



Escuela Superior de Ingenieros Industriales  
Industri Injineruen Goimailako Eskola

UNIVERSIDAD DE NAVARRA - NAFARROAKO UNIBERTSITATEA

# Aprenda Maple V Release 5

*como si estuviera en primero*

San Sebastián, Octubre 1998

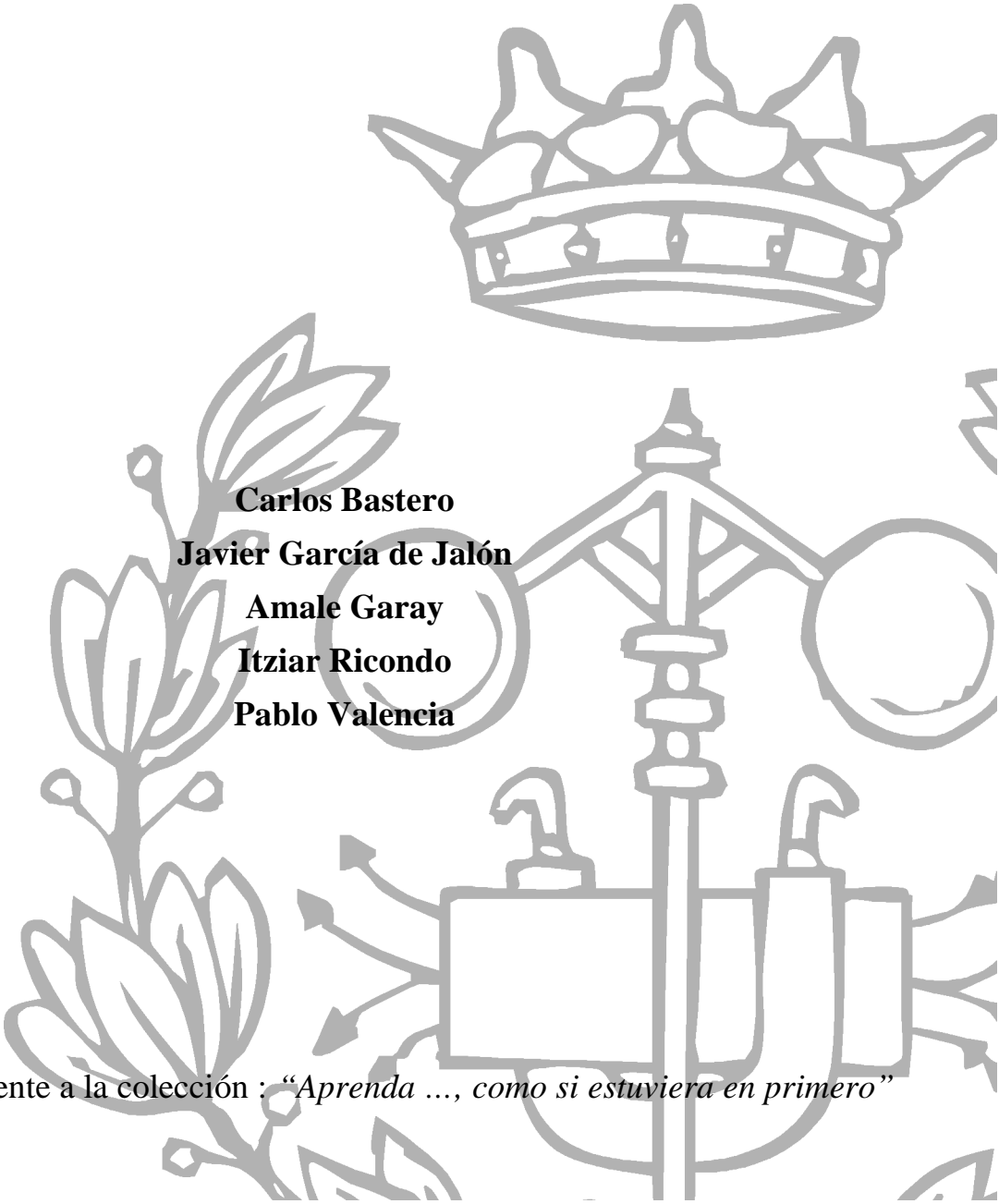


Carlos Bastero • Javier García de Jalón  
Amale Garay • Itziar Ricondo • Pablo Valencia



# Aprenda MAPLE V Release 5

*como si estuviera en primero*



**Carlos Bastero**  
**Javier García de Jalón**  
**Amale Garay**  
**Itziar Ricondo**  
**Pablo Valencia**

Perteneciente a la colección : “Aprenda ..., como si estuviera en primero”

