *HorizontalGlue( )* como elemento de *BoxRow*. Si se coloca después (a la derecha) de dos (o más)elementos, estos permanecerán juntos en la parte izquierda de la ventana. Si se escribe antes (a la izquierda) de dos elementos, se quedarán en la parte derecha de la ventana. Es necesario escribir *'halign'='none'* en el elemento *BoxColumn* para que obedezca al orden de *HorizontalGlue( )*. En este ejemplo se muestra todo esto; copie y ejecute este Maplet y cambie el tamaño de la ventana para ver cómo actúa *HorizontalGlue( )*.

```
> ordenaplet:=Maplet(
     BoxLayout('halign'='none',
      BoxColumn(
       BoxRow("Primera fila,izqu",HorizontalGlue()),
         BoxRow("Segunda fila,<-",HorizontalGlue(),"Segunda
         fila,der.->"),
    BoxRow("Tercera fila,izqu.1","Tercera
    fila,izqu.2",HorizontalGlue())
       )
     )
    ):
Maplets[Display](ordenaplet);
```

Nota: Observe que el código no va entre corchetes como es habitual, sino dentro del elemento de diseño *BoxLayout( )*.

8.1.3. Elementos de la barra de menú

Para hacer menús en un Maplet, se ha de indicar el nombre de referencia de la "barra de menú" (*menubar*) en el elemento *Window*. Después se define cada menú desplegable (*menu*) y sus opciones (*menu item*) dentro de la *menubar*. También se pueden definir *CheckBoxMenuItem* y *RadioButtonMenuItem* en cada menú desplegable.

```
> maplet := Maplet(
  Window('menubar'='MB1', [[Button("OK", Shutdown("Closed from
  button"))]]),
  MenuBar['MB1'](
  Menu("File", MenuItem("Close", Shutdown("Closed from menu"))),
  Menu("Opciones",MenuItem("Salir",Shutdown()),MenuItem("Salir2",S
  hutdown()))
 )):
```

8.1.4. Elementos de una barra de herramientas

De manera similar a los menús, primero hay que definir en el elemento *Window* qué *toolbar* utilizaremos y después definiremos cómo es y qué botones tiene. Se pueden incluir *ToolBarButton* (definido en el elemento *ToolBar*) y *ToolBarSeparator* (definido en el elemento *ToolBar*).

```
> maplet:=Maplet(
  Window('toolbar'=TB,[Button("Exit",Shutdown("Exit Boton"))],
  ToolBar[TB](
          ToolBarButton("salir",Shutdown("salir normal")),
          ToolBarSeparator(),
          ToolBarButton("salir2",Shutdown("salir 2"))
 ))):
 result:=Maplets[Display](maplet);
```

8.1.5. Elementos de comandos

Son los comandos que se pueden ejecutar al presionar un botón, escoger una opción de un CheckBox o cambiar el valor de una caja de texto, por ejemplo. Se escriben dentro de los elementos y pueden llevar entre paréntesis el nombre de referencia del objeto que actúan o un resultado de salida.

- **CloseWindow:**

Cierra la ventana en ejecución mediante una referencia a la misma.

```
> [Button("Close This Window", CloseWindow('W2'))]
```

- **Evaluate:**

El comando Evaluate ejecuta un procedimiento de Maple con los argumentos marcados en args de la sesión actual de Maple.

- **RunDialog:**

El elemento *RunDialog* ejecuta elementos de diálogo (ver apartado 1.1.6) y debe contener la referencia al diálogo que se va a mostrar en pantalla. Después se define el diálogo, por ejemplo, *MessageDialog(“ “).*

Un ejemplo de estas dos últimas opciones:

```
> maplet := Maplet(
   Window([
      [TextField['TF1']()],
      [
         Button("Differentiate with respect to x", Evaluate('TF1' =
'diff(TF1, x)')),
         Button("Help", RunDialog('MD1')),
         Button("Exit", Shutdown(['TF1']))
      ]
   ]),
   MessageDialog['MD1']("See ?diff for help with the
differentiation command")
):
Maplets[Display](maplet);
```

- **RunWindow:**

Hay que hacer referencia a la ventana que se quiere activar.

