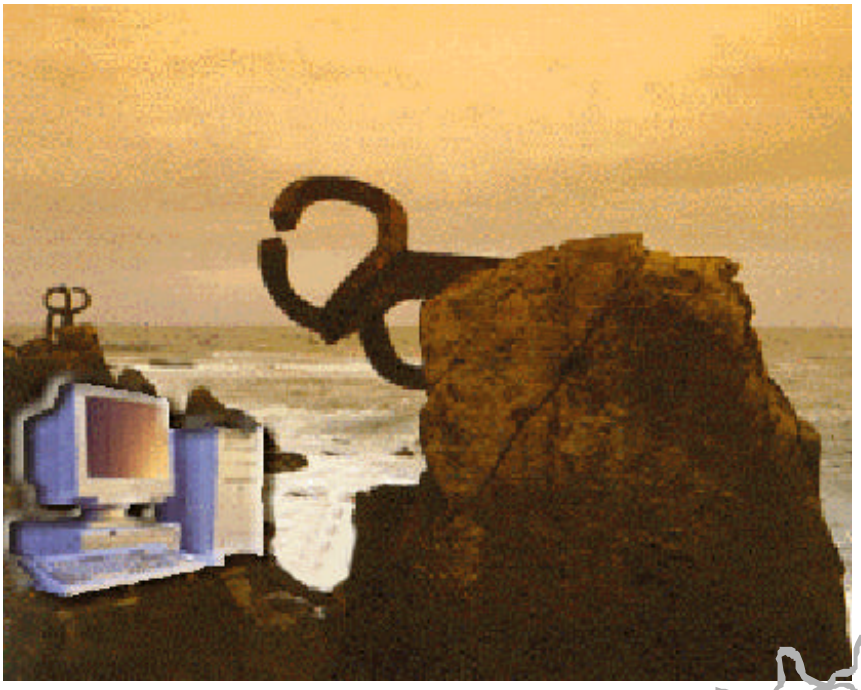
Escuela Superior de Ingenieros Industriales
Industri Injineruen Goimailako Eskola

UNIVERSIDAD DE NAVARRA - NAFARROAKO UNIBERTSITATEA

Aprenda Informática ...

Aprenda Maple 8

como si estuviera en primero



San Sebastián, Noviembre 2002

Ainhoa Ochoa Álvarez • Iñaki Esnaola Cano
Eduardo Portu Repollés • Eduardo Granados Mateo

COORDINADO POR:
Carlos Bastero de Eleizalde

Aprenda MAPLE 8

como si estuviera en primero

Ainhoa Ochoa Álvarez
Iñaki Esnaola Cano
Eduardo Portu Repollés
Eduardo Granados Mateo

Coordinado por Carlos Bastero de Eleizalde

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

PRESENTACIÓN

El manual ***Aprenda Maple 8 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 y por Javier García de Jalón, Amale Garay, Itziar Ricondo y Pablo Valencia, en Maple 5.

Maple 8 es un software con muchísimas posibilidades y exponerlas todas sería poco adecuado para la pretensión de este manual. En esta nueva edición se ha pretendido introducir las librerías especializadas y presentar aquellos comandos que, en nuestra opinión, pueden tener mayor interés para los alumnos de Ingeniería Industrial y de Ingeniería de Telecomunicación. Nos ha parecido conveniente resaltar la capacidad simbólica del programa así como su potencialidad desde el punto de vista numérico, que queda reforzada si el usuario dispone de Matlab.

El manual está distribuido en secciones de tal modo que, trabajadas en el orden propuesto, ayuda a entender mejor cómo funciona Maple. En el último capítulo se explican unos rudimentos de programación en Maple, introduciendo los procedimientos (*procedures*), módulos (*modules*) y los *maplets.*

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

Por último, existe una lista electrónica de usuarios de Maple (The Maple Users Group, MUG) que permite discutir problemas, resolver dudas avanzadas, etc. Para recibir información de cómo suscribirse basta con enviar un mensaje a la dirección majordomo@scg.uwaterloo.ca con el comando **info maple-list** en el texto del mensaje.

San Sebastián, 10 de octubre de 2002

1- INTRODUCCIÓN A MAPLE 8

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 ser este programa.

Además, Maple cuenta con un gran conjunto de herramientas gráficas que permiten visualizar los resultados (a 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 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 8?

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 8, que tiene algunas mejoras respecto a la versión anterior.

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***. En cualquier caso, el programa arranca y aparece la ventana de trabajo (ver figura 1), que es similar a la de muchas otras aplicaciones de ***Windows***. 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 8.

2- DESCRIPCIÓN GENERAL DE MAPLE 8

2.1. MANEJO DE LA AYUDA DE MAPLE

El sistema de ayuda de Maple 8 se parece al de las demás aplicaciones de Windows, aunque tiene ciertas peculiaridades que conviene conocer. Además de ser accesible mediante el menú ***Help*** de la ventana principal de Maple, podemos acceder a la ayuda sobre un comando específico desde la hoja de trabajo anteponiendo un signo de interrogación a dicho comando. Por ejemplo:

```
> ?Diff;
```

El método anterior abre una ventana con toda la información disponible sobre el comando deseado. Otra forma de abrir la misma ventana es colocar el cursor sobre el nombre del comando y ver que en el menú ***Help*** se ha activado la opción de pedir información sobre ese comando en particular (***Help/Help on "comando"***). Si se está interesado en una información más específica, se pueden utilizar los comandos siguientes:

```
info(comando)
usage(comando)
related(comando)
example(comando)
```

Estos dan información particular sobre para qué sirve, cómo se usa, qué otros comandos relacionados existen y algunos ejemplos sobre su uso, respectivamente.

El ***browser*** o sistema de exploración del ***Help*** tiene un interés particular. Con él, se puede examinar cualquier función, las distintas librerías, etc. Eligiendo ***Help/Using Help*** aparece una ventana con una pequeña introducción de cómo emplear la ayuda de Maple. En la parte superior de la ventana aparece el browser con el que se puede acceder a todas las páginas del Help de Maple. Otras formas prácticas de buscar ayuda es a través de ***Topic Search*** y ***Full Text Search*** del menú ***Help***, que proporcionan ayuda sobre un tema concreto.

Asimismo, es recomendable efectuar el ***New User´s Tour*** si nunca se ha manejado Maple, ya que éste nos proporciona un resumen de las funcionalidades y comandos del programa más genéricos y utilizados con ejemplos ilustrativos.

2.2. NÚCLEO, LIBRERÍAS E INTERFACE DE USUARIO

Maple consta de tres partes principales: el ***núcleo***, las ***librerías***, y la ***interface de usuario.***

El ***núcleo***, o *kernel*, es la parte central del programa (escrita en el lenguaje C), encargada de realizar las operaciones matemáticas fundamentales.

Las ***librerías*** deben su existencia a que Maple dispone de más de 3000 comandos. Sólo los más importantes se cargan en memoria cuando el programa comienza a ejecutarse. La mayor parte de los comandos están agrupados en distintas librerías temáticas, que están en el disco del ordenador. Para poder ejecutarlos, hay que cargarlos anteriormente. Puede optarse por cargar un comando o función aislada o cargar toda una librería. Esta segunda opción es la más adecuada si se van a utilizar varias funciones de la misma a lo largo de la sesión de trabajo. También el usuario puede crear sus propias librerías.

El comando ***readlib(namefunc)*** carga en memoria la función solicitada como argumento. Por su parte, el comando ***with(library)*** carga en memoria toda la librería especificada. Con el *Browser* de Maple se pueden ver las librerías disponibles en el programa y las funciones de que dispone cada librería.

Maple dispone de funciones de librería que se cargan automáticamente al ser llamadas (para el usuario son como las funciones o comandos del núcleo, que están siempre cargados). Las restantes funciones deben ser cargadas explícitamente por el usuario antes de ser utilizadas. Ésta es una fuente importante de dificultades para los que comienzan. El tema de las librerías se abordará en una segunda parte de este manual, deteniéndose en las de mayor utilidad.

La ***interface de usuario*** se encarga de todas las operaciones de entrada/salida, y en general, de la comunicación con el exterior. Seguidamente analizaremos este último aspecto.

2.3. DESCRIPCION DEL ENTORNO DE TRABAJO

Cuando se arranca Maple aparece la ventana principal, que corresponde a una ***hoja de trabajo*** (*worksheet*). En una hoja de trabajo hay que distinguir entre las regiones de *entrada*, *salida* y *texto.* Puede aparecer un cuarto tipo de región –de *gráficos*– si con el comando ***Paste*** se pegan sobre ella gráficos copiados de otras ventanas.

A medida que se van ejecutando comandos en la hoja de trabajo, Maple va creando variables, almacenando resultados intermedios, etc. Al estado del programa en un determinado momento del trabajo se le llama ***estado interno*** del programa, que contiene las variables definidas por el usuario, modificaciones de los valores por defecto, resultados anteriores e intermedios, etc. Se puede volver en cualquier momento al estado interno inicial tecleando el comando ***restart*** o pulsando el icono situado más a la derecha de la barra de herramientas (función equivalente).

A diferencia de MATLAB (en el que se podían recuperar sentencias anteriores con las flechas, pero no ejecutar directamente en la línea en que habían sido escritas), en Maple el usuario puede moverse por toda la hoja de trabajo, situando el cursor en cualquier línea, ejecutando comandos en cualquier orden, editando y volviendo a ejecutar sentencias anteriores, insertando otras nuevas, etc. Es evidente que eso puede modificar el estado interno del programa, y por tanto afectar al resultado de alguna de esas sentencias que dependa de ese estado. El usuario es responsable de hacer un uso inteligente de esa libertad que Maple pone a su disposición.

En Maple no se pueden utilizar las flechas para recuperar comandos anteriores. Sin embargo, se puede hacer uso de la barra de desplazamiento vertical de la ventana para ir a donde está el comando, colocar el cursor sobre él, editarlo si se desea, y volverlo a ejecutar pulsando la tecla ***intro***. También puede utilizarse el ***Copy*** y ***Paste*** entre distintas regiones de la hoja de trabajo.

En Maple pueden mantenerse abiertas varias hojas de trabajo a la vez. Los estados internos de dichas hojas de trabajo son los mismos.

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

Maple permite la posibilidad de organizar los trabajos que se realicen en la hoja, de una manera muy clara y estructurada. Pueden escribirse textos, insertar comentarios, formar grupos, formar secciones y establecer accesos directos a otras partes de la hoja de trabajo o a otras hojas.

Las hojas de trabajo tienen cuatro tipos de ***regiones*** o zonas, que se comportan de un modo muy diferente. Son las siguientes:

1. – región de entrada
2. – región de salida
3. – región de texto
4. – región de gráficos

La entrada puede efectuarse de dos maneras diferentes: ***Maple Notation*** o ***Standard Math***, que se seleccionan del menú ***Options/Input Display.*** También puede cambiarse el tipo de notación de una sentencia clicando sobre ella con el botón derecho y seleccionando la opción de ***Standard Math*** del cuadro de diálogo que aparece.

Si está seleccionada la opción ***Maple Notation***, las sentencias se irán viendo en la hoja de trabajo a la vez que se escriben. Éstas deberán seguir la notación de Maple y acabar en punto y coma.

Si por el contrario está activa la opción ***Standard Math***, aparecerá junto al prompt un signo de interrogación (?) y al teclear la sentencia se escribirá en la barra de textos, en lugar de escribirse en la hoja de trabajo. Para que se ejecute la sentencia hay que pulsar dos veces seguidas la tecla “intro”.

Una opción muy interesante en Maple 8 es la posibilidad de insertar datos de entrada mediante las paletas disponibles. Esta opción simplifica mucho la entrada de datos, sobre todo al principio, cuando el usuario novel desconoce la tipografía empleada por el programa para los distintos tipos de objetos. En el menú ***View/ Palettes*** disponemos de 4 tipos de paletas: una para introducir símbolos, otra para expresiones, otra para vectores y una cuarta para matrices. Así si queremos calcular la integral definida de $x^2 dx$ entre 0 y 1, tendremos dos opciones. La primera será emplear el comando adecuado para su cálculo, con su sintaxis bien definida y la segunda será emplear el menú ***expresion palettes*** y ahí seleccionar el icono de integral indefinida:

```
> int(x^2,x=0..1); #primera opción
```

$$\frac{1}{3}$$

$$> \int_0^1 x^2\,dx \qquad \text{#Segunda opción}$$

$$\frac{1}{3}$$

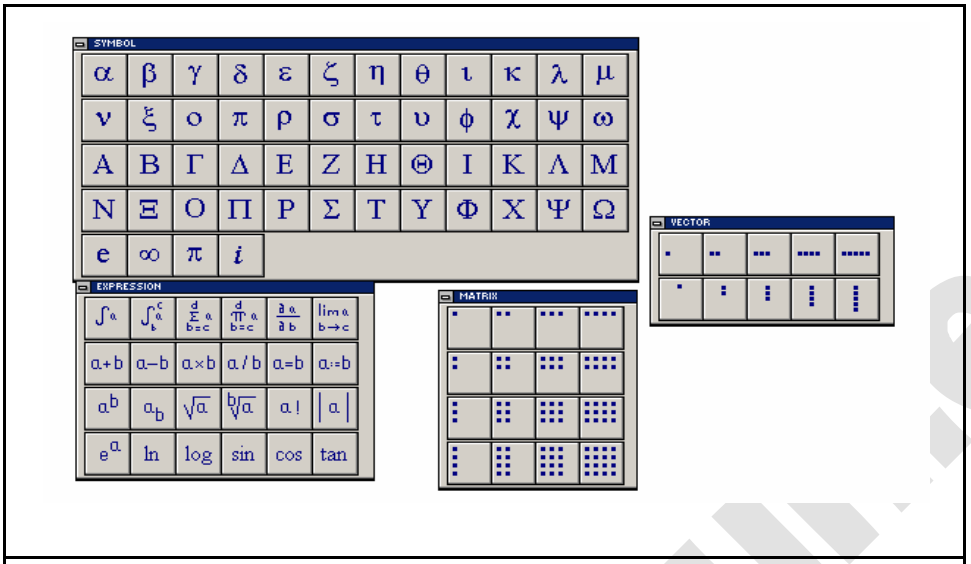


Figura1. Tipos de paletas disponibles en Maple

Para cada tipo de región se puede elegir un color y tipo de letra diferente, con objeto de distinguirla claramente de las demás. Con el menú ***Format/Styles*** se pueden controlar estas opciones. Antes de empezar a trabajar puede ser conveniente establecer unos formatos para cada tipo de región. De esta manera se abre la caja de diálogo mostrada en la figura 2; en ella se pueden modificar todos los estilos de las tres zonas o regiones.

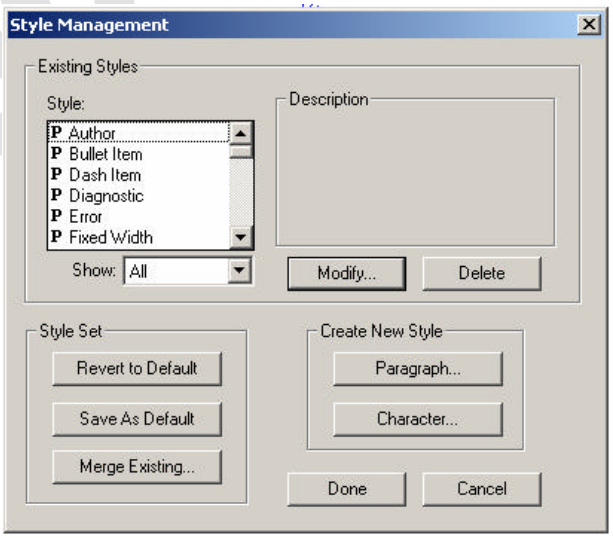


Figura 2. Selección del tipo de letra para las diferentes regiones.

Existen también los llamados ***grupos de regiones***. El agrupar varias regiones tiene ciertas ventajas, como por ejemplo el poder ejecutar conjuntamente todas las regiones de entrada del grupo pulsando la tecla ***Intro***. Para ver y distinguir los grupos, se pueden hacer aparecer unas *líneas de separación* en la pantalla con el comando ***View/Show Group Ranges***.