<b>1.</b>	<b>INTRODUCCIÓN A MAPLE V Y AL CÁLCULO SIMBÓLICO .....</b>	<b>1</b>
1.1.	¿QUÉ ES MAPLE V? .....	1
1.2.	ALGUNOS EJEMPLOS INTRODUCTORIOS .....	2
1.2.1.	<i>Formatos de entrada y de salida</i> .....	2
1.2.2.	<i>Fin de sentencia y comentarios</i> .....	2
1.2.3.	<i>Recuperación de los últimos resultados del programa</i> .....	3
1.2.4.	<i>Algunos ejemplos sencillos</i> .....	3
1.2.4.1.	Cálculos .....	3
1.2.4.2.	Generación de ficheros en otros lenguajes .....	6
1.2.4.3.	Gráficos .....	8
1.2.4.4.	Álgebra matricial .....	9
1.2.4.5.	Hojas de cálculo (Spreadsheets) .....	10
<b>2.</b>	<b>DESCRIPCIÓN GENERAL DE MAPLE V .....</b>	<b>11</b>
2.1.	EL HELP DE MAPLE .....	11
2.2.	NÚCLEO, LIBRERÍAS E INTERFACE DE USUARIO .....	11
2.3.	HOJA DE TRABAJO .....	12
2.3.1.	<i>Organización de una hoja de trabajo. Grupos de regiones</i> .....	12
2.3.2.	<i>Edición de hojas de trabajo</i> .....	14
2.3.3.	<i>Modos de trabajo</i> .....	14
2.3.4.	<i>Estado interno del programa</i> .....	14
2.3.5.	<i>Sistema de ventanas de Maple V</i> .....	14
2.3.6.	<i>Librerías</i> .....	15
2.4.	OBJETOS DE MAPLE .....	15
2.4.1.	<i>Números y variables</i> .....	15
2.4.2.	<i>Cadenas de caracteres</i> .....	16
2.4.3.	<i>Operador de concatenación (.)</i> .....	17
2.4.4.	<i>Constantes predefinidas</i> .....	17
2.4.5.	<i>Expresiones y ecuaciones</i> .....	17
2.4.6.	<i>Secuencias o Sucesiones</i> .....	17
2.4.7.	<i>Conjuntos (sets)</i> .....	18
2.4.8.	<i>Listas (lists)</i> .....	19
2.5.	FUNCIONES MEMBER, SORT, SUBSOP Y SUBS .....	20
2.6.	FUNCIONES CONVERT Y MAP .....	21
2.7.	VARIABLES EVALUADAS Y NO-EVALUADAS .....	21
<b>3.</b>	<b>POLINOMIOS Y FRACCIONES RACIONALES.....</b>	<b>27</b>
3.1.	POLINOMIOS DE UNA Y MÁS VARIABLES .....	27
3.1.1.	<i>Polinomios de una variable</i> .....	27
3.1.2.	<i>Polinomios de varias variables</i> .....	29
3.2.	FUNCIONES RACIONALES .....	30
3.3.	TRANSFORMACIONES DE POLINOMIOS Y EXPRESIONES RACIONALES .....	31
3.4.	OTRA FORMA DE OPERAR CON POLINOMIOS Y FRACCIONES RACIONALES .....	32
<b>4.</b>	<b>ECUACIONES Y SISTEMAS DE ECUACIONES. INECUACIONES.....</b>	<b>33</b>
4.1.	RESOLUCIÓN SIMBÓLICA .....	33
4.2.	RESOLUCIÓN NUMÉRICA .....	34
<b>5.</b>	<b>PROBLEMAS DE CÁLCULO DIFERENCIAL E INTEGRAL .....</b>	<b>35</b>
5.1.	CÁLCULO DE LÍMITES .....	35
5.2.	DERIVACIÓN DE EXPRESIONES .....	36
5.3.	INTEGRACIÓN DE EXPRESIONES .....	38
5.4.	DESARROLLOS EN SERIE .....	40
<b>6.</b>	<b>OPERACIONES CON EXPRESIONES.....</b>	<b>41</b>

6.1. SIMPLIFICACIÓN DE EXPRESIONES .....	41
6.1.1. <i>Función expand</i> .....	41
6.1.2. <i>Función combine</i> .....	42
6.1.3. <i>Función simplify</i> .....	43
6.2. MANIPULACIÓN DE EXPRESIONES .....	44
6.2.1. <i>Función normal</i> .....	44
6.2.2. <i>Función factor</i> .....	44
6.2.3. <i>Función convert</i> .....	44
6.2.4. <i>Función sort</i> .....	45
<b>7. FUNCIONES DE ÁLGEBRA LINEAL.....</b>	<b>46</b>
7.1. LIBRERÍA LINALG.....	46
7.2. VECTORES Y MATRICES .....	46
7.3. FUNCIÓN EVALM Y OPERADOR MATRICIAL &* .....	49
7.4. INVERSA Y POTENCIAS DE UNA MATRIZ.....	50
7.5. COPIA DE MATRICES.....	50
7.6. FUNCIONES BÁSICAS DE ÁLGEBRA LINEAL. ....	51
7.6.1. <i>Función matadd</i> .....	51
7.6.2. <i>Función charmat</i> .....	52
7.6.3. <i>Función charpoly</i> .....	52
7.6.4. <i>Funciones colspace y rowspace</i> .....	52
7.6.5. <i>Función crossprod</i> .....	52
7.6.6. <i>Función det</i> .....	52
7.6.7. <i>Función dotprod</i> .....	53
7.6.8. <i>Función eigenvals</i> .....	53
7.6.9. <i>Función eigenvects</i> .....	53
7.6.10. <i>Función gausselim</i> .....	53
7.6.11. <i>Función inverse</i> .....	54
7.6.12. <i>Función iszero</i> .....	54
7.6.13. <i>Función linsolve</i> .....	54
7.6.14. <i>Función multiply</i> .....	54
7.6.15. <i>Función randmatrix</i> .....	54
7.6.16. <i>Función rank</i> .....	54
7.6.17. <i>Función trace</i> .....	54
<b>8. GRÁFICOS EN 2 Y 3 DIMENSIONES.....</b>	<b>55</b>
8.1. GRÁFICOS BIDIMENSIONALES.....	55
8.1.1. <i>Expresiones de una variable</i> .....	55
8.1.2. <i>Funciones paramétricas</i> .....	56
8.1.3. <i>Dibujo de líneas poligonales</i> .....	56
8.1.4. <i>Otras funciones de la librería plots</i> .....	56
8.1.5. <i>Colores y otras opciones de plot</i> .....	57
8.2. GRÁFICOS TRIDIMENSIONALES.....	58
8.2.1. <i>Expresiones de dos variables</i> .....	58
8.2.2. <i>Otros tipos de gráficos 3-D</i> .....	59
8.3. ANIMACIONES .....	60
<b>9. FUNCIONES DEFINIDAS MEDIANTE EL OPERADOR FLECHA (-&gt;) .....</b>	<b>61</b>
9.1. FUNCIONES DE UNA VARIABLE .....	61
9.2. FUNCIONES DE DOS VARIABLES .....	62
9.3. CONVERSIÓN DE EXPRESIONES EN FUNCIONES.....	63
9.4. OPERACIONES SOBRE FUNCIONES .....	64
9.5. FUNCIONES ANÓNIMAS .....	65
<b>10. ECUACIONES DIFERENCIALES .....</b>	<b>65</b>
10.1. INTEGRACIÓN DE ECUACIONES DIFERENCIALES ORDINARIAS .....	65
10.2. INTEGRACIÓN DE ECUACIONES DIFERENCIALES EN DERIVADAS PARCIALES.....	66