Si se va a abrir una ventana y cerrar otra, se escribirá en el elemento que lo ordene:

```
> Button("Integration",Action(RunWindow('W3'),CloseWindow('W1')))
```

- **SetOption:**

Permite cambiar algunas opciones de determinados elementos del Maplet utilizando para identificarlos su nombre de referencia. Por ejemplo, borrar una caja de texto o cambiar el color de la fuente de una etiqueta (véase el ejemplo del Plotter-Elementos del cuerpo de la ventana).

- **Shutdown:**

Cierra un Maplet que se está ejecutando. Tiene la opción de devolver un valor a una sesión de Maple, incluso puede devolver los valores específicos guardados en una caja de texto o cualquier valor fijo.

Un ejemplo de *SetOption* y *Shutdown*:

```
> with(Maplets[Elements]):
maplet2 := Maplet([
      [Label('caption'="Enter an expression")],
      ["Input Field:", TextField['TF1'](20)],
      [
```

```
          Button("Change Font", SetOption('TF1'('font') = 'F1')),
          Button("Change Color", SetOption('TF1'('background') =
'red')),
          Button("Exit", Shutdown(['TF1']))
       ]
         ],
    Font[F1]("courier", size=14)
):
Maplets[Display](maplet2);
```

8.1.6. Elementos de diálogo

Los diálogos son pequeñas ventanas que suministran información al usuario, como pueden ser mensajes de aviso o de advertencia, o entrada de datos, como nombres de archivos. Los usuarios pueden responder a un diálogo mediante los botones incluidos en el mismo e indicar la acción a seguir según sea el botón que se presione. El autor de un Maplet puede modificar algunas de las características de las ventanas de diálogo, como por ejemplo el título de la ventana, el mensaje... Los diálogos son ejecutados mediante el elemento RunDialog. La mayoría de ellos tienen las opciones *onapprove* y *oncancel* (en función de los botones que tengan) y en ellas se definen las acciones a ejecutar cuando se pulse uno u otro botón.

- **AlertDialog:**

Advierte de un riesgo potencial. Permite al usuario elegir entre la opción de continuar (OK) o parar (Cancel).

```
> maplet := Maplet(AlertDialog(
    "Assuming x > 0 leads to a contradiction",
    'onapprove' = Shutdown("true"),
    'oncancel' = Shutdown("FAIL")
)):
Maplets[Display](maplet);
```

- **ColorDialog:**

Muestra una paleta de colores estándar para elegir un color.

- **ConfirmDialog:**

Permite al usuario especificar cómo se desarrolla una acción. Por ejemplo, si tenemos un diálogo con el texto: “Es x mayor que 0?”, se presentarán las opciones Yes, No y Cancel.

- **FileDialog:**

Es un diálogo diseñado para elegir un archivo en concreto.

- **InputDialog:**

El elemento InputDialog es similar al elemento AlertDialog con la diferencia de que el InputDialog contiene una caja de texto a través de la cual el usuario puede modificar datos o introducirlos. Se puede incluir un valor inicial en la caja de texto cuando se inicia el diálogo.

- **MessageDialog:**

Presenta información al usuario y se cierra haciendo clic sobre el botón OK que incluye.

- **QuestionDialog:**

Presenta una pregunta al usuario y permite responder Yes o No.

8.2. HERRAMIENTAS

Las herramientas de los Maplets son ayudas a los programadores de Maplets. El subpaquete Maplets[tools] contiene rutinas para manipular e interactuar con maplets y elementos de los maplets.

Este paquete es accesible mediante la instrucción with(Maplets[Tools]). Para detalles concretos sobre herramientas se puede consultar Maplets[Tools]. Algunas de las rutinas útiles podrían ser las siguientes:

- **AddAttribute:**

Añade atributos a un elemento construido con anterioridad.

- **AddContent:**

Añade contenido a un maplet construido previamente.