En Maple existen comandos para partir un grupo y para unir dos grupos contiguos (comando ***Edit/ Split or Join Group***). Si de este menú elegimos ***Split Execution Group***, se parte el grupo por encima de la línea en la que está el cursor. Si elegimos ***Join Execution Group***, se une el grupo en el que está el cursor con el grupo posterior. Si lo

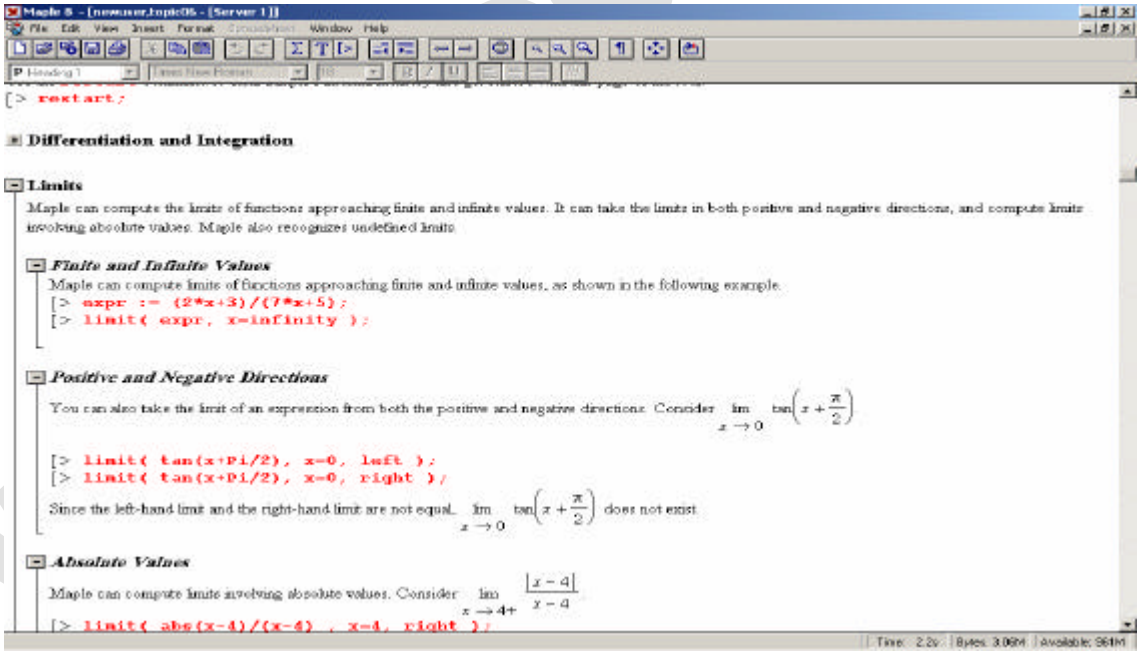
que se quiere es introducir un nuevo grupo entre los ya existentes, se posiciona el cursor en el lugar en el que se quiere insertar el grupo, se clica en ***Insert/Execution Group*** y se indica si se quiere insertar antes o después del cursor.

En Maple hay que distinguir entre pulsar ***Intro*** y ***Mayus+Intro***. En el primer caso se envía a evaluar directamente el comando introducido (si tiene el carácter de terminación: punto y coma o dos puntos) y pasa a la zona de entrada del grupo siguiente (si no existe, la crea). La segunda supone que el usuario va a seguir escribiendo otro comando del mismo grupo en la línea siguiente y amplía la zona de entrada del grupo actual en el que está el cursor. Recuerde que Maple permite escribir varios comandos en una misma línea (cada uno con su carácter de terminación). Realice algunas pruebas.

Maple permite interrumpir los cálculos, una vez iniciados: si el usario decide cancelar un comando que ha comenzado ya a ejecutarse, basta clicar sobre el botón ***Stop*** en la barra de herramientas de la ventana principal, o pulsar las teclas ***Control+Pausa***.

Para escribir texto se clica en el botón —señalado a la derecha— o se selecciona ***Insert/Text Input***. Sobre el texto pueden aplicarse diferentes estilos de manera análoga a como se hace en Word. Para volver introducir sentencias en un nuevo grupo de ejecución hay que clicar el botón —situado a la derecha—. Si únicamente se quiere introducir un comentario que no se tenga en cuenta al ejecutar la hoja de trabajo, se escribe el símbolo '#' y el programa ignora el resto de la linea a partir de dicho símbolo.

Se pueden introducir secciones y subsecciones con ***Insert/Section*** (***Subsection***) y presentar la hoja de trabajo de formas similares a la que se muestra a continuación:



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***) o la marca (***Bookmark***), 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 otra parte, existen varias formas de salir de Maple: se pueden teclear los comandos ***quit***, ***done***, y ***stop***, que cierran la hoja de trabajo; se puede utilizar el menú ***File/Exit*** o simplemente teclear ***Alt+F4*** para salir del programa.

2.3.2. Edición de hojas de trabajo

Toda la hoja de trabajo es accesible y puede ser editada. Esto lleva a que un mismo comando pueda ejecutarse en momentos diferentes con estados internos muy diferentes, y dar por ello resultados completamente distintos. Es importante tener esto en cuenta para no verse implicado en efectos completamente imprevisibles. Para eliminar efectos de este tipo, puede ser útil recalcular completamente la hoja de trabajo, empezando desde el primer comando. Esto se hace desde el menú ***Edit*** con la opción ***Execute Worksheet***. Cuando se recalcula una hoja de trabajo, los gráficos se recalculan. Antes de ejecutar el comando ***Execute Worksheet*** conviene cerciorarse de que en el menú ***File/ Preferences/ I/O Display*** esté seleccionada la opción ***Replace Output***. Este modo será visto con más detalle en la sección siguiente.

2.3.3. Modos de trabajo

Maple dispone de varios ***modos*** de trabajo, que se seleccionan en el menú ***File/ Preferences/ I/O Display***. Dichos modos son los siguientes:

– ***File/ Preferences/ I/O Display Replace Output***. Si está activado, cada resultado o salida sustituye al anterior en la región de salida correspondiente. 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.

– ***File/ Preferences/ I/O Display Insert Mode***. 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.

Lo ordinario es trabajar con la opción ***Replace Output*** activada.

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 la 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 de cerrar la hoja de trabajo. Recuérdese que con el comando ***restart*** se puede volver en cualquier momento al estado inicial, anulando todas las definiciones y cambios que se hayan hecho en el espacio de trabajo.

A veces puede ser conveniente eliminar todas o parte de las regiones de salida. Esto se logra con ***Edit/Remove Output/From Woeksheet*** 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***. Este fichero no es un fichero de texto, y por tanto no es visible ni editable con ***Notepad*** o ***Word***. Para guardar un *fichero de texto* con la misma información del fichero ****.mws*** hay que elegir ***Save As***, y en la caja de diálogo que se abre elegir ***Maple Text*** en la lista desplegable ***Save File As Type***. En este lista encontraremos otros tipos de formatos como son HTML, RTF, LaTeX y XML.

2.4. OBJETOS EN MAPLE 8

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 más importantes de estos objetos.

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(%);
```

4.70997594667669698935310887254386010986 8

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

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);
```

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

Maple permite una gran libertad para definir nombres de variables. Se puede crear una nueva variable en cualquier momento. A diferencia de C y de otros lenguajes, no se declaran previamente. Tampoco tienen un tipo fijo: el tipo de una misma variable puede cambiar varias veces a lo largo de la sesión. No existe límite práctico en el número de caracteres del nombre (sí que existe un límite, pero es del orden de 500). En los nombres de las variables se distinguen las mayúsculas y las minúsculas. Los nombres asignados en Maple pueden incluir cualquier carácter alfanumérico así como los guiones y guiones bajos, pero *no pueden empezar por un número*. Asimismo es conveniente no empezar un nombre por un guión bajo ya que el programa usa ese tipo de nombres para su clasificación interna. Nombres válidos son "polinomio", "cadena_caracter". Inválidos serían "2_fase" (empieza por número) y "x&y" (porque & no es un carácter alfanumérico).

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. Además el comando length se utiliza para saber la longitud de nuestra cadena. Así:

```
> 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"

```
> length(cadena);
```

12

2.4.3. Operador de concatenación (||)

Una doble barra separando dos nombres, o un nombre y número, actúa como *operador de concatenación*, esto es, el resultado es un único nombre con la doble barra eliminada. Véase un ejemplo:

```
> a||1, diga||33;
```

$a1, diga33$

Nótese que en anteriores versiones de Maple (en Maple V Release 5 por ejemplo), el operador de concatenación era el punto (.). Las precauciones que requería este operador a la hora de no confundirlo con extensiones de ficheros por ejemplo, llevaron a su cambio por la doble barra (operador actual).

2.4.4. 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;
```

2.4.5. 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.

2.4.6. Secuencias o sucesiones

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. 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 apóstrofos 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;
```

$$a1, a2, a3, a4$$

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***.

2.4.7. Conjuntos (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 { }. 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. Si se pasa un *set* como argumento a la función ***whattype*** la respuesta es ***set***.

2.4.8. Listas (lists)

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*. La respuesta de la función ***whattype*** cuando se le pasa una lista como argumento es ***list***.

Los operadores ***union***, ***intersect*** y ***minus*** no operan sobre listas. 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

2.4.9. 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. Además, 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
```

$$potencias := \mathrm{array}(1 .. 3, 1 .. 3, [\ ])$$

```
> 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 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));
```

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

2.4.10. 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.

2.4.11. Hojas de cálculo (Spreadsheets)

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.*** 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.

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:

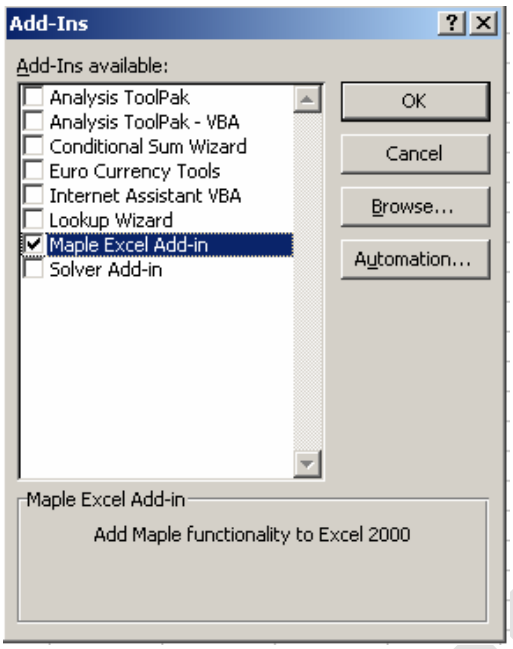
	A	B	C	D
1	1			
2	2			
3	3			
4	4			

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 cambiarán de color. Para recalcular toda la hoja se utiliza el botón.

En la barra de menús disponemos de un menú llamado ***Spreadsheet*** en el que podemos realizar también estas operaciones de relleno sin necesidad de pulsar sobre los iconos comentados. 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 y ahí veremos una opción llamada *Maple Excel Add-In* que tendremos que seleccionar.



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 que nos ayuda a iniciarnos en el trabajo de Excel con Maple.

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

Se trata de funciones muy utilizadas con las listas y con bs 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

La función ***sort*** se aplica a listas, no a secuencias o *sets*. Su objetivo es ordenar la lista de acuerdo con un determinado criterio, normalmente alfabético o numérico (***sort*** se aplica también a polinomios, y entonces hay otros posibles criterios de ordenación que veremos más adelante en la sección correspondiente). En el caso de las listas el comando ***sort*** tiene dos opciones principales: *lexorder* (orden alfabético, por defecto) y

length (según la longitud de cada elemento de la lista). Por ejemplo, las siguientes sentencias convierten un *set* en una lista (pasando por una secuencia, que es el resultado de la función ***op***) y luego la ordenan alfabéticamente y, posteriormente, por longitud:

```
> list1:=[op(set1)];
```

$$list1 := [x, v, t, yuv, zxc, us]$$

```
> sort(list1,lexorder);
```

$$[t, us, v, x, yuv, zxc]$$

```
> sort(list1,length);
```

$$[x, v, t, us, yuv, zxc]$$

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; #ejemplo con un polinomio, cambiando el signo
al segundo 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);
```

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

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);
```

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

2.6. LOS COMANDOS ZIP, MAP Y CONVERT

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)];
```

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

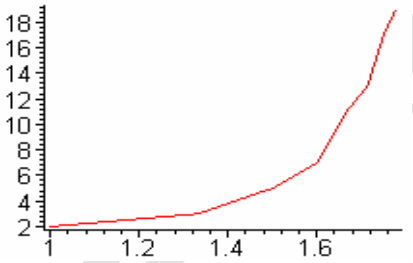
```
> Y:=[seq(ithprime(i), i=1..8)];
```

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

```
> 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);
```



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.

Finalmente vamos a ver un último comando que puede llegar a resultar muy útil al trabajar con distintos tipos de datos: el comando ***convert***. Este permite convertir datos de un tipo a otro distinto. Así, por ejemplo, permite realizar conversiones entre listas y sets. Recuérdese que cualquier secuencia puede convertirse en un set encerrándola entre llaves { } o en una lista encerrándola entre corchetes [ ]. Recíprocamente, cualquier set o lista se puede convertir en una secuencia por medio de la función ***op***.

```
> lista3:=[perro,gato,caballo];
```

$$lista3 := [perro, gato, caballo]$$

```
> convert (lista3,set);
```

$$\{perro, gato, caballo\}$$

Este último comando tiene muchas más aplicaciones, sobre todo en el trabajo con expresiones que se verán más adelante en el apartado titulado "*Operaciones con expresiones*".

2.7. VARIABLES EVALUADAS Y NO-EVALUADAS

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
```

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

La respuesta anterior se explica por sí misma: es una *secuencia* con las dos soluciones de la ecuación dada.

En Maple una variable puede *tener asignado su propio nombre* (es decir estar sin asignar o *unassigned*), *otro nombre diferente* o un *valor numérico*. Considérese el siguiente ejemplo:

```
> polinomio := 9*x**3 - 37*x**2 + 47*x - 19;
```

$$polinomio := 9x^3 - 37x^2 + 47x - 19$$

Ahora se van a calcular las raíces de este polinomio, con su orden de multiplicidad correspondiente. Obsérvense los resultados de los siguientes comandos con los comentarios incluidos con cada comando:

```
> roots(polinomio);  # cálculo de las raíces (una simple y otra doble)
```

$$\left[\left[\frac{19}{9}, 1\right], [1, 2]\right]$$

```
> factor(polinomio);  # factorización del polinomio
```

$$(9x - 19)(-1 + x)^2$$

```
> subs(x=19/9, polinomio);  #comprobar la raíz simple
```

$$0$$

```
> x; polinomio;  # no se ha hecho una asignación de x o polinomio
```

$$x$$

$$9x^3 - 37x^2 + 47x - 19$$

La función ***subs*** realiza una substitución de la variable **x** en **polinomio**, pero no asigna ese valor a la variable **x**. La siguiente sentencia sí realiza esa asignación:

```
> x:= 19/9; polinomio; #ahora sí se hace una asignación a x y
polinomio
```

$$x := \frac{19}{9}$$

$$0$$

Ahora la variable **x** tiene asignado un valor numérico. Véase el siguiente cambio de asignación a otro nombre de variable:

```
> x:= variable; polinomio;
```

$$x := variable$$

$$9\,variable^3 - 37\,variable^2 + 47\,variable - 19$$

```
> variable := 10; x; polinomio; # cambio indirecto de asignación
```

$$variable := 10$$

$$10$$

$$5751$$

Para que **x** vuelva a estar asignada a su propio nombre (en otras palabras, para que esté *desasignada*) se le asigna su nombre entre apóstrofos:

```
> x := 'x'; polinomio; # para desasignar x dándole su propio nombre
```

$$x := x$$

$$9x^3 - 37x^2 + 47x - 19$$

Los apóstrofos **'x'** hacen que ***x*** se evalúe a su propio nombre, suspendiendo la evaluación al valor anterior que tenía asignado. 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

$$a1 := a \,||\, i$$

$$a1 := a1$$

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');
```

El siguiente ejemplo (se omiten los resultados del programa) muestra cómo se puede saber si una variable está asignada o no:

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

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

Existen también las funciones ***assign*** y ***unassign***. La primera de ellas, que tiene la forma ***assign(name, expression);*** equivale a ***name := expression;*** 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. Por su parte, la función ***unassign*** puede desasignar varias variables a la vez. Considérese el siguiente ejemplo, en el que se comienza definiendo un conjunto de ecuaciones y otro de variables:

```
> ecs := {x + y = a, b*x -1/3*y = c}; variables := {x, y};
```

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

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

A continuación se resuelve el conjunto de ecuaciones respecto al de variables, para hallar un conjunto de soluciones[1]:

[1] Es lógico que tanto las ecuaciones como las variables sean *sets* o conjuntos, pues no es importante el orden, ni tiene sentido que haya elementos repetidos.