10.2.1. Integración de ecuaciones diferenciales en derivadas parciales homogéneas.....	66
10.2.2. Integración de ecuaciones diferenciales en derivadas parciales no homogéneas.....	67
10.2.3. Representación de las soluciones .....	67
<b>11. PROGRAMACIÓN .....</b>	<b>68</b>
11.1. ESTRUCTURAS DE PROGRAMACIÓN.....	68
11.1.1. Bifurcaciones: sentencia if.....	68
11.1.2. Bucles: sentencia for.....	69
11.1.3. Bucles: sentencia while .....	70
11.1.4. Bucles: sentencia for-in.....	70
11.1.5. Sentencias break y next .....	71
11.2. PROCEDIMIENTOS: DEFINICIÓN .....	71
11.2.1. Parámetros.....	72
11.2.2. Variables locales y variables globales.....	72
11.2.3. Options .....	74
11.2.4. El campo de descripción .....	74
11.3. PROCEDIMIENTOS: VALOR DE RETORNO .....	75
11.3.1. Asignación de valores a parámetros.....	75
11.3.2. Return explícito.....	75
11.3.3. Return de error .....	75
11.4. GUARDAR Y RECUPERAR PROCEDIMIENTOS.....	77
11.5. EJEMPLO DE PROGRAMACIÓN CON PROCEDIMIENTOS.....	77
11.6. EL DEBUGGER.....	80
11.6.1. Sentencias de un procedimiento .....	80
11.6.2. Breakpoints.....	81
11.6.3. Watchpoints .....	81
11.6.4. Watchpoints de error.....	82
11.6.5. Otros comandos .....	82
<b>12. COMANDOS DE ENTRADA/SALIDA.....</b>	<b>83</b>
12.1. CONTROL DE LA INFORMACIÓN DE SALIDA.....	83
12.2. LAS LIBRERÍAS DE MAPLE.....	83
12.3. GUARDAR Y RECUPERAR EL ESTADO DE UNA HOJA DE TRABAJO.....	83
12.4. LECTURA Y ESCRITURA DE FICHEROS.....	84
12.5. FICHEROS FORMATEADOS .....	85
12.6. GENERACIÓN DE CÓDIGO.....	85

Los ejemplos de Maple se pueden obtener de la dirección de internet:

[http://www1.ceit.es/Asignaturas/Ecsdif2/Manual\\_Maple/Maple.htm](http://www1.ceit.es/Asignaturas/Ecsdif2/Manual_Maple/Maple.htm)

Sugerencias y comentarios se pueden dirigir a la dirección de e-mail:

[cbastero@ceit.es](mailto:cbastero@ceit.es)

## 1. INTRODUCCIÓN A MAPLE V Y AL CÁLCULO SIMBÓLICO

El programa que se describe en este manual es probablemente muy diferente a todo lo que se ha 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.

Este manual es una re-elaboración y ampliación del manual del mismo título de la Release 3 realizado por **Javier García de Jalón, Rufino Goñi Lasheras, Francisco Javier Funes, Iñigo Girón Legorburu, Alfonso Brazález Guerra y José Ignacio Rodríguez Garrido**. Se han modificado las partes del programa que han cambiado al cambiar la versión y se han desarrollado otros aspectos que han sido implementados en la Release 5.

### 1.1. ¿QUÉ ES MAPLE V?

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 *M*athematical *P*Leasure. 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 ESIISS en Octubre de 1994 fue la Release 3. La versión instalada actualmente es la Release 5, que tiene algunas mejoras respecto a la versión anterior.

Para arrancar Maple desde *Windows NT* o *Windows 95* 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 V.

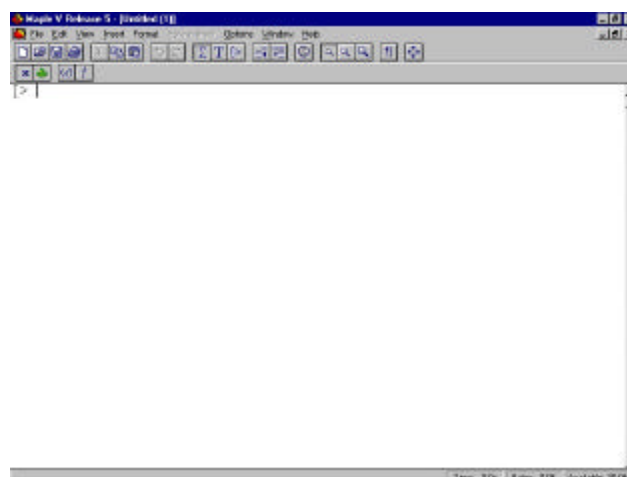


Figura 1. Ventana principal de Maple V.