- **Get:**

Devuelve el valor de un elemento especificado de un maplet en ejecución. Debe ser utilizado dentro de un procedimiento. No se puede utilizar en la definición de un maplet.

- **ListBoxSplit:**

Convierte el valor de una ListBox en una lista de strings.

- **Print:**

Imprime la estructura de datos en XML. Son incluidos los valores por defecto. Esto es útil cuando un maplet no se comporta como se desea.

- **Set:**

No se puede utilizar en la definición de un maplet. Debe usarse dentro de un procedimiento. La función Set determina el valor de un elemento específico de un maplet que está ejecutándose.

- **StartEngine:**

Empieza el entorno de los Maplets.

- **StopEngine:**

Detiene el entorno de los Maplets. Todos los Maplets que estén en ejecución se cerrarán.

8.3. EJECUTAR Y GUARDAR MAPLETS

Como se ha dicho al principio, para ejecutar (mostrar) un Maplet, se debe utilizar la función Display.

Un Maplet se puede guardar como tal, aislando sus sentencias de código en un *worksheet* distinto si lo tenemos en una página con más cosas y escogiendo "Maplet" en la opción de "Guardar Como". Para ejecutarlo se puede hacer doble clic sobre el archivo guardado y se ejecutará basándose en un programa llamado Maplet Viewer, dado que no son programas compilados.

8.4. RECOMENDACIONES

A continuación presentamos algunos consejos que facilitarán la escritura y lectura de los Maplets:

- Emplear una línea para cada elemento (si es breve, escribir más de uno por línea)
- Utilizar las comillas simples (‘ ‘) para escribir nombres de referencia para evitar la interrupción del programa en caso de volver a asignar ese nombre de referencia a otro elemento
- Escribir escalonadamente utilizando la barra espaciadora o la tecla TAB para diferenciar claramente qué elementos están dentro de otros, etc.
- Se pueden introducir *execution groups* (>) situándose con el cursor a la izquierda del símbolo ">" y presionando ENTER. Para unir después todas las sentencias de ejecución en una sola, bloquear todas (tecla MAYÚS y arrastrar) y presionar F4 o Edit->Split or Join-> Join Execution Groups.

9- CODE GENERATION PACKAGE

Una de las ventajas de la programación con Maple es la posibilidad de traducir el lenguaje de Maple al de otros programas. Para ello, existe un paquete llamado CodeGeneration. Sin embargo, las funciones más específicas como las contenidas en subpaquetes, pueden no traducirse correctamente.

9.1. OPCIONES Y LIMITACIONES DE *CODE GENERATION*

- No puede traducir bucles del tipo *for-in* pero sí los *for* y *while*
- Las variables del tipo *numeric*, *float*, *sfloat* e *integer* son reconocidas por *CodeGeneration* pero las tres primeras son consideradas equivalentes y se traducen a variables de punto flotante. No se pueden traducir los números complejos.
- Si no se declara, el tipo de variable del resultado de un procedimiento se deduce y se le aplica
- En general, las funciones trigonométricas, hiperbólicas y logarítmicas son reconocidas por el paquete
- Los *arrays* y *rtables* son traducidos como arrays del tipo del lenguaje al cual traducimos
- La función *optimize* se emplea para optimizar el código de Maple antes de ser traducido.

  Hay multitud de opciones disponibles en el paquete –cada traductor tiene propiedades especiales (*Details*), las cuales se pueden consultar mediante la ayuda.
- Para cargar el paquete se ha de escribir:

```
> restart:
> with(CodeGeneration):
```

9.2. TRADUCTORES DE CODEGENERATION

Hay cinco traductores de lenguaje: C, Fortran, Java, Matlab y VisualBasic:

9.2.1. Traductor C

La función C traduce código Maple a ANSI C. Su notación es:

C(x, cgopts)

Si el parámetro x es una expresión algebraica, entonces se genera una sentencia en C asignando una variable a la expresión. Si el parámetro, en cambio, es una lista, una rtable, entonces se producirá un array en lenguaje C. Sólo los elementos inicializados de una rtable se traducirán al C.