```
> soluciones := solve(ecs, variables);
```

$$soluciones := \left\{ y = -3\,\frac{-b\,a+c}{3\,b+1},\ x = \frac{3\,c+a}{3\,b+1} \right\}$$

El resultado anterior no hace que se asignen las correspondientes expresiones a x e y. Para hacer esta asignación hay que utilizar la función ***assign*** en la forma:

```
> x, y; assign(soluciones); x, y; # para que x e y se asignen
realmente
```

$$x, y$$

$$\frac{3\,c+a}{3\,b+1},\ -3\,\frac{-b\,a+c}{3\,b+1}$$

```
> unassign('x', 'y'); x, y;  # si se desea desasignar x e 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 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;
```

$$a := b$$

$$b := c$$

$$c := 3$$

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

$$3$$

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 apóstrofos*, 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 apóstrofos 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
```

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

2.8.1. 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:

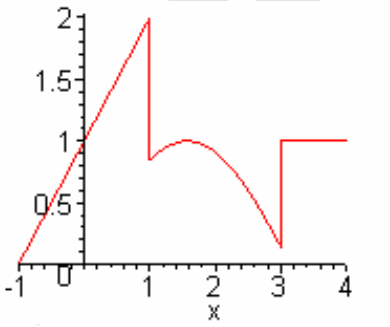
```
> f:=x->piecewise(x<=1, x+1,1<x and x<3,sin(x),1);
```

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

```
> 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);
```



2.8.2. 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:

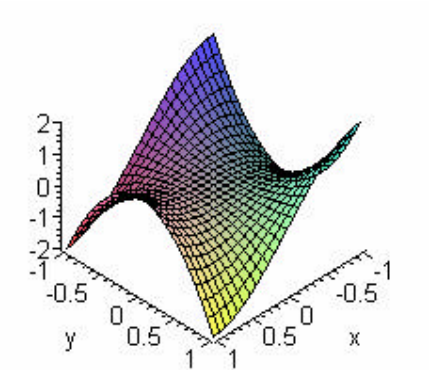
```
> f:=(x,y)->x^3-3*x*y^2;
```

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

```
> 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);
```



2.8.3. 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
```

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

```
> 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
```

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

2.8.4. 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));
```

$$n := n$$

$$\zeta(n)$$

$$\zeta(2)$$

$$subs$$

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

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.***

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

$$0.8819212643$$

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:=5+3*I; Re(num);Im(num);conjugate(num);
```

$$num := 5 + 3\,I$$

$$5$$

$$3$$

$$5 - 3\,I$$

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

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

$$\sqrt{34}$$

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

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

```
> expr:=sin(a+b*I);
```

$$expr := \sin(a + b\,I)$$

```
> evalc(expr);
```

$$\sin(a)\cosh(b) + \cos(a)\sinh(b)\,I$$

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:

```
> p1 := -3*x + 7*x^2 -3*x^3 + 7*x^4;
```

$$p1 := -3x + 7x^2 - 3x^3 + 7x^4$$

Existen comandos para preguntar por el tipo de estructura, grado, etc. Ejecute los siguientes comandos para preguntar si p1 es un polinomio, cuál es el coeficiente del término de mayor grado y qué grado tiene dicho polinomio:

```
> type( p1, 'polynom' );
```

$$\text{true}$$

```
> lcoeff(p1), degree(p1);
```

$$7, 4$$

Se pueden realizar operaciones de suma y producto de polinomios como con cualquier otro tipo de variables:

```
> p2 := 5*x^5 + 3*x^3 + x^2 -2*x + 1;
```

$$p2 := 5x^5 + 3x^3 + x^2 - 2x + 1$$

```
> 2*p1 + 4*p2 + 3;
```

$$-14x + 18x^2 + 6x^3 + 14x^4 + 20x^5 + 7$$

```
> p1 * p2;
```

$$\left(-3x + 7x^2 - 3x^3 + 7x^4\right)\left(5x^5 + 3x^3 + x^2 - 2x + 1\right)$$

```
> expand(%);
```

$$-17x^6 + 11x^4 - 20x^3 + 13x^2 - 3x + 56x^7 + 4x^5 - 15x^8 + 35x^9$$

En el resultado anterior puede verse que Maple no ordena los términos de modo automático. Para que lo haga, hay que utilizar el comando sort:

```
> sort(%);
```

$$35x^9 - 15x^8 + 56x^7 - 17x^6 + 4x^5 + 11x^4 - 20x^3 + 13x^2 - 3x$$

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
```

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

Maple dispone de numerosas funciones para manipular polinomios. Para utilizar las funciones coeff y degree, el polinomio debe estar en forma agrupada (collected form). A continuación se muestran algunos otros ejemplos, sin los resultados:

```
> 'p1' = p1, 'p2' = p2;
> coeff( p2, x^3 );  # para determinar el coeficiente de x^3 en p2
> coeff( p2, x, 3 );  # equivalente a la anterior
> coeffs(p2, x);
> coeffs(p2, x, 'powers'); powers;
```

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:

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

$$q := \frac{5}{7}x + \frac{15}{49}$$

$$-\frac{53}{49}x^3 + x^2 - \frac{53}{49}x + 1$$

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

true

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

$$-\frac{53}{49}x^3 + x^2 - \frac{53}{49}x + 1$$

$$\frac{5}{7}x + \frac{15}{49}$$

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

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

false

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

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

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.

```
> poli := expand(p1*p2);
> roots(poli); #devuelve las raices y su multiplicidad
> factor(poli);
```

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;
```

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

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

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

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

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

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

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

Para ordenar según las potencias de x:

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

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

o según las potencias de y:

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

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

Otros ejemplos de manipulación de polinomios de dos variables son los siguientes (no se incluyen los resultados):

```
> coeff(poli, x^3), coeff(poli, x, 3);
> coeffs(poli, x, 'powers'); powers;
```

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:

```
> 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 (no se incluyen ya los resultados):

```
> 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;
> racexp := f/g;
> normal(racexp);
```

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);
```

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

```
> 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. 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.*

Maple nos ofrece también la posibilidad de controlar el número de soluciones mediante el parámetro **maxsols** en el caso de que existan múltiples. Conviene comentar que si Maple no es capaz de encontrar la solución o ésta no existe, el programa devuelve el valor NULL. Por otra parte, el comando de resolución espera que se le mande como argumento una ecuación. Si lo que le introducimos es una expresión, lo interpretará de la forma expresión=0. Finalmente, hay que indicar a Maple que incógnitas hay en la ecuación. Si no lo hacemos, Maple la resolverá para todas ellas.

```
> solve({x+y=0},{x});
```

$$\{x = -y\}$$

```
> solve({x+y=0});
```

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

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
```

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

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, x=Pi/2);  # devuelve el límite cuando x tiende a
Pi/2.
```

$$0$$

```
> limit((-x^2+x+1)/(x+4), x=infinity);
```

$$-\infty$$

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

$$undefined$$

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

$$\infty$$

$$-\infty$$

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, en vez de la forma Maple Notation empleada en los ejemplos anteriores. El último ejemplo, empleando la notación Standard Math:

> $\lim_{x \to \left(\frac{\pi}{2}\right)^-} \tan(x)$; $\lim_{x \to \left(\frac{\pi}{2}\right)^+} \tan(x)$

$$\infty$$

$$-\infty$$

Introduzca ahora cualquier límite mediante la notación Standard Math. Para ello clique sobre el icono anterior y abra las paletas Symbol Palette y Expression Palette con View/Palettes. Clique sobre el icono de límite y obtendrá:

> $\lim_{? \to ?} ?$

Ahora sólo tiene que sustituir las interrogaciones ? para construir 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(tan(x),x=Pi/2,left);
```

$$\lim_{x \to \left(\frac{\pi}{2}\right)^-} \tan(x)$$

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

$$\infty$$

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

$$\lim_{x \to \left(\frac{\pi}{2}\right)^-} \tan(x) = \infty$$

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
```

$$3\,x^2$$

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

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

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

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

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

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

```
> 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
```

$$2\,x\,y + 3\,y^2\,x^2$$

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

$$2 + 12\,x\,y$$

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:

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

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

```
> derivada:=diff(f,x); #expresión
```

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

```
> f_prima:=unapply(derivada,x); #ahora f_prima es una función
```

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

```
> f_prima(3);
```

$$-2\,\mathbf{e}^{(-6)}$$

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^3+2*x,x)=diff(x^3+2*x,x);
```

$$\frac{d}{dx}(x^3+2x)=3x^2+2$$

```
> Diff(x^3+2*x,x); value(%);
```

$$\frac{d}{dx}(x^3+2x)$$

$$3x^2+2$$

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
```

$$der := (x,y,z) \to x\,\mathbf{e}^{(x\,y)}+x\,z$$

Al clicar con el botón derecho en el *output* o la salida de cualquier expresión, le aparecerá un ***menú contextual***. Si elige la opción ***Differenciate***, Maple escribirá y ejecutará la derivada automáticamente.

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(2*x*exp(x^2), x);
```

$$e^{(x^2)}$$

```
> int(sin(y)*cos(y), y);
```

$$-\frac{1}{2}\cos(y)^2$$

```
> int(1/exp(x^2)+x, x);
```

$$\frac{1}{2}\sqrt{\pi}\ \mathrm{erf}(x)+\frac{1}{2}x^2$$

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(1/x, x=2..4);
```

$$\ln(4)-\ln(2)$$

```
>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. 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((x^2*y^3),x),y); # integra respecto de x y luego respecto de
y.
```

$$\frac{1}{12}x^3y^4$$

```
> int(int(int(x^2*y^2*z^2, x=1..2), y=1..2), z=1..2); # realiza la
integral definida respecto de las tres variables
```

$$\frac{343}{27}$$

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(1/(x^2+x),x=2..infinity)=int(1/(x^2+x),x=2..infinity);
```

$$\int_2^{\infty}\frac{1}{x^2+x}\,dx=-\ln(2)+\ln(3)$$

Puede también introducir integrales, tanto definidas como indefinidas, utilizando la notación Standard Math.

Por otra parte, al clicar sobre la salida de cualquier expresión, el menú contextual sólo le dará la posibilidad de realizar integrales indefinidas mediante la opción Integrate. Una manera de conseguir una integral definida es: primero construirla clicando en el menú contextual Constructions/Definite Integral; tras fijar los extremos, vuelva al clicar con el botón derecho en Evaluate.

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);
```

$$f1 := x \to x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7$$

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:

```
> exp1:=exp(x)*sin(x);
```

$$exp1 := e^{x} \sin(x)$$

```
> tay_1:=taylor(exp1,x,5); tay_2:=taylor(exp1,x,8);
```

$$tay\_1 := x + x^2 + \frac{1}{3}x^3 + O(x^5)$$

$$tay\_2 := x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7 + O(x^8)$$

```
> aprox_1:=convert(tay_1,polynom); aprox_2:=convert(tay_2,polynom);
```

$$aprox\_1 := x + x^2 + \frac{1}{3}x^3$$

$$aprox\_2 := x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7$$

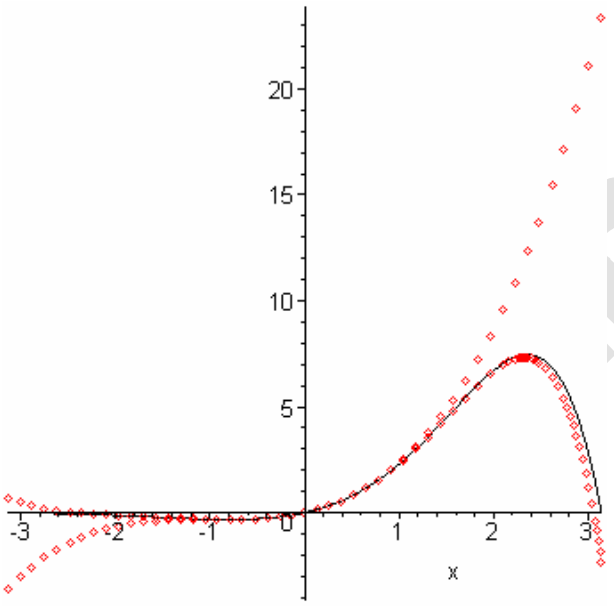
```
> f1:=unapply(aprox_1,x); f2:=unapply(aprox_2,x);
```

$$f1 := x \to x + x^2 + \frac{1}{3}x^3$$

$$f2 := x \to x + x^2 + \frac{1}{3}x^3 - \frac{1}{30}x^5 - \frac{1}{90}x^6 - \frac{1}{630}x^7$$

```
> plot1:=plot(exp1,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):
          Warning, the name changecoords has been redefined

>display({plot1,plot2,plot3});
```



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(%);
> y:=unapply(y(x),x);
```

$$y := x \rightarrow \mathbf{e}^{(a\,x)}$$

```
> y(0);
```

$$1$$

Ahora realizaremos otro ejemplo f''(x)+f(x)=sin(x)+cos(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;
> eqn:=diff(f(x),x$2)+f(x)=cos(x)+sin(x); #la ecuación diferencial
```

$$eqn := \left(\frac{d^2}{dx^2} f(x)\right) + f(x) = \cos(x) + \sin(x)$$

```
> 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);
```

$$sol := x \rightarrow \cos(x) + \frac{1}{2}\sin(x)\,x - \frac{1}{2}\cos(x)\,x$$

```
> sol(0); #comprobamos que cumple las condiciones iniciales
```

$$1$$

```
> sol_prima:=D(sol); sol_prima(1);
```

$$sol\_prima := x \rightarrow \frac{1}{2}\sin(x) + \frac{1}{2}\cos(x)\,x + \frac{1}{2}\sin(x)\,x - \frac{1}{2}\cos(x)$$

$$0$$

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);
```

$$\left[x = 0., y(x) = 3.64000000000000013, \frac{d}{dx} y(x) = -7.18642469929130012\right]$$

```
> sol2(2);
```

$$\left[x = 2., y(x) = 1.34349999999999969, \frac{d}{dx} y(x) = 2.42203818337829935\right]$$

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.

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);
```

$$\frac{x^3}{y\,(z+1)}+\frac{4\,x^2}{y\,(z+1)}+\frac{3\,x}{y\,(z+1)}$$

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
```

$$\cos(2\,x)=2\,\cos(x)^2-1$$

```
> cos(x*(y+z)): %=expand(%); #función más complicada
```

$$\cos(x\,(y+z))=\cos(x\,y)\,\cos(x\,z)-\sin(x\,y)\,\sin(x\,z)$$

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)*.

```
> restart;
> ln(x/y): %=expand(%);  #no conoce los signos
```

$$\ln\left(\frac{x}{y}\right)=\ln\left(\frac{x}{y}\right)$$

```
> 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)*(y+z);
```

$$poli:=(x+1)\,(y+z)$$

```
> eval(poli)=expand(poli);
```

$$(x+1)\,(y+z)=x\,y+x\,z+y+z$$

```
> eval(poli)=expand(poli,x+1); #indicando lo que no queremos expandir
```

$$(x+1)\,(y+z)=(x+1)\,y+(x+1)\,z$$

4.1.2. Función combine

Es el comando que realiza la tarea inversa a la que hace ***expand***. La función ***combine*** combina varias expresiones para conseguir una más compacta o reducida. Para ello, en muchas ocasiones, 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.

Al utilizar ***combine*** es necesario 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***, y ***Psi*** (función poligamma). Las reglas de combinación que se aplican en cada caso son las siguientes:

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
```

Veamos a continuación algunos ejemplos prácticos del uso de la función ***combine***:

```
> 4*sin(x)^3: %=combine(%,trig);
```

$$4\sin(x)^3 = -\sin(3x) + 3\sin(x)$$

```
> exp(x)*exp(x+y): %=combine(%,exp);
```

$$\mathbf{e}^{x}\,\mathbf{e}^{(x+y)} = \mathbf{e}^{(2x+y)}$$

```
> x^y/(x^(x+2*y)): %=combine(%,powers);
```

$$\frac{x^{y}}{x^{(x+2y)}} = x^{(-y-x)}$$

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***.

```
> 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
```

$$\ln(x\sim) - \ln(y\sim) = \ln\left(\frac{x\sim}{y\sim}\right)$$

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*,…), 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.

Compruebe la salida de este primer ejemplo (función trigonométrica) con la obtenida en el apartado anterior mediante la función combine: el resultado sigue siendo el mismo, pero expresado de forma diferente.

```
> 4*sin(x)^3: %=simplify(%);
```

$$4\sin(x)^3 = -4\sin(x)\,(-1+\cos(x)^2)$$

```
> exp(x)*exp(y)+cos(x)^2+sin(x)^2: % = simplify(%);
```

$$\mathbf{e}^x\,\mathbf{e}^y + \cos(x)^2 + \sin(x)^2 = \mathbf{e}^{(x+y)} + 1$$

```
> exp(x)*exp(y)+cos(x)^2+sin(x)^2: % = simplify(%, trig); #solo se
aplica la trigonométrica
```

$$\mathbf{e}^x\,\mathbf{e}^y + \cos(x)^2 + \sin(x)^2 = \mathbf{e}^x\,\mathbf{e}^y + 1$$

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);
```

$$x\,y + y\,z + y + 1$$

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));
```

$$\frac{1}{f(x)+1}$$

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
```

$$\frac{x+y}{x^2-2xy+y^2}$$

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;
```

$$pol := x^5-x^4-x^3-x^2-2x+2$$

```
> factor(pol); #no consigue en los enteros
```

$$x^5-x^4-x^3-x^2-2x+2$$

```
> 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);
```

$$\frac{(11)\ (41)}{(2)^2\ (3)}$$

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
fraction	GAMMA	hex	horner	hostfile	hypergeom
lessthan	lessequal	list	listlist	ln	matrix
metric	mod2	multiset	name	octal	parfrac
polar	polynom	radians	radical	rationa l	ratpoly
RootOf	series	set	sincos	sqrfree	tan
trig	vector				

Considérese el ejemplo siguiente:

```
> (x^2-x-3)/(x^3-x^2): convert(%, parfrac, x);
```

$$3\frac{1}{x^2}+4\frac{1}{x}-3\frac{1}{x-1}$$

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 o incluso para cambiar de base un número:

```
> des:=taylor(sin(x),x,8);
```

$$des := x-\frac{1}{6}x^3+\frac{1}{120}x^5-\frac{1}{5040}x^7+O(x^8)$$

```
> des:=convert(des,polynom); #convirtiendo desarrollos en polinomios
```

$$des := x-\frac{1}{6}x^3+\frac{1}{120}x^5-\frac{1}{5040}x^7$$

```
> cos(x): % = convert(%, exp); # tercer argumento opcional
```

$$\cos(x)=\frac{1}{2}e^{(Ix)}+\frac{1}{2}\frac{1}{e^{(Ix)}}$$

```
> sinh(x): % = convert(%, exp);
```

$$\sinh(x) = \frac{1}{2}e^{x} - \frac{1}{2}\frac{1}{e^{x}}$$

```
> lista:=[1,2,3,4];
```

$$lista := [1, 2, 3, 4]$$

```
> conjunto:=convert (lista,set); #cambiando de tipo de objeto
```

$$conjunto := \{1, 2, 3, 4\}$$

```
> num_dec:=46;
```

$$num\_dec := 46$$

```
> num_bin:=convert(num_dec,binary); num_hex:=convert(num_dec,hex);
```

$$num\_bin := 101110$$

$$num\_hex := 2E$$

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;
```

$$p := y^3 + y^2x^2 + x^3 + x^5$$

```
> sort(p, [x,y]); # ordena según la suma de exponentes
```

$$x^5 + x^2y^2 + x^3 + y^3$$

```
> sort(p, y); # ordena según el exponente de y
```

$$y^3 + x^2y^2 + x^5 + x^3$$

```
> sort(p,[x,y], plex); # ordena alfabéticamente
```

$$x^5 + x^3 + x^2y^2 + y^3$$

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)
```

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. Más adelante se tratan más a fondo algunos de especial relevancia.