## 1.2. ALGUNOS EJEMPLOS INTRODUCTORIOS

Maple tiene bastantes peculiaridades. Antes de empezar a exponerlas con un cierto detalle, es conveniente presentar algunos ejemplos sencillos que den una idea de qué es y para qué sirve. De todas formas, antes de ver estos ejemplos, se van a dar unas breves explicaciones sobre cómo funciona esta aplicación. Aunque no es estrictamente necesario, se recomienda leer las secciones siguientes junto a un PC con Maple instalado en el que se puedan reproducir los distintos ejemplos.

### 1.2.1. Formatos de entrada y de salida

La ventana que se abre al arrancar Maple se conoce con el nombre de *worksheet* (hoja de trabajo). En esta hoja se permite disponer de *zonas* o *regiones* de *entrada*, de *salida* y de *texto*. 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”. La característica de esta opción es que pueden escribirse las sentencias matemáticas con ayuda de las paletas que se desprenden del menú *View/Palettes*, de una manera más clara que la que se tiene con la notación propia de 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.

Para guardar estos formatos, se debe elegir el comando *Save Settings* en el menú *File*.

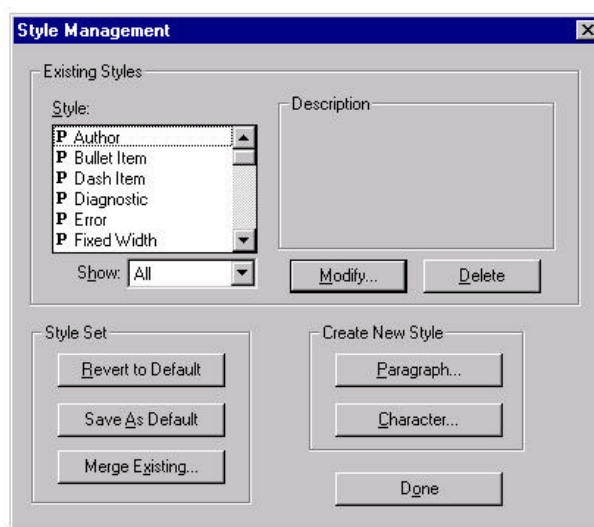


Figura 2. Selección del tipo de letra para las diferentes regiones.

### 1.2.2. Fin de sentencia y comentarios

Si se trabaja en el modo *Maple Notation*, todas las sentencias deben terminar con un carácter *punto y coma* (;). En esto Maple se parece más a C que a MATLAB. De hecho, si no se pone el carácter de terminación y se pulsa *Intro*, el programa seguirá esperando a que se complete la instrucción—dará un *warning* advirtiendo que no se ha cerrado la instrucción—. Esto implica que Maple permite escribir en varias líneas. Si alguna vez sucede eso, se puede poner el carácter (;) en la línea

en la que se esté y volver a pulsar **Intro**. Si se está trabajando en el modo **Standard Math** no es necesaria la terminación **punto y coma (;)**.

También se puede utilizar el carácter **dos puntos (:)** como terminación de línea, pero en este caso no se imprime ninguna salida en la pantalla. Siempre que interese ver qué es lo que hace el programa, se debe utilizar el punto y coma (;) como terminación de línea. Cuando se está aprendiendo a manejar el programa, es útil disponer de toda la información de salida, que además en Maple está especialmente bien presentada.

### 1.2.3. Recuperación de los últimos resultados del programa

El último resultado o salida obtenido del programa puede recuperarse para ser utilizado en una instrucción posterior por medio del carácter **porcentaje (%)**. De forma análoga, **(%%)** representa el penúltimo resultado y **(%%%)** el antepenúltimo resultado. Esto es muy útil para poder utilizar un resultado en el comando siguiente sin haber tenido que asignarlo a ninguna variable.

### 1.2.4. Algunos ejemplos sencillos

#### 1.2.4.1. Cálculos

Para empezar diremos que Maple puede actuar como una simple calculadora numérica. Teclee la siguiente sentencia (a continuación incluimos el resultado que debe aparecer):

```
> sin(5.34*Pi/2);
```

$\sin(2.670000000 \pi)$

Vemos que en realidad no ha hecho lo que esperábamos que hiciera. Esto es porque Maple busca siempre no cometer errores numéricos (errores de redondeo en las operaciones aritméticas), y la forma más segura de evitarlo es dejar para más adelante el realizar las operaciones aritméticas (la división de 5.34 entre 2 sí ha sido realizada pues no introduce errores). Las operaciones aritméticas serán realizadas cuando el usuario lo decida, por ejemplo mediante el comando **evalf**:

```
> evalf(sin(5.34*Pi/2));
```

.8607420265

Otra forma de realizar la operación es clicando con el botón derecho sobre la expresión de salida de Maple — $\sin(2.670000000 \pi)$ — y clicar **Approximate** del menú que se despliega, eligiendo después el número de cifras totales que se quiere que tenga el resultado.

Con Maple se puede controlar fácilmente la precisión de los cálculos (por defecto, calcula con 10 cifras decimales). Por ejemplo, ejecute los siguientes comandos:

```
> Digits := 20;
```

*Digits := 20*

```
> evalf(sin(5.34*Pi/2));
```

.86074202700394363711

```
> Digits := 10;
```

*Digits := 10*

donde el último comando devuelve la precisión a su valor por defecto. 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.