En el caso de que el parámetro x sea una lista en la forma nm=expr donde nm es un nombre y expr es una expresión algebraica, se entenderá que representa una secuencia de sentencias de asignación. En este caso, también se generará una secuencia del mismo tipo en C.

Por último, si el parámetro x es un procedimiento, se generará una función de C, con todas las sentencias necesarias para generar un código similar (no tiene porqué ser idéntico, pero opera idénticamente).

El parámetro cgopts representa cualquiera de las opciones disponibles para CodeGeneration descritas en ?CodeGenerationOptions.

Un pequeño ejemplo de traducción podría ser el siguiente:

```
> f := proc(n)
local x, i;
x := 0.0;
for i to n do
x := x + i;
end do;
end proc:
C(f);
double f (int n)
{
  double x;
  int i;
  double cgret;
  x = 0.0e0;
  for (i = 1; i <= n; i++)
  {
    x = x + (double) i;
    cgret = x;
  }
  return(cgret);
}
```

9.2.2. Función Fortran

Su sintaxis es similar a la de la función C, como se puede observar:

Fortran(**x**, **cgopts**)

Donde el argumento x puede ser una expresión, una lista, una rtable, un array o un procedimiento, en cuyos casos se traduce de la misma manera que en la función C, con la diferencia de que en los procedimientos se puede traducir tanto a una función como a una rutina, según le convenga a Maple. El argumento cgopts representa las opciones presentes en el paquete CodeGeneration.

Continuando con el ejemplo expuesto en la función C, su versión en Fortran sería:

```
> f := proc(n)
  local x, i;
  x := 0.0;
  for i to n do
    x := x + i;
  end do;
end proc:
Fortran(f);
```

```
      doubleprecision function f (n)
              integer n
              doubleprecision x
              integer i
              doubleprecision cgret
              x = 0.0D0
              do 100, i = 1, n, 1
                x = x + dble(i)
                cgret = x
      100     continue
              f = cgret
              return
            end
```

9.2.3. Función Java

Esta función, traduce Maple al lenguaje Java mediante la siguiente sintaxis:

Java(**x**, **cgopts**)

Donde x, al igual que en las otras dos funciones de este paquete representa una expresión, array, rtable o procedimiento, y se traducen de la misma manera. En el caso de que x sea un procedimiento, se creará una clase en Java. El ejemplo anteriormente mostrado tendría el siguiente aspecto en Java:

```
> f := proc(n)
local x, i;
x := 0.0;
for i to n do
x := x + i;
end do;
end proc:
Java(f);
class CodeGenerationClass {
  public static double f (int n)
  {
    double x;
    int i;
    double cgret;
    x = 0.0e0;
    for (i = 1; i <= n; i++)
    {
      x = x + (double) i;
      cgret = x;
    }
    return(cgret);
  }
}
```

9.2.4. Función Maple

En este caso también la sintaxis es similar y en caso de que x sea un procedimiento, se crea un módulo de Matlab con una función representado el procedimiento.

Matlab(**x**, **cgopts**)

```
> f := proc(n)
```

```
  local x, i;
  x := 0.0;
  for i to n do
    x := x + i;
  end do;
end proc:
Matlab(f);
function freturn = f(n)
  x = 0.0e0;
  for i = 1:n
    x = x + i;
    cgret = x;
  end
  freturn = cgret;
```

9.2.5. Función Visual Basic

Las posibilidades del argumento x son las mismas que en los demás casos.

VisualBasic(**x**, **cgopts**)

```
> f := proc(n)
  local x, i;
  x := 0.0;
  for i to n do
    x := x + i;
  end do;
end proc:
VisualBasic(f);
Public Module CodeGenerationModule
  Public Function f(ByVal n As Integer) As Double
    Dim x As Double
    Dim i As Integer
    Dim cgret As Double
    x = 0.0E0
    For i = 1 To n
      x = x + CDbl(i)
      cgret = x
    Next
    Return cgret
  End Function
End Module
```