- **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.
- **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.
- **powseries**: comandos para crear y manipular series de potencias representadas de la forma general.
- **VectorCalculus**: Cálculo multivariable y vectorial.

5.2. 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.

En este manual nos vamos a centrar en algunas de las funciones del subpaquete Calculus 1. Éste 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, ApproximateInt, ArcLength, Asymptotes, Clear, CriticalPoints, DerivativePlot, ExtremePoints, FunctionAverage, FunctionChart, GetMessage, GetNumProblems, GetProblem, Hint, InflectionPoints, Integrand, InversePlot, MeanValueTheorem, NewtonQuotient, NewtonsMethod, PointInterpolation, RiemannSum, RollesTheorem , Roots, Rule, Show, ShowIncomplete, ShowSteps,*

A continuación veremos algunas de las funciones más interesantes.

♦ **Rule:** Aplica una determinada regla a un problema de Cálculo 1. La sintaxis de la función es Rule[**rule**](**expr**, **opn**), donde rule es la regla que se quiere aplicar, expr es la expresión algebraica del problema y opn especifica el tipo de problema. Las reglas que se pueden aplicar son las siguientes:

- Reglas de diferenciación:

chain	(f(g(x)))' = f'(g(x))*g'(x)
constant	c' = 0
constantmultiple	(c*f)' = c*f'

difference	(f-g)' = f'-g'
identity	x' = 1
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:

constant	Int(c, x)= c*x
	Int(c, x=a..b) = c*b - c*a
constantmultiple	I(c*f(x)) = c*I(f(x))
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 una 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)

-Reglas de límites:

constant	L(c)= c
constantmultiple	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

Ej:

```
> with(Student[Calculus1]):
> Rule[product](Diff(x^2*sin(x^2), x));
```

$$\frac{d}{dx}(x^2\sin(x^2)) = \left(\frac{d}{dx}(x^2)\right)\sin(x^2) + x^2\left(\frac{d}{dx}\sin(x^2)\right)$$

```
> Rule[chain](%);
```

$$\frac{d}{dx}(x^2\sin(x^2)) = \left(\frac{d}{dx}(x^2)\right)\sin(x^2) + x^2\left(\frac{d}{d\_X}\sin(\_X)\right)\Big|_{\_X = x^2}\left(\frac{d}{dx}(x^2)\right)$$

```
> Rule[sin](%);
```

$$\frac{d}{dx}(x^2\sin(x^2)) = \left(\frac{d}{dx}(x^2)\right)\sin(x^2) + x^2\cos(x^2)\left(\frac{d}{dx}(x^2)\right)$$

- **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. Veamos unos ejemplos:

```
> with(Student[Calculus1]):
> 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[%](%%);
```

$$\lim_{n\to\infty}\frac{2^n-3^n}{\ln(n)}=\lim_{n\to\infty}\left(2^n\ln(2)-3^n\ln(3)\right)n$$

```
> Hint(%);
```

$$[product]$$

```
> Rule[%](%%);
```

$$\lim_{n\to\infty}\frac{2^n-3^n}{\ln(n)}=\left(\lim_{n\to\infty}2^n\ln(2)-3^n\ln(3)\right)\left(\lim_{n\to\infty}n\right)$$

```
> Hint(%);
```

$$[identity]$$

```
> Rule[%](%%);
```

$$\lim_{n\to\infty}\frac{2^n-3^n}{\ln(n)}=\left(\lim_{n\to\infty}2^n\ln(2)-3^n\ln(3)\right)\infty$$

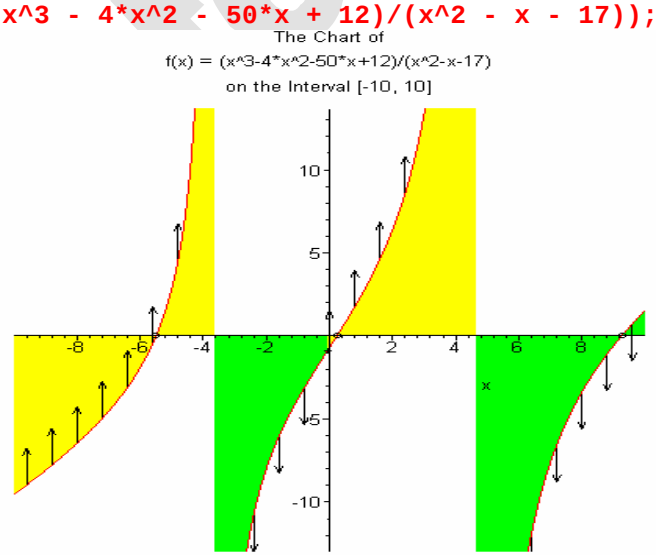
```
> limit((2^n-3^n)/(ln(n)),n=infinity);
```

$$-\infty$$

En este ejemplo queda claro que el camino que propone no siempre es el mejor ya que este límite se podía haber resuelto de una manera mucho más sencilla. (Examen Cálculo 1, Febrero curso 97-98).

- **FunctionChart:** Esta función nos representa la función con datos de interés para su estudio. La sintaxis es FunctionChart(**f(x)**, **x**, opts), donde x es la variable independiente. Ej:

```
> with(Student[Calculus1]):
FunctionChart((x^3 - 4*x^2 - 50*x + 12)/(x^2 - x - 17));
```



5.3. 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);
```

[*Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm, CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation, ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, CreatePermutation, CrossProduct, DeleteColumn, DeleteRow, Determinant, 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, MatrixInverse, MatrixMatrixMultiply, MatrixNorm, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor, Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent, Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank, 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 linalg, 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);
```

5.3.1. 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 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);
```

$$[0 \quad 1]$$

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);
```

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

5.3.2. Función evalm y operador matricial &*

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)
```

$$A := \begin{bmatrix} 1 & 1 & 1 \\ x & x & x \\ x^2 & x^2 & x^2 \end{bmatrix}$$

```
> B:=Matrix(3,3,1);
```

$$B := \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
> A+B;
```

$$\begin{bmatrix} 2 & 2 & 2 \\ 1+x & 1+x & 1+x \\ 1+x^2 & 1+x^2 & 1+x^2 \end{bmatrix}$$

```
> 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 &*
```

$$\begin{bmatrix} 1+x+x^2 & 1+x+x^2 & 1+x+x^2 \\ 1+x+x^2 & 1+x+x^2 & 1+x+x^2 \\ 1+x+x^2 & 1+x+x^2 & 1+x+x^2 \end{bmatrix}$$

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 los tamaños) 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.

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);
```

$$\begin{bmatrix} \frac{1}{x} & \frac{1}{x} & \frac{1}{x} \\ 1 & 1 & 1 \\ x & x & x \end{bmatrix}$$

$$\begin{bmatrix} 1+y & 1 & 1 \\ x & x+y & x \\ x^2 & x^2 & x^2+y \end{bmatrix}$$

5.3.3. 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;
```

$$B := \begin{bmatrix} 1 & 1 & 1 \\ x & x & x \\ x^2 & x^2 & x^2 \end{bmatrix}$$

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;
```

$$B_{1,2} := \alpha$$

$$\begin{bmatrix} 1 & \alpha & 1 \\ x & x & x \\ x^2 & x^2 & x^2 \end{bmatrix}$$

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.

5.3.4. 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);
```

$$\begin{bmatrix} 185531 & 487478 & 126008 \\ 87904 & 163117 & 64252 \\ 5476 & 12010 & 4027 \end{bmatrix}$$

5.3.5. 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.

- **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);
```

$$\left[\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \frac{1}{2} \end{bmatrix}\right]$$

```
> RowSpace(A);
```

$$\left[\left[1, 0, \frac{-2 + 3\,a + 2\,c}{-46 + a}\right], \left[0, 1, -\frac{68 + c}{-46 + a}\right]\right]$$

- **Determinant:** Esta función calcula el determinante de una matriz definida de forma numérica o simbólica. La sintaxis es Determinant(**A**, **m**), donde A es la matriz y m(opcional) es el método empleado para el cálculo del determinante, para más información acerca de los distintos métodos dirigirse a la ayuda del comando. Ej:

```
> 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);
```

$$-(2\,a^3\,b\,c - 2\,a\,b^2\,c + 4\,a^2 + 2\,b\,a - 2\,c^2 - 4\,a^2\,b + 2\,a^3\,b + 2\,b^2 - b^2\,a - 2\,a^3 + 2\,c^3 - 2\,b^4\,c + 2\,b^4\,c^2 + 2\,b^4\,a - b^4\,a^2 + 3\,b\,c^2 - 3\,b\,c^3 + b\,c\,a^2 - 2\,b\,a\,c^2 - c^2\,a^2\,b - 4\,b)/(b\,c)$$

- **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'. 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');
```

El efecto de la opción 'implicit'.

$$\begin{bmatrix} 1 \\ \text{RootOf}(\_Z^2 - 2\,\_Z - 1, index = 1) \\ \text{RootOf}(\_Z^2 - 2\,\_Z - 1, index = 2) \end{bmatrix}$$

- **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 'Vector', 'Vector[row]', 'Vector[column]', o 'list'. Por último outpots (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);
```

$$v, e := \begin{bmatrix} 2 \\ 2 \\ -4 \end{bmatrix}, \begin{bmatrix} -2 & -1 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{bmatrix}$$

```
> 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]]);
```

$$B := \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 5 & 10 & 15 \end{bmatrix}$$

Maple calcula el vector ***c*** para cada valor propio de ***A***. Por cada valor propio se devuelve un conjunto de la forma siguiente:

{valor propio, multiplicidad, vectores propios, ...}

donde *multiplicidad* puede estar o no estar presente.

```
> Eigenvectors(B, output='list');
```

$$\left[\left[0, 2, \left\{\begin{bmatrix}-2\\1\\0\end{bmatrix}, \begin{bmatrix}-3\\0\\1\end{bmatrix}\right\}\right], \left[20, 1, \left\{\begin{bmatrix}\frac{1}{2}\\1\\\frac{5}{2}\end{bmatrix}\right\}\right]\right]$$

- **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);
```

$$\begin{bmatrix}-1 & -3 & -6\\0 & -4 & -12\\0 & 0 & -4\end{bmatrix}$$

- **LinearSolve:** Esta función nos devuelve el vector x que satisface A.x = B. La sintaxis es LinearSolve(**A**, **B**, **m**, **t**, **c**, **ip**, **outopts**, **methopts**), donde A y B(opcional) son las matrices del problema, m(opcional) es el método empleado en la resolución ('none', 'solve', 'subs', 'Cholesky', 'LU', 'QR', 'SparseLU', 'SparseDirect' o 'SparseIterative'), t(opcional) es la variable usada en las soluciones, ip(opcional) especifica si la salida sobrescribe el argumento B, outopts(opcional) son las opciones del constructor de la salida y methopts(opcional) son las opciones del método empleado en la resolución. Ej:

```
> 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);
```

$$M := \begin{bmatrix}1 & 1 & 3 & -1 & 0\\1 & 1 & 1 & 1 & 1\\1 & -2 & 1 & -1 & 1\\4 & 1 & 8 & -1 & 0\end{bmatrix}$$

$$\begin{bmatrix}\frac{25}{6}\\\frac{4}{3}\\\frac{-5}{2}\\-2\end{bmatrix}$$

5.4. 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, GCRD, LCLM, MeijerGsols, PDEchangecoords, RiemannPsols, Xchange, Xcommutator, Xgauge, abelsol, adjoint, autonomous, bernoullisol, buildsol, buildsym, canoni, caseplot, casesplit, checkrank, chinisol, clairautsol, constcoeffsols, convertAlg, convertsys, dalembertsol, dcoeffs, de2diffop, dfieldplot, diffop2de, 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, 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, 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)**, **_mu = int_factor_form**, **try_hard=true**), 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. Cuando la opción try_hard está en true buscará factores integrantes dependientes en n-1 variables donde n es el orden de la EDO. Por último _mu(opcional), fuerza a que busque factores integrantes de una determinada forma. Ej:

```
> with(DEtools):
> EDO := diff(y(x),x) = (y(x)^2-x^2)/(2*x*y(x));
```

$$EDO := y' = \frac{y^2 - x^2}{2\,y\,x}$$

```
> mu := intfactor(EDO);
```

$$\mu := \frac{y}{x}, \frac{y}{y^2 + x^2}$$

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**, **solutionForm**), donde **dvar** es la variable dependiente de la ecuación; **partsol** es una solución parcial de la ecuación (o una lista de soluciones); y **solutionform** sirve para indicar si la solución debe ser resuelta explícitamente, es decir, si ponemos *basis* nos devolverá la solución a la ecuación, si lo dejamos en blanco devolverá una ecuación de orden menor. 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.

$$y'' - 3\,y' + 2\,y$$

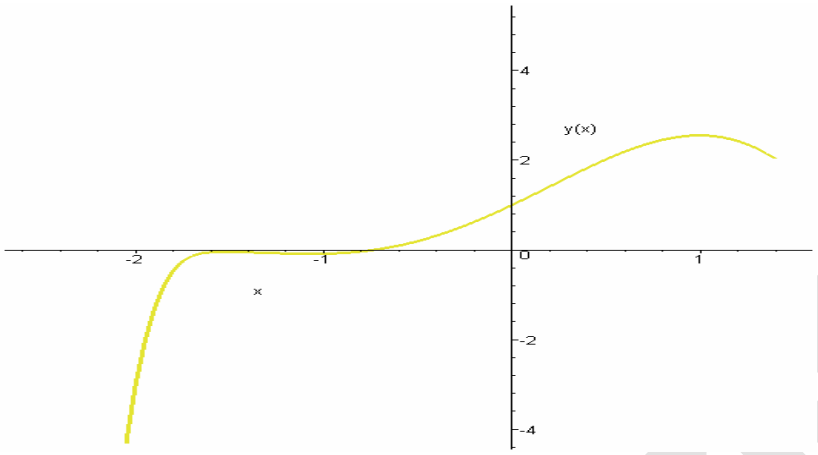
```
> reduceOrder( de, y(x), sol, basis);
```

En este caso nos resuelve la ecuación.

$$\left[\mathbf{e}^x,\ \mathbf{e}^{(2x)},\ \frac{1}{2}\,\mathbf{e}^{(3x)}\right]$$

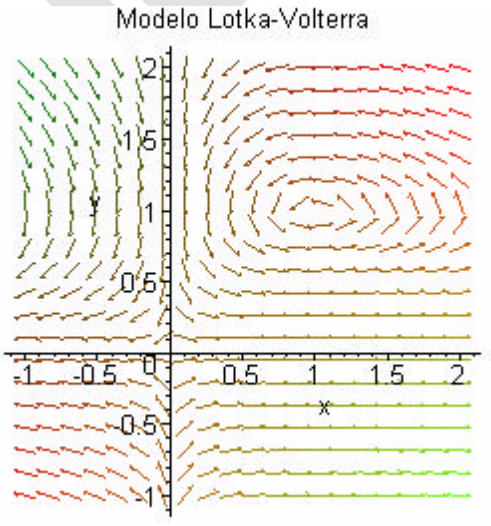
- **DEplot:** Esta función nos permite representar las soluciones de sistemas de ecuaciones diferenciales. Tiene multitud de maneras de ser llamada pasando distintos argumentos, veamos la más sencilla (podemos acceder a las demás haciendo ?DEplot). En este caso la sintaxis es DEplot(**deqns**, **vars**, **trange**, **options**) donde deqns es una lista o conjunto de EDOs de primer orden, o una única ecuación de cualquier orden. El parámetro vars es la variable dependiente, o un conjunto o lista de variables dependientes, trange es el rango de la variable independiente. Ej:

```
> 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);
```



- **dfieldplot:** Nos muestra el campo de direcciones de un sistema de ecuaciones diferenciales. La sintaxis es dfieldplot(**deqns**, **vars**, **trange**, **yrange**, **xrange**, **options**), 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]);
```



Algunos comandos que pueden tener interés se describen brevemente a continuación.

El comando *pdsolve* puede encontrar soluciones a ecuaciones en derivadas parciales. En cada solución aparecerán funciones arbitrarias como _F1, _F2...