En Maple, el operador de asignación es  $(:=)$ . Como ya se ha dicho, todas las sentencias de Maple deben terminar con  $(;)$  ó  $(:)$ . Recuerde que con  $(:)$  se suprime la línea o líneas de salida correspondientes, aunque Maple realice los cálculos u operaciones indicadas. Se pueden poner varios comandos o sentencias en la misma línea de entrada; cada comando debe finalizar con  $(;)$  o  $(:)$ .

```
> 100/3: 100/6; (24+7/3);
```

$$\frac{50}{3}$$

$$\frac{79}{3}$$

donde se puede observar que el comando 100/3: no produce salida. Maple abre líneas de continuación hasta que se teclea el carácter de terminación  $(;)$ . Observe el siguiente ejemplo:

```
> (2.4*13)+
> 15;
```

46.2

Con números enteros, Maple utiliza "matemáticas exactas", es decir, mantiene las divisiones en forma de cociente, las raíces en forma de raíz, etc. y trata de simplificar al máximo las expresiones aritméticas que se le proporcionan:

```
> 2^8/5+3^(1/2);
```

$$\frac{256}{5} + \sqrt{3}$$

Como ya se ha visto, para forzar la obtención de un resultado numérico en coma flotante se utiliza la función **evalf**:

```
> evalf(2^8/5+3^(1/2));
```

52.93205081

Se van a ver a continuación algunos ejemplos de cálculo simbólico, que es la verdadera especialidad de Maple: teclee los siguientes comandos, ahorrándose, si lo desea, la parte de comentarios:

```
> ec := a*x^2 + b*x + c; # se define una ecuación de segundo grado
```

$$ec := ax^2 + bx + c$$

```
> sols := solve(ec, x); # se llama a una función que resuelve la ecuación
```

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

```
> sols[1]; # se hace referencia a la primera de las dos soluciones
```

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

```
> subs(x=sols[2], ec); # se sustituye la segunda solución en la ecuación
```

$$\frac{1}{4} \frac{(-b - \sqrt{b^2 - 4ac})^2}{a} + \frac{1}{2} \frac{b(-b - \sqrt{b^2 - 4ac})}{a} + c$$

```
> normal(%); # se simplifica el resultado anterior representado por (%)
```

0

```
> int(1+y+4*y^2, y) := int(1+y+4*y^2, y); # integral de un polinomio
```

$$\int 1 + y + 4y^2 dy := y + \frac{1}{2}y^2 + \frac{4}{3}y^3$$

Este resultado merece una pequeña explicación: en el comando ejecutado para obtener este resultado, la integral figura a ambos lados del operador de asignación, aunque sólo se ha ejecutado la parte de la derecha. El ponerlo también a la izquierda sirve para mejorar la presentación del resultado, pues la integral sin efectuar se imprime como primer miembro de la asignación. Véanse los ejemplos siguientes:

```
> diff(%, y); # derivando para comprobar la respuesta anterior
```

$$1 + y + 4y^2$$

```
> diff(%, y); # volviendo a derivar respecto a y
```

$$1 + 8y$$

A continuación se presentan algunos otros ejemplos de interés. Se anima al lector a que teclee los siguientes comandos y compruebe los consiguientes resultados:

```
> f := arctan((2*x^2-1)/(2*x^2+1));
```

$$f := \arctan\left(\frac{2x^2 - 1}{2x^2 + 1}\right)$$

```
> derivada := diff(f, x); # derivada de f respecto a x
```

$$\text{derivada} := \frac{4 \frac{x}{2x^2 + 1} - 4 \frac{(2x^2 - 1)x}{(2x^2 + 1)^2}}{1 + \frac{(2x^2 - 1)^2}{(2x^2 + 1)^2}}$$

```
> normal(derivada); # simplificación del cociente
```

$$4 \frac{x}{4x^4 + 1}$$

```
> int(%, x); # integral del resultado anterior respecto a x
```

$$\arctan(2x^2)$$

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

$$4 \frac{x}{4x^4 + 1}$$

```
> integrate(f, x); # integrate es sinónimo de int
```

$$x \arctan\left(\frac{2x^2-1}{2x^2+1}\right) - \frac{1}{4} \ln(2x^2-2x+1) - \frac{1}{2} \arctan(2x-1) + \frac{1}{4} \ln(2x^2+2x+1) - \frac{1}{2} \arctan(2x+1)$$

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

$$\arctan\left(\frac{2x^2-1}{2x^2+1}\right) + \frac{x \left( 4 \frac{x}{2x^2+1} - 4 \frac{(2x^2-1)x}{(2x^2+1)^2} \right)}{1 + \frac{(2x^2-1)^2}{(2x^2+1)^2}} - \frac{1}{4} \frac{4x-2}{2x^2-2x+1} - \frac{1}{1+(2x-1)^2}$$

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

```
> normal(%);
```

$$\arctan\left(\frac{2x^2-1}{2x^2+1}\right)$$

Como se ve en los ejemplos anteriores, Maple permite manejar y simplificar expresiones algebraicas verdaderamente complicadas.

#### 1.2.4.2. Generación de ficheros en otros lenguajes

Una de las posibilidades más interesantes de Maple es la de generar ficheros fuente de C (y/o FORTRAN) para evaluar las expresiones resultantes. Esto se puede hacer utilizando el menú contextual o introduciendo las órdenes por el teclado. Ejecute los siguientes ejemplos y observe los resultados:

```
> res := integrate(f, x);
```

$$res := x \arctan\left(\frac{2x^2-1}{2x^2+1}\right) - \frac{1}{4} \ln(2x^2-2x+1) - \frac{1}{2} \arctan(2x-1) + \frac{1}{4} \ln(2x^2+2x+1) - \frac{1}{2} \arctan(2x+1)$$

```
> precision := double;
```