```
>  pde := D[1, 1, 2, 2, 2](U)(x, y) = 0;
```

$$pde := D_{1,1,2,2,2}(U)(x,y) = 0$$

```
> pdsolve(pde, U(x, y));
```

$$U(x,y) = \_F5(x) + \_F4(x)\,y + \frac{1}{2}\,\_F3(x)\,y^2 + \_F2(y) + \_F1(y)\,x$$

Maple también puede resolver ecuaciones diferenciales en derivadas parciales no homogéneas:

```
> pde := D[1, 1, 2, 2, 2](U)(x, y) = sin(x*y);
```

$$pde := D_{1,1,2,2,2}(U)(x,y) = \sin(x\,y)$$

```
> pdsolve(pde, U(x, y));
```

$$U(x,y) = \_F5(x) + \_F4(x)\,y + \frac{1}{2}\,\_F3(x)\,y^2 + \_F2(y) + \_F1(y)\,x - x\left(-\frac{x\,y\,\mathrm{Si}(x\,y) + \cos(x\,y)}{x^2} + \frac{\frac{1}{2}\,x^2\,y^2\,\mathrm{Ci}(x\,y) - \frac{1}{2}\cos(x\,y) - \frac{1}{2}\,x\,y\sin(x\,y)}{x} + \frac{\cos(x\,y)}{x}\right)$$

La representación de las soluciones de las ecuaciones diferenciales en derivadas parciales se suele realizar mediante superficies.

Sea la E.D.:

```
> pde := D[1](z)(x, y) + z(x, y)*D[2](z)(x, y) = 0;
```

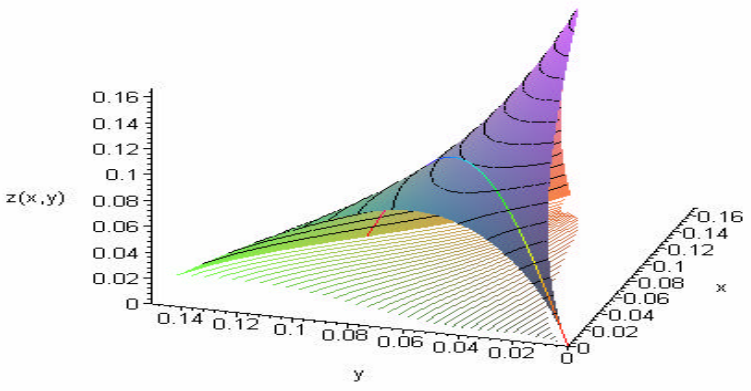
$$pde := D_1(z)(x,y) + z(x,y)\,D_2(z)(x,y) = 0$$

Para representar la superficie se debe introducir el valor inicial, que es una curva parametrizada en el espacio 3D.

```
> 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, z(x,y), [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);
```



5.5. 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.

5.5.1. Transformada de Laplace

5.5.1.1 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
```

$$\text{Laplace}\left(\frac{d^2}{dt^2}\,\text{f}(t),\,t,\,s\right)=s\,(s\,\text{F}(s)-\text{f}(0))-\text{D}(f)(0)$$

```
> 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);
```

$$s\,(s\,\text{F}(s)-\text{f}(0))-\text{D}(f)(0)$$

```
> eval(%);
```

$$s^2\,\text{F}(s)$$

```
> 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);
```

$$\frac{\mathbf{e}^{(-s\,a\sim)}}{s}$$

Como se verá más adelante, los usuarios podrán mediante ***addtable*** añadir sus propias funciones a la tabla interna que dispone el programa para calcularlas.

5.5.1.2 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);
```

$$\text{Heaviside}(t-4)\,(\mathbf{e}^{(-3t+12)}\cos(t-4)-3\,\mathbf{e}^{(-3t+12)}\sin(t-4))$$

```
> factor(%);
```

$$\text{Heaviside}(t-4)\,\mathbf{e}^{(-3t+12)}\,(\cos(t-4)-3\sin(t-4))$$

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);
```

$$F(s)$$

```
> eqn:=diff(f(t),t$2)+f(t)=Heaviside(t-3); #ecuación
```

$$eqn := \left(\frac{d^2}{dt^2}f(t)\right)+f(t)=\text{Heaviside}(t-3)$$

```
> f(0):=0; D(f)(0):=1; #condiciones iniciales
```

$$f(0) := 0$$

$$D(f)(0) := 1$$

```
> laplace(eqn,t,s); #transformamos la ecuación
```

$$s^2F(s)-1+F(s)=\frac{\mathbf{e}^{(-3s)}}{s}$$

```
> sol:=solve(%,F(s)); #resolvemos
```

$$sol := \frac{s+\mathbf{e}^{(-3s)}}{s(s^2+1)}$$

```
> f(t):=invlaplace(sol,s,t); #aplicamos la transformada inversa
```

$$f(t) := \sin(t)+\text{Heaviside}(t-3)\,(1-\cos(t-3))$$

5.5.2. Transformada de Fourier

5.5.2.1 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
```

$$w^4\,\mathrm{fourier}(\mathrm{f}(t),t,w)$$

```
> F:=int(g(tau)*h(t-tau),tau=-infinity..infinity);
```

$$F := \int_{-\infty}^{\infty} \mathrm{g}(\tau)\,\mathrm{h}(t-\tau)\,d\tau$$

```
> fourier(F,t,w); #teorema de la convolución
```

$$\mathrm{fourier}(\mathrm{g}(t),t,w)\,\mathrm{fourier}(\mathrm{h}(t),t,w)$$

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:

$$\mathrm{F}(s) := \frac{\sqrt{2}}{\sqrt{\pi}} \int_0^{\infty} \mathrm{f}(t)\,\sin(s\,t)\,dt$$

$$\mathrm{F}(s) := \frac{\sqrt{2}}{\sqrt{\pi}} \int_0^{\infty} \mathrm{f}(t)\,\cos(s\,t)\,dt$$

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
```

$$\frac{t^3}{t^2+1}$$

5.5.2.2 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:

$$f(t)=\frac{1}{2}\left(\frac{1}{\pi}\int_{-\infty}^{\infty}F(\omega)\,\mathbf{e}^{(\omega\,t\,I)}\,d\omega\right)$$

Su funcionamiento es completamente análogo al de la transformada directa. Veamos algunos ejemplos:

```
> fourier(t/(t^2+1),t,w);
```

$$\pi\,(\mathbf{e}^{w}\,\text{Heaviside}\,(-w)-\mathbf{e}^{(-w)}\,\text{Heaviside}\,(w))\,I$$

```
> invfourier(%,w,t);
```

$$\frac{t}{t^2+1}$$

```
> eqn:=diff(y(t),t$2)-y(t)=cos(t)*sin(t); #ecuación diferencial
```

$$eqn:=\left(\frac{d^2}{dt^2}y(t)\right)-y(t)=\cos(t)\sin(t)$$

```
> fourier(eqn,t,w);
```

$$-\text{fourier}\,(y(t),t,w)\,(1+w^2)=\frac{1}{2}I\,\pi\,(\text{Dirac}\,(w+2)-\text{Dirac}\,(w-2))$$

```
> solve(%,'fourier'(y(t),t,w));
```

$$\frac{\frac{-1}{2}I\,\pi\,(\text{Dirac}\,(w+2)-\text{Dirac}\,(w-2))}{1+w^2}$$

```
> 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);
```

$$\frac{1}{2}\sin(2\,t)=\frac{1}{2}\sin(2\,t)$$

5.5.3. 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, *exp* 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);
```

$$\text{laplace}(g(3p+2), p, x)$$

```
> addtable(laplace,g(a*x+b),G(s+a)/(b-a),x,s,{a,b}); #añadiendo con
parámetros
> 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`);
> restart;
> with(inttrans):
> read(`Mi_tabla.m`);
> laplace(f(t),t,s);
```

$$F(s)$$

5.6. 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 *statplots* nos permite representar los datos estadísticos de diversas maneras.

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](datos
2);
```
$$\frac{\sqrt{6}}{3} = \frac{\sqrt{6}}{3}$$

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)];
```
$$datos1 := [18, 5, \text{Weight}(5, 4)]$$

```
> 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. 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. Por ejemplo:

```
> 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.
```

$$\frac{34}{5}$$

- **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. 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)];
```

$$datos1 := [\mathrm{Weight}(2, 2), \mathrm{Weight}(3, 20), \mathrm{Weight}(4, 2)]$$

```
> describe[kurtosis](datos1);
```

$$6.$$

```
> datos2:=[Weight(2,2),Weight(3,3),Weight(4,2)];
```

$$datos2 := [\mathrm{Weight}(2, 2), \mathrm{Weight}(3, 3), \mathrm{Weight}(4, 2)]$$

```
> 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. 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, gap)*, donde gap (opcional) es el hueco entre las distintas clases. Ej:

Cuando el número de observaciones es impar 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, gap)* donde *which* es el percentil que se quiere calcular y *gap* (opcional) es el hueco entre las distintas clases. Ej:

```
> data:=[seq(i/19+17*i, i=1..20)];describe[percentile[15]](data);
```

$$data := \left[\frac{324}{19}, \frac{648}{19}, \frac{972}{19}, \frac{1296}{19}, \frac{1620}{19}, \frac{1944}{19}, \frac{2268}{19}, \frac{2592}{19}, \frac{2916}{19}, \frac{3240}{19}, \frac{3564}{19}, \frac{3888}{19}, \frac{4212}{19}, \frac{4536}{19}, \frac{4860}{19}, \frac{5184}{19}, \frac{5508}{19}, \frac{5832}{19}, 324, \frac{6480}{19}\right]$$

$$\frac{972}{19}$$

```
> mispercentiles:=[seq(describe[percentile[9*i]],i=1..6)];
```

Los percentiles requeridos son:

$$mispercentiles := [describe_{percentile_9}, describe_{percentile_{18}}, describe_{percentile_{27}}, describe_{percentile_{36}}, describe_{percentile_{45}}, describe_{percentile_{54}}]$$

```
> mispercentiles(data);
```

Los imprimimos en pantalla:

$$\left[\frac{2916}{95}, \frac{5832}{95}, \frac{8748}{95}, \frac{11664}{95}, \frac{2916}{19}, \frac{17496}{95}\right]$$

- **quadraticmean**: Computa la media cuadrática de un conjunto de datos, la sintaxis es *describe[quadraticmean](datos)*, 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, gap)*, donde *which* es el cuantil requerido como una fracción entre 1 y 0, *offset* es una cantidad añadida a la posición calculada y *gap* (opcional) es el hueco entre las distintas clases. 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, gap)*, donde *which* es el cuartil requerido y *gap* (opcional) es el hueco entre las distintas clases. Ej:

```
> datos:=[10,20,30,40,50,60,70,80];
> describe[quartile[1]](datos)=describe[quantile[1/4]](datos);
```

Podemos hacer la misma operación con cuantiles.

$$datos := [10, 20, 30, 40, 50, 60, 70, 80]$$

$$20 = 20$$

- **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);evalf(%);
```

$$\frac{6278}{121}$$

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, 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[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]]);
```

$$z = 1 + x + \frac{y}{2}$$

```
>fit[leastsquare[[x,y,z]]]([[1,2,3,5,5,5],[2,4,6,8,8,8],[3,5,7,10,15,1
5]]);
```

$$z = 1 + \frac{13}{3}x - \frac{7}{6}y$$

```
>fit[leastsquare[[x,y,z]]]([[1,2,3,5,5],[2,4,6,8,8],[3,5,7,10,Weight(1
5,2)]]);
```

$$z = 1 + \frac{13}{3}x - \frac{7}{6}y$$

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]);
```

$$eq\_fit := y = x^2 + 3x - 4$$

- **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), 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, 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* son los argumentos que corresponden a las funciones.

Ejemplos:

```
> statevalf[pdf,normald](3);
```

0.004431848412

```
> statevalf[icdf,normald[7,2]](0.39);
```

6.441361931

```
> statevalf[cdf,normald](2);
```

0.9772498681

5.6.1. Ejercicio global

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);
```

Calculamos ahora las características de tendencia central y de dispersión

```
> range(datos);
```

Valores extremos

$$35 .. 69$$

```
> median(datos);
```

Mediana

$$51$$

```
> mean(datos);evalf(%);
```

Media

$$\frac{1293}{25}$$

$$51.72000000$$

```
> variance(datos):evalf(%);
```

Varianza

$$89.12160000$$

```
> standarddeviation[1](datos):evalf(%);
```

Desviación estándar

$$9.536268042$$

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
> skewness(datos):evalf(%);
                                0.05900558828
```

5.7. **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.

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.

Botones adicionales para opciones gráficas 2-D

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 o si es diferente.

Puesto que tenemos muchos tipos de gráficos distintos y cada uno tiene distintas opciones, lo más comodo 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:

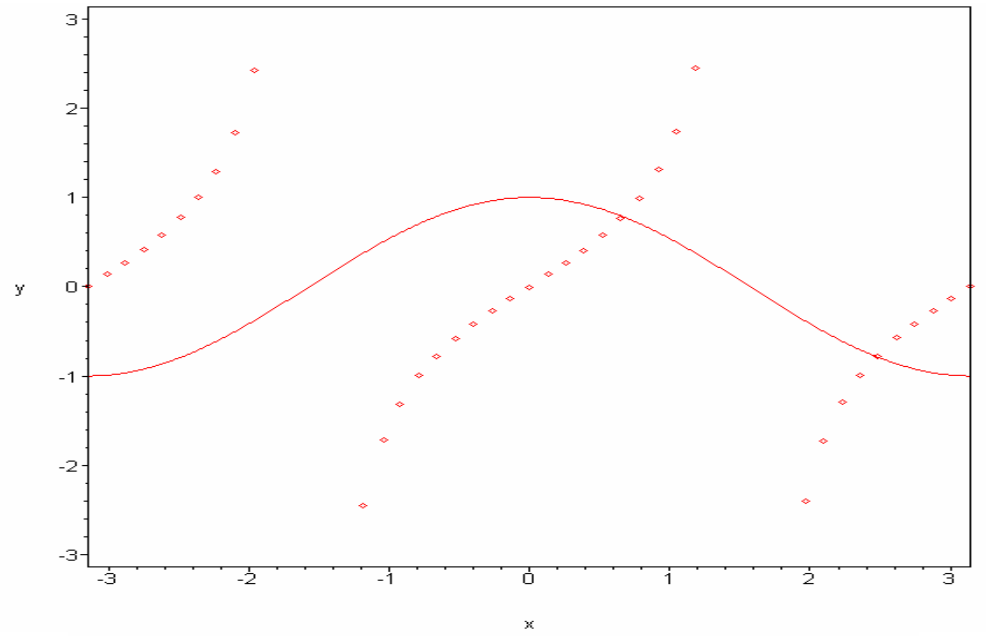
```
>with(plots);
```

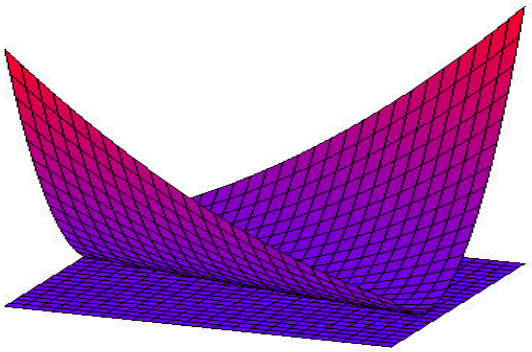
[*animate, animate3d, animatecurve, arrow, changecoords, complexplot, complexplot3d, conformal, conformal3d, contourplot, contourplot3d, coordplot, coordplot3d, cylinderplot, densityplot, display, display3d, fieldplot, fieldplot3d, gradplot, gradplot3d, graphplot3d, implicitplot, implicitplot3d, inequal, interactive, listcontplot, listcontplot3d, listdensityplot, listplot, listplot3d, loglogplot, logplot, matrixplot, 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**, insequence=true, **options**) donde L es la lista, conjunto o array de estructuras plot que se quieren visualizar; la opción insequence(opcional) hace referencia a si se quiere que se muestren los diferentes gráficos en una secuencia, es decir frame a frame (por defecto es false) y options son las opciones del gráfico. 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});
```

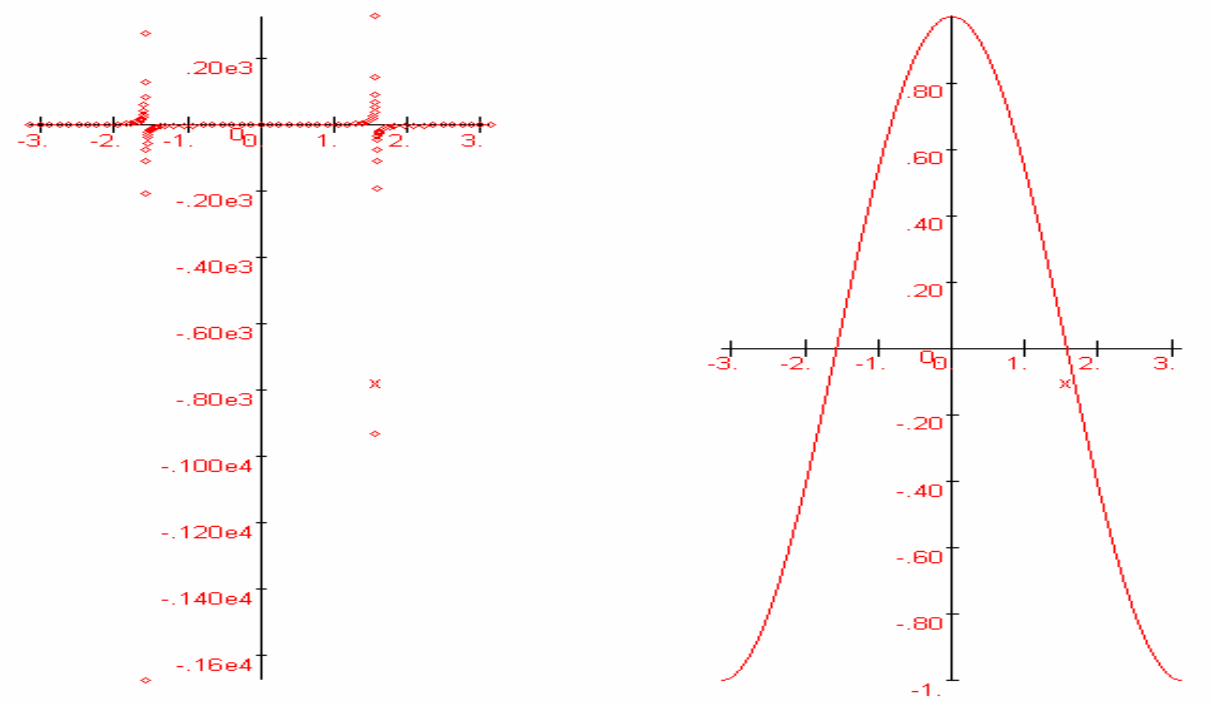
ejemplos con arrays de plots





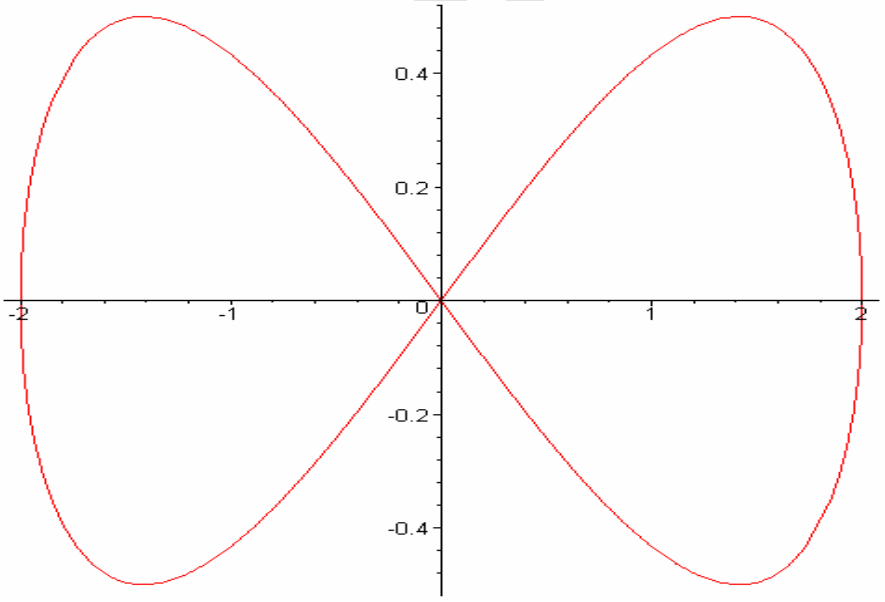
```
> A := array(1..2):
A[1] := plot(tan(x),x=-Pi..Pi,style=point):
A[2] := plot(cos(x),x=-Pi..Pi):
display(A);
```