*precision := double*

```
> fortran(res, 'optimized');
```

```
t1 = x**2
t2 = 2.DO*t1
t7 = datan((t2-1.DO)/(t2+1.DO))
t9 = 2.DO*x
t11 = dlog(t2-t9+1.DO)
t14 = datan(t9-1.DO)
t17 = dlog(t2+t9+1.DO)
t20 = datan(t9+1.DO)
t22 = x*t7-t11/4.DO-t14/2.DO+t17/4.DO-t20/2.DO
```

```

> readlib(cost) (res);
      12 additions + 17 multiplications + divisions + 5 functions
> readlib(C);
      proc() ... end
> C(res);
      t0 = x*atan((2.0*x*x-1.0)/(2.0*x*x+1.0))-log(2.0*x*x-2.0*x+1.0)/4-atan(
      2.0*x-1.0)/2+log(2.0*x*x+2.0*x+1.0)/4-atan(2.0*x+1.0)/2;
> C(res, 'optimized');
      t1 = x*x;
      t2 = 2.0*t1;
      t7 = atan((t2-1.0)/(t2+1.0));
      t9 = 2.0*x;
      t11 = log(t2-t9+1.0);
      t14 = atan(t9-1.0);
      t17 = log(t2+t9+1.0);
      t20 = atan(t9+1.0);
      t22 = x*t7-t11/4.0-t14/2.0+t17/4.0-t20/2.0;

```

Maple tiene también la posibilidad de escribir el código FORTRAN o C resultante en un fichero de disco. Considérese un ejemplo un poco más complicado:

```

> ecuacion := x^3 - 5*a*x^2 = 1;
      ecuacion :=  $x^3 - 5ax^2 = 1$ 
> solucion := solve(ecuacion, x): #Se omite la salida

```

Ejecutando ahora la siguiente instrucción, Maple crea un fichero llamado *file.c* en el directorio *temp* del disco *c:*. Es fácil de comprobar que dicho fichero contine el código C necesario para la primera de las soluciones de la ecuación anterior (es necesario poner una doble barra invertida porque la primera es interpretada como un carácter de escape que afecta al carácter que le sigue).

```

> C(solucion[1], filename = `c:\\temp\\file.c`);

```

El código puede ser optimizado, por ejemplo:

```

> C(solucion[2], optimized);
      t1 = a*a;
      t2 = t1*a;
      t6 = sqrt(81.0+1500.0*t2);
      t9 = pow(108.0+1000.0*t2+12.0*t6,0.3333333333333333);
      t12 = t1/t9;
      t15 = sqrt(3.0);
      t21 = -t9/12.0-25.0/3.0*t12+5.0/3.0*a+sqrt(-1.0)*t15*(t9/6.0-50.0/3.0*t12
)/2.0;

```

y, como antes, guardado en un fichero de disco con este formato:

```

> C(solucion[1], filename = `c:\\temp\\file2.c`);

```

Se pueden calcular las operaciones necesarias para evaluar la expresión anterior, sin y con optimización:

```
> readlib(cost) (solucion[1]);
```

**8 additions + 23 multiplications + 8 functions**

```
> cost(optimize(solucion[1]));
```

**5 additions + 10 multiplications + divisions + 4 functions + 5 assignments**

### 1.2.4.3. Gráficos

Maple tiene también grandes capacidades gráficas, en algunos aspectos superiores a las de MATLAB. A continuación se van a ver dos ejemplos con funciones que aparecen en el estudio de las vibraciones de sistemas con un grado de libertad. Se incluyen las ventanas gráficas resultantes (figuras 3 y 4).

– Dibujo en dos dimensiones:

```
> xi:= 0.2: formula := beta -> 1/((1-beta^2)^2 + (2*xi*beta)^2)^0.5;
```

$$formula := \beta \rightarrow \frac{1}{\left( (1 - \beta^2)^2 + 4 \xi^2 \beta^2 \right)^{.5}}$$

```
> plot(formula, 0.0..5.0);
```

– Dibujo en tres dimensiones:

```
> form := (beta, xi) -> 1/((1-beta^2)^2 + (2*xi*beta)^2)^0.5;
```

$$form := (\beta, \xi) \rightarrow \frac{1}{\left( (1 - \beta^2)^2 + 4 \xi^2 \beta^2 \right)^{.5}}$$

```
> plot3d(form, 0..5, 0.05..1);
```

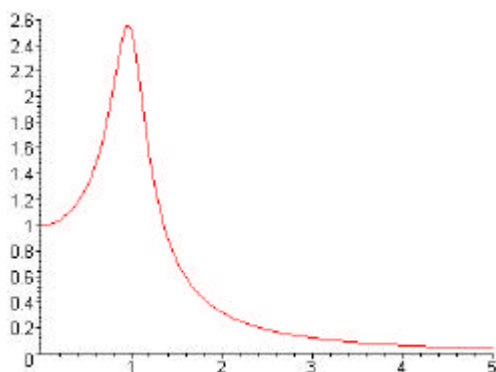


Figura 3. Función plana.

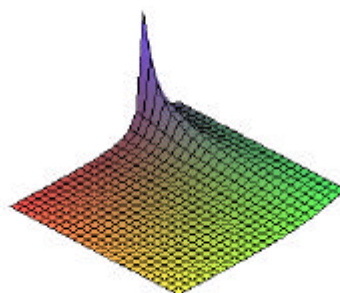


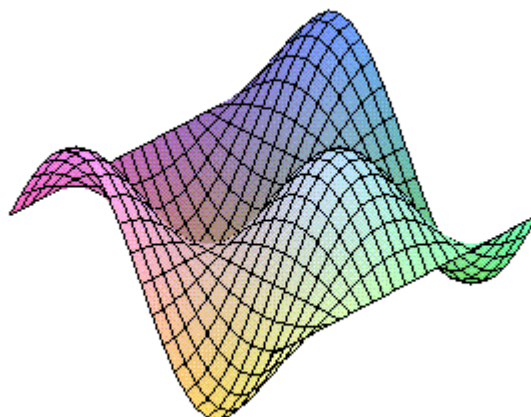
Figura 4. Función tridimensional.

También podría obtenerse la gráfica de una ecuación clicando con el botón derecho del ratón sobre la **ecuación** (distinto de función) de salida de Maple y seleccionando del cuadro de diálogo que aparece la opción **Plot**. Haga la prueba tecleando:  $ec:=x^2*\sin(x)$ ;

El programa también permite hacer animaciones en dos y tres dimensiones. Teclee las siguientes sentencias:

```
> restart;
> with(plots);
> animate3d( cos(t*x)*sin(t*y), x=-Pi..Pi, y=-Pi..Pi, t=1..2 );
```

Aparecerá inicialmente la figura que se muestra a continuación. Clicando sobre el botón de reproducción comienza la animación.



Pueden verse este y otros ejemplos representativos en *Help/New User's Tour* en el apartado *Graphics*.

#### 1.2.4.4. Álgebra matricial.

De forma análoga, Maple puede trabajar también con matrices de modo simbólico. Se presenta a continuación un ejemplo muy sencillo. Ejecute los siguientes comandos y observe los resultados:

```
> with(linalg): # se carga la librería de funciones de álgebra lineal
> A := matrix( 3, 3, [[1-x, 0, x], [0,-x, 1+x], [0, 1, x]] );
```

$$A := \begin{bmatrix} 1-x & 0 & x \\ 0 & -x & 1+x \\ 0 & 1 & x \end{bmatrix}$$

```
> B := inverse(A);
```

$$B := \begin{bmatrix} -\frac{1}{-1+x} & \frac{x}{(-1+x)(x^2+1+x)} & \frac{x^2}{(-1+x)(x^2+1+x)} \\ 0 & -\frac{x}{x^2+1+x} & \frac{1+x}{x^2+1+x} \\ 0 & \frac{1}{x^2+1+x} & \frac{x}{x^2+1+x} \end{bmatrix}$$


```
> d := det(A);
```

$$d := (1-x)(-x^2-1-x)$$

### 1.2.4.5. Hojas de cálculo (Spreadsheets).


Esta es una de las innovaciones que presenta el programa respecto a la versión anterior. Se trata de 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:  $x^{(\sim A1)}$ . Con ( $\sim A1$ ) nos referimos a la casilla 'A1'—el símbolo  $\sim$  se puede obtener tecleando **126** mientras se mantiene pulsada la tecla **Alt**—. Seleccione las cuatro primeras casillas de la segunda columna y repita el proceso anterior con el botón . En la casilla 'C1' teclee:  $\text{int}(\sim B1, x)$ . En la 'D1' teclee:  $\text{diff}(\sim C1, x)$  y arrastre de nuevo las expresiones hacia abajo. La hoja de cálculo que se obtiene es la siguiente:

	A	B	C	D
1	1	$x$	$\frac{1}{2}x^2$	$x$
2	3	$x^3$	$\frac{1}{4}x^4$	$x^3$
3	5	$x^5$	$\frac{1}{6}x^6$	$x^5$
4	7	$x^7$	$\frac{1}{8}x^8$	$x^7$

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 .

Seguro que estos ejemplos le habrán servido para empezar a ver qué cosas es Maple capaz de realizar. Maple es un programa que puede ser de gran utilidad en las áreas más diversas de la ingeniería.

## 2. DESCRIPCIÓN GENERAL DE MAPLE V

### 2.1. EL HELP DE MAPLE

El **help** de Maple se parece al de las demás aplicaciones de **Windows**, aunque tiene también algunas peculiaridades que conviene conocer. Además de poder explorar el menú **Help** de la ventana principal del programa, se puede pedir ayuda sobre un comando concreto desde la hoja de trabajo tecleando ¿comando.

Por ejemplo ?int

El método anterior abre una ventana con toda la información disponible sobre dicho comando. 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)
```

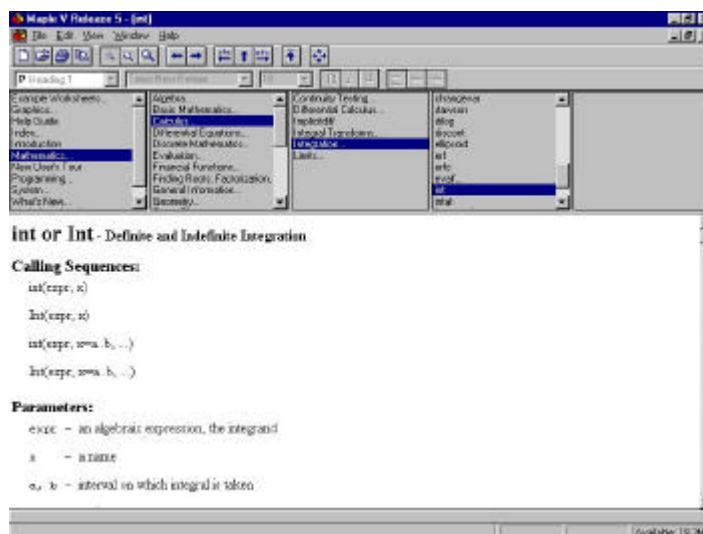


Figura 3. Ventana de Ayuda de Maple V.

que 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 por 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.

### 2.2. NÚCLEO, LIBRERÍAS E INTERFACE DE USUARIO

Maple consta de tres partes principales: el **núcleo** o *kernel*, que es la parte central del programa (escrita en lenguaje C), encargada de realizar las operaciones matemáticas fundamentales; las **librerías**, que son conjuntos de funciones relacionadas que residen en el disco y son llamadas cuando se las necesita; y la **interface de usuario**, que se encarga de todas las operaciones de entrada/salida, y en general, de la comunicación con el exterior.

Como consecuencia, Maple tiene dos tipos de comandos: los que afectan al núcleo y los comandos que se refieren a la interface de usuario.



## 2.3. HOJA 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**.

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. Se puede hacer **Copy** y **Paste** entre dichas hojas de trabajo. 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

De ordinario, estas regiones se visualizan con colores y tipos de letra diferentes, de forma que son muy fáciles de distinguir. Con el comando **Format/Styles** se pueden modificar los tipos de letra de las tres primeras regiones citadas.

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 distintos 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 clicke 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 usuario decide cancelar un comando que ha comenzado ya a ejecutarse, basta clickear sobre el botón **Stop** en la barra de herramientas de la ventana principal, o pulsar las teclas **Control+Pausa**.

Para escribir texto se clicke 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 clickear 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 línea 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:

## Esto es una sección

Esto es un comentario de una sección.