5.7.1. Ejemplo usando arrays



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):

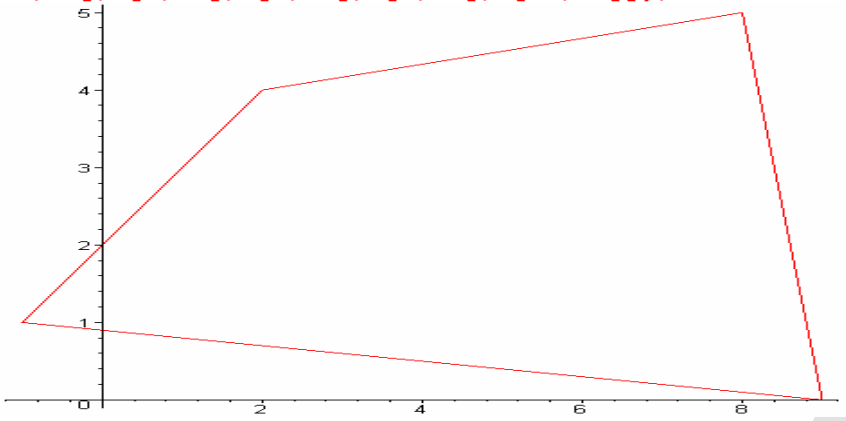
```
> plot([2*sin(t), sin(t)*cos(t), t=0..2*Pi]);
```



Para no ver el dibujo distorsionado tiene que añadir la opción **scaling=constrained** o bien clicar sobre el icono correspondiente.

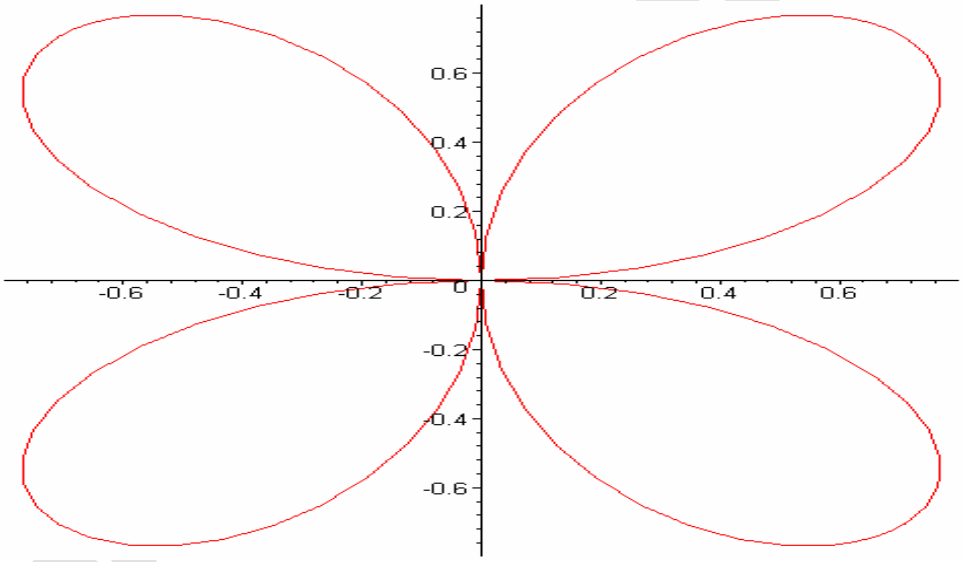
Se pueden dibujar asimismo *líneas poligonales*, es decir, conjuntos de puntos unidos por líneas rectas bien mediante la función ***plot,*** bien mediante ***polygonplot*** de la siguiente manera: las dos coordenadas de cada punto se indican de forma consecutiva entre corchetes [ ], en forma de lista de listas. Obsérvese el siguiente ejemplo, en el que se dibuja un cuadrilátero:

```
> plot([[-1, 1], [2, 4], [8, 5], [9, 0], [-1, 1]]);
```



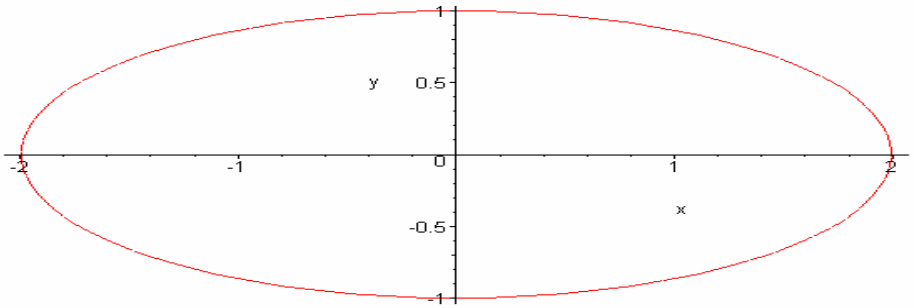
- **polarplot:** Esta función nos permite dibujar una o más curvas en un espacio bidimensional dadas unas coordenadas polares. La sintaxis es polarplot(**L**, **options**), donde L es un conjunto de curvas bidimensionales y options son las opciones del gráfico a las que se puede acceder mediante el botón derecho del ratón. Ej:

```
> with(plots):
polarplot(sin(2*theta));
```



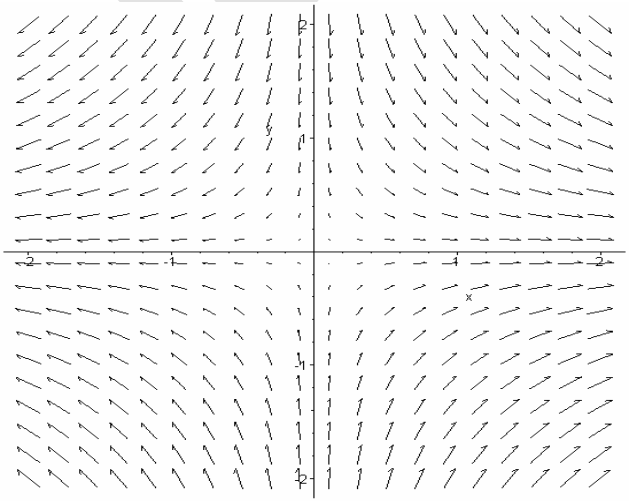
- **implicitplot:** Nos permite dibujar curvas en dos dimensiones de expresiones dadas de manera implícita. La sintaxis es implicitplot(**expr1**, **x**=**a**..**b**, **y**=**c**..**d**, **options**) en el caso de una expresión y implicitplot(**f**, **a**..**b**, **c**..**d**, **options**) en el caso de una función, en ambas definimos los rangos en lo que se quiere trabajar. En el caso tridimensional tenemos una función similar, implicitplot3d, el funcionamiento de esta función es análogo a lo explicado para el caso bidimensional, para más ayuda teclear ?implicitplot3d. Ejemplo del caso bidimensional:

```
> with(plots):
implicitplot((x^2)/4 + y^2 = 1,x=-4..4,y=-1..1);
```



- **fieldplot**: Mediante esta función imprimimos un campo vectorial, 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 caso tridimensional tenemos una función similar, fieldtplot3d, el funcionamiento de esta función es análogo a lo explicado para el caso bidimensional (teniendo en cuenta que los rangos son para un espacio tridimensional en este caso), para más ayuda teclear ?fieldplot3d. Ejemplo del caso bidimensional:

```
> 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);
```



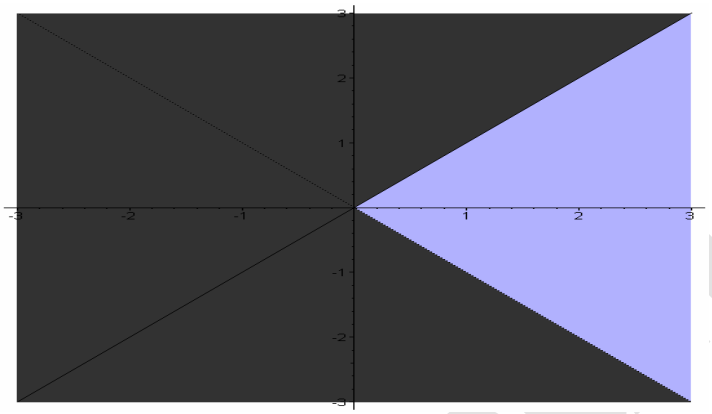
- **inequal:** Ofrece la posibilidad de representar sistemas de inecuaciones en 2 variables, la sintaxis es inequal(**ineqs**, **xspec**, **yspec**, **options**), xspec e yspec son los rangos en los que se representa y options son las siguientes 4 opciones que tenemos en el gráfico:

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:

```
> with(plots):
inequal({x+y>0,x-y>=0}, x=-3..3, y=-3..3 );
```

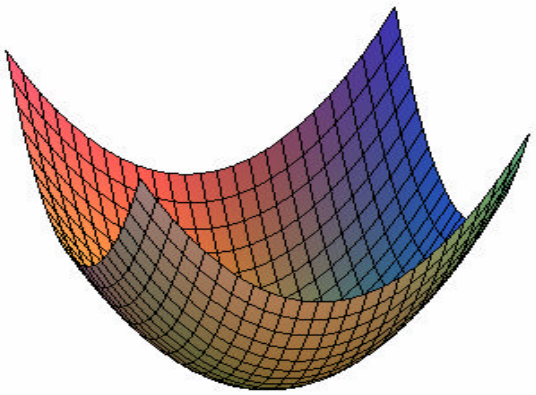


- **plot3d:** Podemos representar gráficos tridimensionales definiendo una expresión de 2 variables en cuyo caso hay que definir los rangos en los que queremos representarla, o bien definiéndola como función de 2 variables, en este caso pondremos los rangos sin indicar la variable, se entiende que van 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.

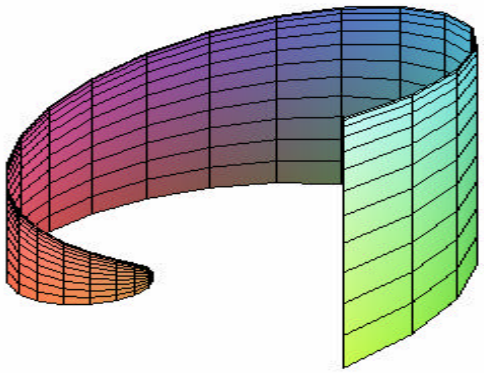
Finalmente hay un botón para controlar que se dibuje con la misma escala según los tres ejes.

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.

Otra función para representar gráficos tridimensionales es ***smartplot3D*** (sólo disponible en Windows 95, Windows NT o superior). De hecho, cuando clicamos sobre ***Plots*** en el menú contextual Maple escribe y ejecuta automáticamente ***smartplot3D*** para expresiones de 2 variables independientes.

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);
```

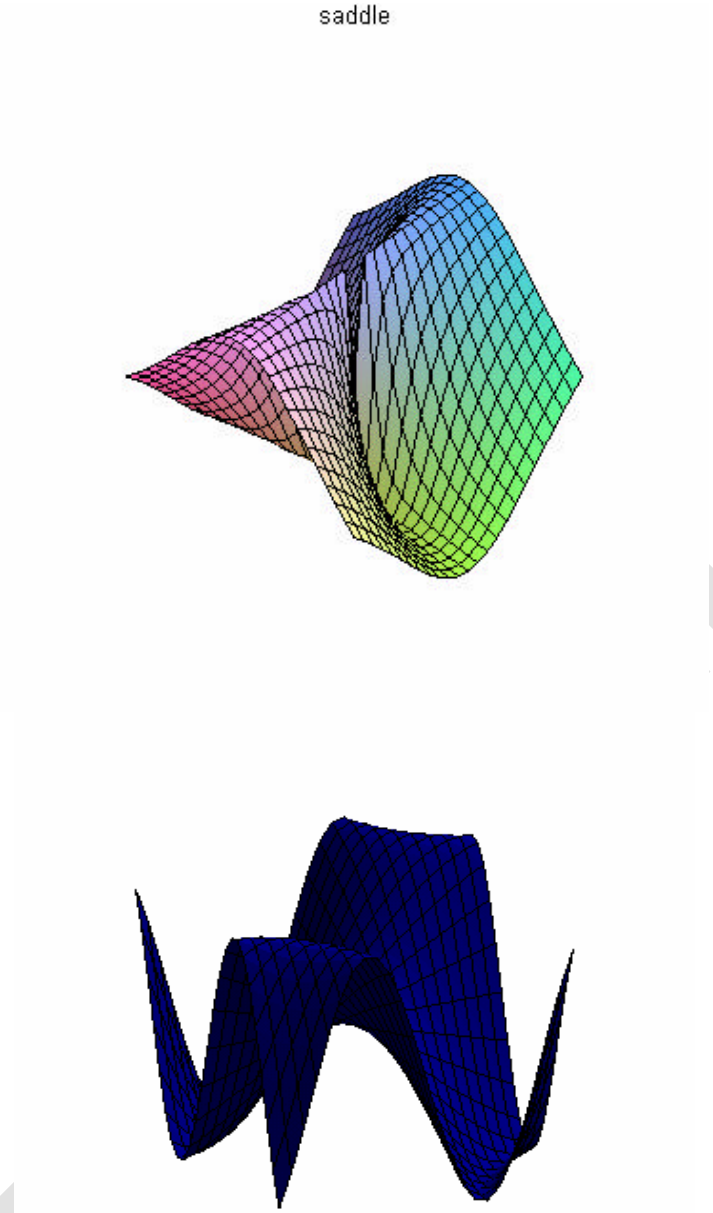


En el caso de las gráficas tridimensionales aumenta notablemente el número de opciones o parámetros de Maple que puede controlar el usuario. Aquí sólo se van a citar dos, pero para más información se puede teclear ?plot3d[options].

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]);
```

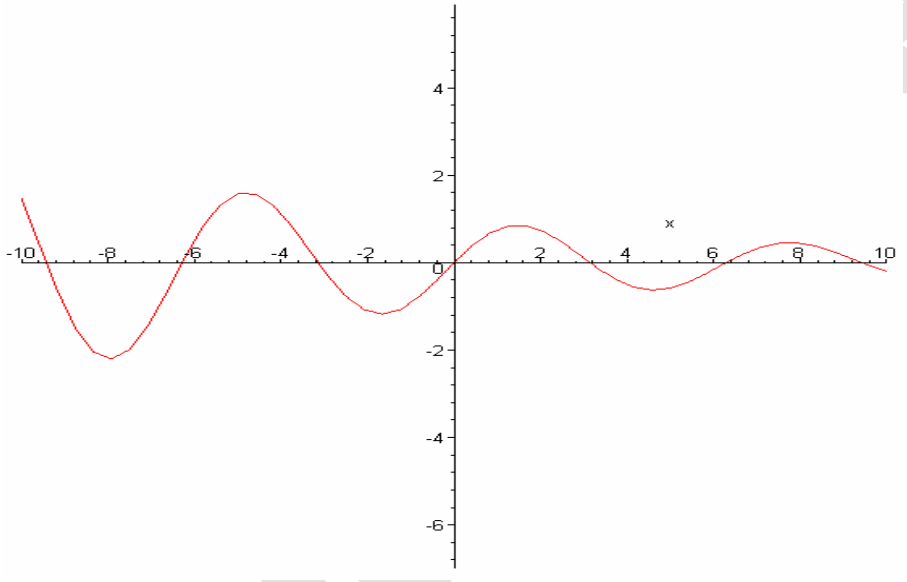


- **animate:** Maple realiza *animaciones* con gran facilidad. En las animaciones se representa una función que varía en el tiempo o con algún parámetro. Este

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:

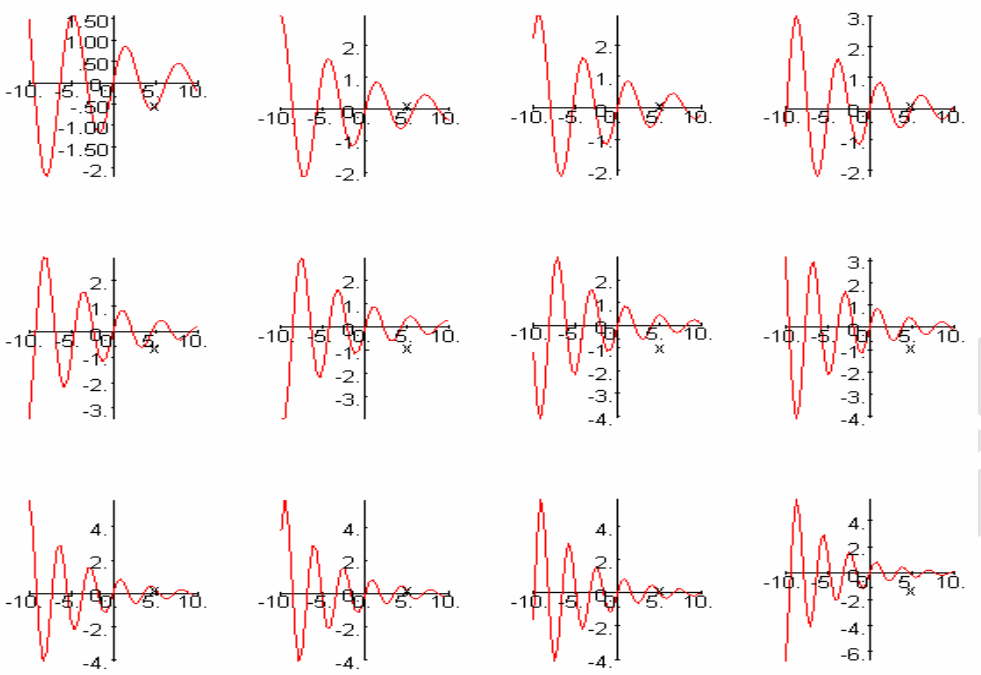
```
> with(plots):
animate( exp(-0.1*x*t)*sin(x*t),x=-10..10,t=1..2,frames=12);
```



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:

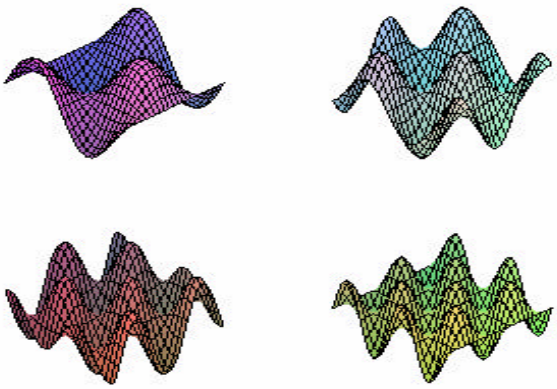
```
> display(%);
```

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(%);
```



6- INTRODUCCIÓN A LA PROGRAMACIÓN CON MAPLE 8

Maple 8, así como todas las versiones anteriores de Waterloo Maple, dispone de un lenguaje de programación propio, con sentencias similares a los demás lenguajes de programación no dirigidos específicamente a la matemática. Por otro lado tenemos la opción de crear interfaces mediante unas nuevas estructuras llamadas Maplets, que se mostrarán más adelante. También permite el desarrollo de procedimientos, módulos, funciones y expresiones. Una de las grandes ventajas representadas por este lenguaje de programación es la posibilidad de traducirlo a otros diferentes como pueden ser ANSI C, Fortran o JAVA, de gran utilidad y que se mostrará con detalle en este apartado.

6.1. SENTENCIAS BÁSICAS DE PROGRAMACIÓN

6.1.1. La sentencia u operador *if*

Se trata de la sentencia de condición más utilizada, que en el caso del Maple 8 se puede escribir tanto como sentencia como operador. La forma más sencilla de utilizarla es la siguiente (como sentencia):

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*** o ***fi***. Si no se desean utilizar ***elif*** ni ***else***, basta con no incluirlos en la sentencia ***if***, su uso no es necesario, si bien son un complemento imprescindible para realizar algoritmos con bifurcaciones.

Si la utilizamos como operador, tendrá el siguiente aspecto:

`if`(**expresión de condición**, **expresión si se cumple**, **expresión si no se cumple**)

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.

6.1.2. El bucle *for*

Esta sentencia nos servirá para realizar iteraciones o repeticiones, en su estructura, que se muestra a continuación, se puede observar que no sólo es capaz de repetir la misma operación en cada iteración sino que además incluye un while, que sirve para hacer condiciones durante la ejecución del bucle. El bucle termina con **end do** u **od**:

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

Hay una segunda manera de hacer bucles ***for***, su sintaxis se muestra a continuación:

for **<var>** in **<expresión>** while **<expresión de condición>**
do **<sentencias>** end do;

En este caso, la expresión puede ser un vector, y *var* puede ser cada elemento del mismo, no es necesaria la utilización del ***while***. Por ejemplo:

```
> L:=[7,2,5,8,7,9];
```

$$L := [7, 2, 5, 8, 7, 9]$$

```
> for i in L do
> if i <= 7 then
> print(i);
> end if;
> end do;
```

7

2

5

7

Como se puede observar, en este caso, la variable *i* representa cada elemento del vector *L*, y en caso de que se cumpla la condición impuesta, se imprimirán en pantalla.

6.1.3. El bucle *while*

Es utilizado para ejecutar repetidamente una secuencia de sentencias a menos que una condición deje de satisfacerse, en cuyo caso deja de repetirse. 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**.

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.***

Maple 8 no reconocerá esta sentencia a menos que esté introducida en un bucle o repetición, y devolverá un mensaje de error.

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***.

Maple 8 no reconocerá esta sentencia a menos que esté introducida en un bucle o repetición, y devolverá un mensaje de error.

6.1.6. El comando *map*

El comando ***map*** aplica una función a todos los elementos agregados a un objeto. La forma más simple es:

map(f, objeto)

Un objeto puede ser tanto una variable como una expresión o un procedimiento, es indiferente. Un ejemplo sencillo de aplicación sería el siguiente:

```
> L:=[-1,2,-3,-4,5];
```

$$L := [-1, 2, -3, -4, 5]$$

```
> q:=map(abs, L);
```

$$q := [1, 2, 3, 4, 5]$$

```
> map(x->x^2,L);
```

$$[1, 4, 9, 16, 25]$$

Este comando se considera como iterativo o perteneciente a los comandos de control de flujo, debido a que realiza un bucle recorriendo cada elemento agregado al objeto, por supuesto, este comando tiene muchas más utilidades que aquí no se mostrarán.

6.1.7. Otros comandos sobre iteraciones

- ***Select, remove y selectremove:***

El comando ***select*** devuelve los operandos que evaluados de forma booleana devuelven ***true***. El comando ***remove*** devuelve los operandos que evaluados de forma booleana devuelven ***false***. 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***.

- ***Zip:***

El comando ***zip*** necesita dos vectores a los que aplica una función binaria (una función con dos operandos), devuelve un vector cuyos elementos son el resultado de haber aplicado una función a los dos elementos extraídos de cada vector. Por ejemplo:

```
> zip((x,y)->x+y, [a,b,c,d,e],[1,2,3]);
```

$$[a+1, b+2, c+3]$$

- ***Los comandos*** seq**,** add ***y*** mul***:***

Estos comandos forman secuencias, sumas y multiplicaciones respectivamente.

6.2. PROCEDIMIENTOS CON MAPLE

La definición de un procedimiento de Maple es un grupo de sentencias introducidas en la estructura ***proc()...end proc***. En esta construcción se deben declarar los parámetros y variables que el procedimiento va a utilizar, y especificar las sentencias que forman el cuerpo del procedimiento. También se pueden definir procedimientos simples de una línea usando operadores funcionales.

6.2.1. Definición de procedimiento

La sintaxis general de un procedimiento de Maple es la siguiente:

proc(*P*)
local *L*;
global *G*;
options *O*;
description *D*;
cuerpo del procedimiento
end proc

La letra P representa a los parámetros.

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:

procedureName(A);

El nombre del procedimiento que vamos a invocar es *procedureName* y la letra A representa los parámetros que se van a utilizar en dicho procedimiento.

6.2.2. Componentes de un procedimiento

6.2.2.1 Parámetros

Se puede escribir un procedimiento que sólo funcione con un tipo determinado de entradas. En este caso sería 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. 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);
```

Si se le pasan más parámetros A que los que se necesitan, ignora los que sobran. Si se le pasan menos y los necesita, el programa mandará un mensaje de error; pero si no los necesita, tampoco pasa nada.

6.2.2.2 Variables locales y variables globales

En un proceso pueden existir tanto variables locales como globales. Fuera de un proceso las variables serán globales. Existen dos diferencias principales entre variables locales y variables globales:

Maple considera que las variables locales en diferentes llamadas a un procedimiento son variables distintas, aunque tengan el mismo nombre. De esta forma, un procedimiento puede cambiar el valor de una variable local sin que afecte a variables locales o globales con el mismo nombre pero de otros procedimientos.

Se recomienda declarar el carácter de las variables explícitamente. Si no se hace así, Maple lo asigna. Convierte una variable en local:

- 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` o `mull`.

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 suele ser de interés.

```
> F:=proc()
>     local x, y, z;
>     x:= y^2;  y:= z^2;  z:=3;
>     eval(x);
> end:
> F();
```

$$81$$

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 diferenciar cuándo se está trabajando con números reales y cuándo con enteros. Cuando se trabaja con números reales, Maple opera de manera eficiente, es decir, 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);
```

$$\sin\left(\frac{3}{4}\right)$$

```
> sin(3./4.);
```

$$.6816387600$$

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.

6.2.2.3 Options

Un procedimiento puede tener una o varias opciones que ofrece Maple:

`Options` *`O1, O2, ..., On`*

- **Opciones remember y system**

Cuando se llama a un procedimiento con la opción remember, Maple guarda el resultado en una remember table. Así otra vez que se invoque al procedimiento, Maple chequeará si ha sido llamado anteriormente con los mismos parámetros. Si es así, tomará los resultados directamente en vez de recalcularlos.

La opción `system` permite a Maple borrar resultados anteriores de una remember table.

- **Opciones operator y arrow**

La opción operator permite a Maple hacer simplificaciones extra al procedimiento y la opción arrow indica que se debe mostrar el procedimiento por pantalla utilizando la notación de flechas.

```
> f:=proc()
> option operator, arrow;
> x^2;
> end;
```

$$f := (\,) \rightarrow x^2$$

- **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 no ser que se especifique lo contrario.

```
> f:=proc(expr::anything, x::name)
> option `Copyright 1684 by G.W. Leibniz`;
> Diff(expr,x);
> end;
```

$$f := \mathbf{proc}(expr::anything, x::name)\ \ldots\ \mathbf{end}$$

6.2.2.4 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 el procedimiento tenga la opción de `copyright`.

6.2.3. Description *string*; Procedimientos: valor de retorno

Cuando se ejecuta 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.
2. A través de un return explícito.
3. Mediante un return de error.

6.2.3.1 Asignación de valores a parámetros

Puede ser que se desee escribir un procedimiento que devuelva un valor a través de un parámetro. Es importante saber que Maple evalúa los parámetros sólo una vez, así que una vez que se ha realizado una asignación al parámetro, no se debe hacer referencia a ese parámetro otra vez, ya que no cambiará su valor.

```
> f:=proc(x::evaln)
> x:=-13;
> x;
> end;
> f(q);
```

$$q$$

```
> q;
```

$$-13$$

6.2.3.2 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*.

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:
```

La función `nops` calcula el número de operandos de una expresión.

6.2.3.3 Return de error

Un *return de error* ocurre cuando se llama al comando `ERROR` , que tiene la siguiente sintaxis:

`ERROR (`*`secuencia`*`)`

Normalmente causa la salida del procedimiento a la sesión de Maple, donde se imprime un mensaje de error.

```
Error, (in nombreProc), secuencia
```

`Secuencia` es el argumento del comando `ERROR` y *`nombreProc`* el nombre del procedimiento donde se ha producido el error. Si el procedimiento no tiene nombre el mensaje será:

```
Error, (in unknown), secuencia
```

La variable global *lasterror* almacena el valor del último error. Se puede utilizar junto con el comando *traperror* para buscar errores.

Traperror evalúa sus argumentos: si no hay error, los devuelve evaluados, pero si ocurre algún error cuando Maple está evaluando los argumentos, devuelve el correspondiente mensaje de error. Además, al llamar a *traperror*, se borra lo almacenado en *lasterror*. Así, si el resultado de *traperror* y *lasterror* coinciden se sabe que en esa expresión se ha producido un error.

```
> f := u -> (u^2-1)/(u-1);
```

$$f := u \to \frac{u^2 - 1}{u - 1}$$

```
> printlevel:=3:
> for x in [0, 1, 2] do
> r:=traperror( f(x) );
```

```
> if r=lasterror then
> if r=`division by zero` then
> r:=limit(f(u), u=x)
> else
> ERROR(lasterror)
> fi
> fi;
> lprint(`Result:  x =`, x, `f(x) =`, r)
> od;
```

$$x := 0$$

$$r := 1$$

```
> Result:  x=0   f(x)=1
```

$$x := 1$$

$$r := division\ by\ zero$$

$$r := 2$$

```
> Result:  x=1   f(x)=2
```

$$x := 2$$

$$r := 3$$

```
> Result:  x=2   f(x)=3
```

En el ejemplo se evalúa la función $f(u) = \frac{u^2 - 1}{u - 1}$ para los valores de u 0, 1 y 2 utilizando el comando traperror. Si r, al que se le ha asignado el valor de retorno de traperror, coincide con la respuesta de lasterror quiere decir que se ha producido un error. En ese caso se chequea el tipo de error. Si se trata de una indeterminación por ser el divisor de la expresión igual a 0, se calcula el límite de la función para ese valor de la variable. Si es otro tipo de error, se manda su correspondiente mensaje.

6.2.4. 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**. 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`:

```
> save nombreProc, "nombreProc.m";
> read "nombreProc.m";
```

6.2.5. 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. 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:

Considerando que se desee programar un procedimiento que utiliza una función, 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);
```

$$x \rightarrow \sin(x + 1)$$

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;
```

$$h := (x, y) \rightarrow x\,y$$

```
> shift := (f::procedure) -> (x->f(x+1, args [2..-1])):
> hh := shift(h);
```

$$hh := x \rightarrow h(x+1, \mathrm{args}_{2\,..\,-1})$$

```
> 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);
```

$$y := a\,x^2 + 1$$

6.2.6. Ejemplo de programación con procedimientos

Como ejemplo de procedimiento se realizará la interpolación polinómica mediante los polinomios de Lagrange. Lo primero que debemos hacer (a parte de reiniciar), es decir, cuantos datos vamos a tener en cuenta al aproximar, en nuestro caso serán siete:

```
> restart:
> L:=7;
```

$$L := 7$$

```
> with(linalg):
```

Hemos invocado las funciones del paquete linalg porque nos serán útiles a la hora de hacer el procedimiento. A continuación creamos un vector con las siete abscisas que vamos a utilizar:

```
>  tt:=vector(L,[1,2,3,4,5,6,7]);
```

$$tt := [1, 2, 3, 4, 5, 6, 7]$$

El procedimiento dependerá de dos variables t y n, donde t será nuestra variable independiente y n el grado del polinomio. Su aspecto es el siguiente:

```
> Pol_Lagran:=proc(t,n)
 if n = 1 then product((t-tt[j]),j=2..L)/product((tt[n]-tt[j]),j=2..L)
else product((t-tt[j]),j=1..n-1)*product((t-
tt[j]),j=n+1..L)/(product((tt[n]-tt[j]),j=1..n-1)*product((tt[n]-
tt[j]),j=n+1..L)) fi ;
end proc;
```

Este procedimiento nos devuelve un polinomio cada vez. Para realizar la suma total y realizar la aproximación hacemos un sumatorio de la siguiente manera (se incluye antes el vector con las ordenadas por las que ha de pasar la aproximación):

```
> x:=vector(L,[27,27,43,40,36,34,34]);
```

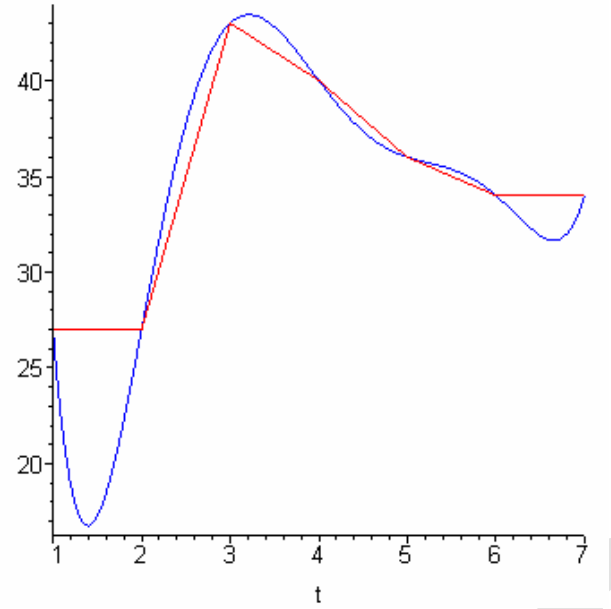
$$x := [27, 27, 43, 40, 36, 34, 34]$$

```
>  x_aproximante:=t->sum(x[k]*Pol_Lagran(t,k),k=1..L);
```

$$x\_aproximante := t \rightarrow \sum_{k=1}^{L} x_k \, \mathrm{Pol\_Lagran}(t, k)$$

Y por último representamos la poligonal que une los puntos dato y la aproximación polinómica para comparar los resultados:

```
> plot1:=plot(x_aproximante(t),t=1..7,color=blue):
> plot2:=plot([seq([tt[i],x[i]],i=1..L)],t=1..7,color=red):
> with(plots):
> display({plot1,plot2});
```



6.3. PROGRAMACIÓN CON MÓDULOS

Los procedimientos permiten asociar una secuencia de comandos con un simple comando. Igualmente, los módulos nos permiten asociar procedimientos y datos externos al Maple. 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 programación en ingeniería:

- **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:

```
> TempGenerator := module()
>    description "Generador de símbolos temporales";
>    export gentemp;
>    local count;
>    count:=0;
>    gentemp := proc()
>         count := 1+count;
>         (T||count)  #Concatena caracteres
>         end proc;
```

```
> end module;
```

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. 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;
```

T2

T3

T4

6.3.1. Sintaxis y semántica de los módulos

La sintaxis de un módulo es muy similar a la de un procedimiento. Una plantilla de un módulo tendría el siguiente aspecto:

```
module()
    local L;
    export E;
    global G;
    options O;
    description D;
    B;
end module
```

6.3.1.1 Parámetros de los módulos

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 B, 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. Muchas declaraciones de los módulos son similares a las de los procedimientos.

Un ejemplo de declaración podría ser la sentencia description, que da información sobre un 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.

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. El resultado de evaluar la definición de un módulo es otro módulo, que se puede visualizar como una colección de exportaciones que son miembros de dicho módulo. La principal diferencia entre estas variables y las locales, es que a éstas podemos acceder antes 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();
```

```
"HELLO WORLD"
```

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 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=nombrep, donde nombrep es el nombre de un procedimiento en la declaración local o exportada de un 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.

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("Datos", "%d %e\n", xy[1], xy[2]) end do:
> fclose("Datos");
```

El archivo Datos a sido guardado en el directorio actual de trabajo. Para determinar dicho 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 Datos. 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("Datos","%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("Datos", [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);
```

96

- ***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.
Esto es una prueba
```

♦ ***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- EL DEBUGGER

Al programar se suelen cometer errores difíciles de localizar mediante una inspección visual. Maple proporciona un *debugger* 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.

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*.

8- MAPLETS

Un maplet es un interface gráfico para el usuario de Maple, que es ejecutado desde una sesión de Maple. Permite al usuario combinar los paquetes y procedimientos de manera interactiva mediante ventanas y diálogos.

Los maplets tienen muchas aplicaciones, pueden ser utilizados para crear calculadoras personalizadas, interfaces, rutinas, preguntas y mensajes entre otras.

8.1. ELEMENTOS DE LOS MAPLETS

Un maplet puede responder a eventos o acciones que hace el usuario, como puede ser hacer clic o cambiar el valor de una caja de texto. Aparte de los elementos puramente visuales como son los botones y menús, los maplets contienen elementos no visuales, generalmente operaciones matemáticas o procedimientos que van detrás del interface y nos devuelven un resultado por pantalla. En lo que sigue se irán describiendo cada uno de estos elementos.

8.1.1. Elementos del cuerpo de la ventana

- **Button:**

Definen un botón que puede aparecer en una ventana tipo maplet. Cada botón está asociado con un elemento tipo Action, que es ejecutado cuando se presiona el botón. Se pueden modificar su fuente, color, texto y otras propiedades. Su sintaxis es:

Button(Opciones)

Donde Opciones puede ser: Caption (Texto en el botón), background (fondo), enabled (activar), font (fuente), onclick (acción que se activa), y otras más que se pueden consultar en la ayuda.

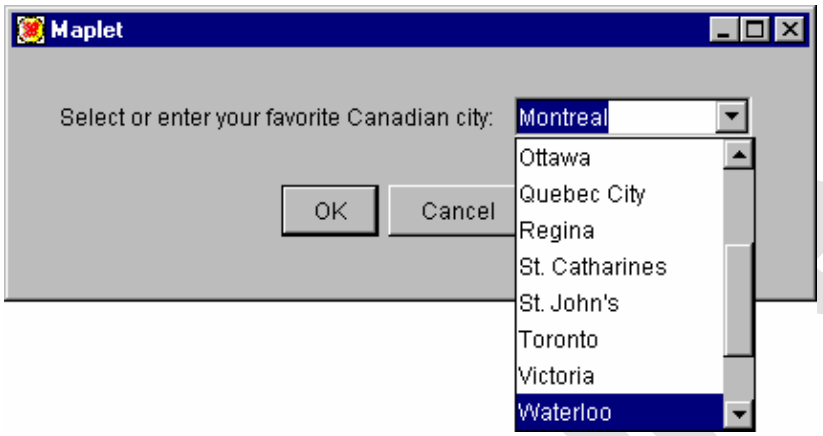
- **Checkbox:**

Son elementos de selección, tienen varias opciones, y se ejecutará la que el usuario haya seleccionado. Tiene el siguiente aspecto:

Su sintaxis es similar a la de Button, es decir: ChechBox(Opciones), donde Opciones pueden ser multitud de características. A veces se suele poner entre corchetes el nombre o descripción del elemento, por ejemplo, podríamos poner: CheckBox[‘opcion1’](caption=red).

♦ **ComboBox:**

Representa una lista de datos predefinidos donde el usuario debe elegir uno de ellos. Su aspecto es el siguiente:



Su sintaxis es la siguiente:

ComboBox(**Opciones, contenido**)

Donde Opciones pueden ser el caption, background y otras características. Contenido representa las opciones posibles de la ComboBox, y se introducen como lista. Un pequeño ejemplo podría ser el siguiente:

```
> with(Maplets[Elements]):
maplet := Maplet([
   [
      "Seleccione una función de las siguientes:  ",
      ComboBox['CB1'](sort(["Seno", "Coseno", "Tangente", "Arcoseno",
"Arcococoseno", "Arcotangente", "exponencial", "logaritmo", "Logaritmo
neperiano", "Suma", "Resta", "Multiplicación", "División"], lexorder),
"Otras")
   ],
   [Button("OK", Shutdown(['CB1'])), Button("Cancel", Shutdown())]
]):
Maplets[Display](maplet);
```

♦ **DropDownBox:**

Una DropDownBox es similar a una ComboBox con una diferencia, en la segunda se pueden introducir datos aparte de los existentes en la lista, mientras que en la DropDownBox no se puede introducir ningún elemento más aparte de los de la lista.

♦ **Label:**

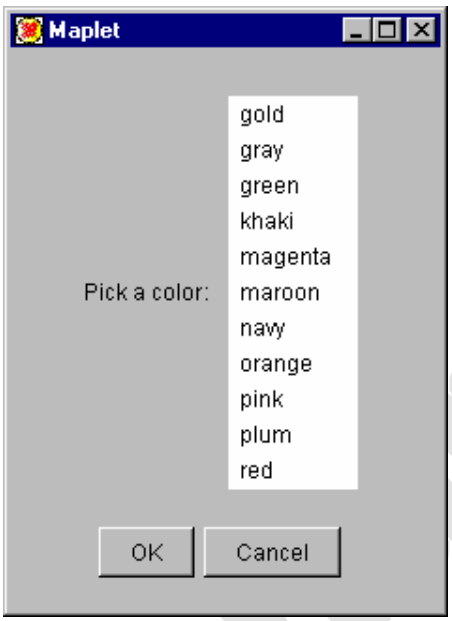
Puede contener tanto texto como imágenes, aunque si se va a utilizar como texto es preferible usar una textbox. Los labels son etiquetas que no sirven para interactuar con el usuario.

♦ **ListBox:**

Una ListBox es similar a una DropDownBox, es una lista en la que el usuario debe elegir uno o más elementos. Para hacer más de una selección se pueden utilizar las

teclas SHIFT y CONTROL. Lo que devuelve este elemento es un string que contiene las selecciones entre comas, que puede ser convertida en lista de strings mediante la función ListBoxSplit del subpaquete Maplets[Tools]. Su sintaxis y apariencia son así:

ListBox(**opts**, **list_box_content**)



- **MathMLEditor:**

Muestra y permite manipular expresiones matemáticas. Para introducir los datos se pueden utilizar las plantillas programadas o directamente mediante el teclado. Las expresiones introducidas pueden ser copiadas y pegadas en la sesión de Maple. Tiene diversas paletas que hacen más fácil la representación de símbolos matemáticos. Un ejemplo sencillo de aplicación sería el siguiente, en el que se permite introducir una expresión matemática y cuando se presiona "OK" se pasa dicha expresión a Maple:

```
> with(Maplets[Elements]):
maplet := Maplet([
   [BoxCell("Introduzca una expresión:")],
   [MathMLEditor('reference'='ME1')],
   [Button("Done", Shutdown([ME1]))]
]):
result:=Maplets[Display](maplet);
MathML[Import](result[1]);
```

- **MathMLViewer:**

Muestra expresiones del tipo MathML. Dichas expresiones no se pueden copiar y pegar en una sesión de Maple.

- **Plotter:**

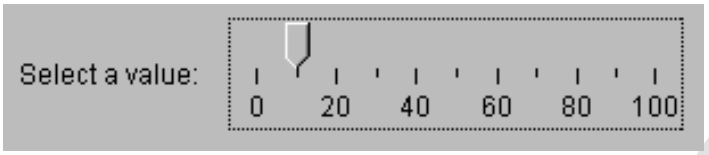
Permite representar funciones en 2-D y 3-D, simplemente muestra de manera estática un plot realizado con Maple en un maplet.

- **RadioButton:**

Es similar a una CheckBox, con la diferencia de que en este caso tan sólo se puede elegir una opción de las mostradas.

- **Slider:**

Permite al usuario especificar un valor entero de un rango de la siguiente manera:



- **Table:**

Los datos son organizados en columnas, sirve pues, como hoja de cálculo o para introducir manualmente multitud de datos.

- **Cajas de texto (TextBox):**

Son espacios donde se permite escribir una o varias líneas para introducir datos u obtenerlos, incluso sirven como simples etiquetas o labels. La opción 'editable' es puesta en su modo true cuando se desea introducir datos, o false cuando se impide al usuario cambiar su contenido. Las cajas de texto tienen menús pop-up que se activan haciendo clic con el botón derecho. Por defecto, una textbox con editable=true, tiene las opciones Cut, Copy, Paste, Delete y Select All activadas en el menú pop-up. Si está en modo editable=false, entonces sólo nos permitirá Copy y Select All en el menú del botón derecho. Además de esto, se pueden incluir más opciones en el menú pop-up mediante la opción 'popupmenu'.

- **TextField:**

Es un campo de entrada o salida de datos de una sola línea, dependiendo si su opción 'editable' está en true o false. Éstas, como las TextBox, tienen menús pop-up al clicar sobre el campo con el botón derecho.

- **ToggleButton:**

Es similar a una CheckBox con la diferencia de su apariencia, en este tipo de elementos se puede poner en los botones tanto texto como imágenes.

8.1.2. Elementos de diseño

El diseño de una ventana define cómo dar posición y tamaño al resto de los elementos de dicha ventana. Hay dos maneras principales de diseñar, con los elementos BoxLayout y GridLayout. Además, permite definir el diseño de la ventana mediante listas anidadas. Cuando se escribe un maplet, los caracteres de texto, los campos de texto, etc son definidos como expresiones en listas. Una ventana de un maplet incluye una lista principal, que contiene listas de strings de textos, y otros elementos. Los principales elementos son los siguientes:

- **BoxLayout:**

Es un esquema de diseño en el que se puede controlar donde aparecen elementos respecto a la horizontal o vertical de otros elementos. Para controlar la posición horizontal se utiliza en elemento BoxRow. El elemento BoxColumn realiza la misma operación pero en la vertical.

- **GridLayout:**

Es un diseño cuadriculado donde todos los elementos deben aparecer en el interior de cada celda. Este tipo de diseño es el más utilizado cuando son simples. Para diseños complejos es mejor utilizar el BoxLayout.

- **Nested lists (listas anidadas):**

Las listas anidadas o listas de listas, pueden ser utilizadas para definir BoxLayouts. En Maple, una lista es una secuencia ordenada de expresiones separadas entre comas y encerradas entre corchetes ([]). Una lista anidada es una lista en la que los elementos pueden ser a su vez listas. Por ejemplo:

```
> NestedList := [1,[2,3],[4,[5,6]],7,8,[9,10]];
```

$$NestedList := [1, [2, 3], [4, [5, 6]], 7, 8, [9, 10]]$$

8.1.3. Elementos de la barra de menú

Los siguientes elementos son clasificados como elementos de una barra de menús: MenuBar (que se debe definir dentro del elemento Window), Menu (definido en un Menu, MenuBar o PopupMenu), MenuItem (definido en MenuBar o PopupMenu), CheckBoxMenuItem (que debe ser definido en un MenuBar o PopupMenu), RadioButtonMenuItem (definido en un MenuBar o PopupMenu), MenuSeparator (definido en MenuBar o PopupMenu) y PopupMenu (que debe ser definido en el elemento de tipo TextField o TextBox).

Una barra de menús puede contener cantidad de menús, cada uno de ellos pueden contener varios miembros, definidos utilizando los elementos MenuItem, CheckBoxMenuItem y RadioButtonMenuItem, y los submenús utilizando elementos Menu anidados. Los separadores se utilizan para distinguir grupos de funciones diferentes (como en cualquier ventana del sistema operativo).

El menú Popup por defecto contiene las opciones Copy y Select All. Si el campo o caja de texto es editable, entonces tiene además las opciones Paste, Delete y Clear. Otras entradas se pueden añadir al menú Popup.

8.1.4. Elementos de una barra de herramientas

Una barra de herramientas puede contener un gran número de botones. Dichos botones se pueden agrupar en diferentes grupos según su utilización utilizando separadores, que producen espacios largos entre botones adyacentes. Los siguientes elementos son clasificados como elementos de una barra de herramientas: ToolBar (debe ser definido en el elemento Window), ToolBarButton (que debe estar definido en el elemento ToolBar) y ToolBarSeparator (definido en el elemento ToolBar)

8.1.5. Elementos de comandos

Un Maplet puede ejecutar programas en función de las acciones que realiza el usuario, como puede ser clicar un botón o cambiar el valor de una caja de texto. Cada elemento de comando se acciona a partir de una acción. Hay elementos no visuales. Los elementos de comando más importantes son:

- **CloseWindow:**

Cierra la ventana en ejecución mediante una referencia a la misma.

- **Evaluate:**

El comando Evaluate ejecuta un procedimiento de Maple con los argumentos marcados en args de la sesión actual de Maple. Un elemento Evaluate puede contener elementos Argument.

- **RunDialog:**

Representa un elemento de diálogo. El elemento RunDialog tan sólo tiene la opción 'dialog', que debe contener la referencia al diálogo que se va a mostrar en pantalla. Si el diálogo ya estaba activado, entonces no ocurrirá nada.

- **RunWindow:**

Representa un elemento Window. Este elemento tiene la opción 'window' que hace referencia a la ventana que se desea activar, en el caso de que ya esté activada, la situación no cambiará.

- **SetOption:**

Permite cambiar algunas opciones mientras un maplet está en ejecución. Por ejemplo, si un usuario hace clic en un botón, la opción 'oncharge' de dicho botón puede hacer uso del elemento SetOption para iniciar un cambio como borrar una caja de texto.

- **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.

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. 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.

- **AlertDialog:**

Advierte de un riesgo potencial. Permite al usuario elegir entre la opción de continuar (OK) o parar (Cancel).

- **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

Para ejecutar un maplet, se debe utilizar la función Display. Se puede hacer de la siguiente manera:

```
> Maplets[Display](Maplet["Hello world", Button ("OK",Shutdown())]);
```

Para guardar un maplet en un archivo se debe hacer aisladamente del resto de instrucciones, además a la hora de guardarlo se hará poniendo la opción guardar como maplet. Para ejecutarlo, además de la manera explicada anteriormente, 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.

9- CODE GENERATION PACKAGE

Una de las ventajas de la programación con Maple es la posibilidad de traducir el lenguaje a ANSI C, Fortran 77 o Java, sin perder las prestaciones. Para ello, existe un paquete con funciones básicas, llamado CodeGeneration. Por otra parte, las funciones más específicas como las contenidas en subpaquetes, pueden no traducirse correctamente.

9.1. OPCIONES DE CODEGENERATION

Hay multitud de opciones disponibles en el paquete, las cuales se pueden consultar mediante la ayuda.

Cada función de CodeGeneration puede ser ejecutada utilizando notación larga o corta indistintamente. En el caso de que no se haya activado el paquete CodeGeneration (with(CodeGeneration)), se deberá acceder a las funciones en su notación larga, de la siguiente manera:

```
CodeGeneration[C](arguments)
```

Además el paquete CodeGeneration incluye un módulo con funciones, utilizando la siguiente notación:

```
CodeGeneration:-function
```

Una de las más interesantes opciones digna de mención es la opción 'optimize', que reduce el código innecesario aumentando la velocidad de ejecución del programa.

9.2. FUNCIONES DE CODEGENERATION

Hay tres funciones únicas en este paquete, cada una de ellas traduce el código al lenguaje correspondiente, dichas funciones son: C, Fortran y Java, que se detallan a continuación:

9.2.1. Función 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);
  }
}
```