```
[> a:=sin(x);#esto es una sentencia dentro de la sección
```

### Esto es una subsección

Con un comentario

```
[> x:=Pi/2;
```

```
[> evalf(a);
```

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 clicke 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 recalculando 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 **Options** están seleccionadas las opciones **Replace Mode**. 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 **Options**. Dichos modos son los siguientes:

- **Options/Replace Mode**. 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.
- **Options/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 Mode** 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 Worksheet** o **From Selection** respectivamente. También puede ser conveniente ejecutar toda o parte de la hoja de trabajo mediante **Edit/Execute/Worksheet** o **Selection** respectivamente. Cuando se ejecuten estos comandos sobre una selección habrá que realizar la selección previamente, como es obvio.

Con el comando **Save** del menú **File** se almacenan los *resultados externos de la sesión de trabajo* (regiones de entrada, salida, texto y gráficos) en un fichero con extensión **\*.mws**. 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**.

### 2.3.5. Sistema de ventanas de Maple V

Maple tiene 5 tipos diferentes de ventanas:

- 1.– ventana principal (hoja de trabajo). Como hemos dicho, puede haber varias hojas de trabajo abiertas al mismo tiempo. En este caso se tienen en todas las hojas de trabajo las mismas

variables. Es posible sin embargo ejecutar dos sesiones de Maple simultáneamente, cada una con sus variables y su hoja de trabajo independiente.

- 2.— ventana de ayuda. Se activa al solicitar la ayuda del **Help**, bien desde el menú, bien desde la línea de comandos de la hoja de trabajo.
- 3.— ventanas gráficas 2-D. Se abre una ventana nueva cada vez que se ejecuta un comando de dibujo 2-D. Estas ventanas tienen sus propios menús, que permiten modificar de modo interactivo algunas características del dibujo (tipo de línea, de ejes, etc. En este sentido Maple es más versátil y fácil de utilizar que MATLAB).
- 4.— ventanas gráficas 3-D. Se abre también una ventana nueva cada vez que se ejecuta un comando de dibujo 3-D. Estas ventanas tienen también sus propios menús, diferentes de los de las ventanas gráficas 2-D.
- 5.— ventana de animaciones (movies). Éste es el último tipo de ventanas gráficas, diferentes también de las dos anteriores. Disponen de unos mandos similares a los de un equipo de video, para poder ver de diversas formas la animación producida.

### 2.3.6. Librerías

Maple dispone de más de 2000 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 primero. Puede optarse por cargar un comando o función aislado 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 (figura 3) se pueden ver las librerías disponibles en Maple 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). La lista de estas funciones se puede obtener con el comando **index**. Las restantes funciones deben ser cargadas explícitamente por el usuario antes de ser utilizadas. Ésta es una fuente importante de dificultades para los usuarios que comienzan.

## 2.4. OBJETOS DE MAPLE

Los **objetos** de Maple son los *tipos de datos* y *operadores* con los que el programa es capaz de trabajar. A continuación se explican los 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(%);
```

4.709975946676696989353109

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);
```

4.709975946676696989353108872543860109868

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. Dichos nombres deben empezar siempre por una letra o un número, y pueden contener caracteres (`_`). También puede ser un nombre de variable una cadena de caracteres encerrada entre comillas inversas (``comillas inversas``), tal como se define en la siguiente sección.

### 2.4.2. Cadenas de caracteres

Las cadenas de caracteres van encerradas entre comillas inversas (acento grave francés). Véanse algunos ejemplos:

```
> nombre := `Bill Clinton`;
```

```
> duda := `¿es más fácil segundo que primero?`;
```

Dentro de una cadena, una doble comilla inversa equivale a una única comilla inversa. Los caracteres especiales que van dentro de una cadena no son evaluados por Maple.

### 2.4.3. Operador de concatenación (.)

Un punto separando dos nombres, o un nombre y número, actúa como *operador de concatenación*, esto es, el resultado es un único nombre con el punto eliminado. Véanse algunos ejemplos:

```
> a.1, diga.33;
      a1, diga33
> fichero.doc; `fichero.doc`; # ojo con los nombres de ficheros
      fichero.doc
      fichero.doc
```

Como se indica en el ejemplo anterior, este operador exige tomar precauciones especiales con los nombres de ficheros que tienen extensión. Para evitar que elimine el punto, se deben poner entre comillas inversas.

### 2.4.4. Constantes predefinidas

Maple dispone de una serie de constantes predefinidas, entre las que están el número **Pi**, la unidad imaginaria **I**, los valores **infinity** y **-infinity**, y las constantes booleanas **true** y **false**.

### 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.

### 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* y las *listas*.

Una *secuencia* es un conjunto de varias expresiones o datos de cualquier tipo separados por comas. Por ejemplo, se puede crear una secuencia de palabras y números en la forma:

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

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

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

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

Puede comprobarse que utilizando la función *seq* no hace falta poner apóstrofes 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 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. Por ejemplo:

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

marzo

marzo, 22, 33, trabajo, trabajo
```

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. 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
```

## 2.5. FUNCIONES MEMBER, SORT, SUBSOP Y SUBS

La función *member* actúa sobre *sets* y listas, pero no sobre secuencias. Su objeto es averiguar si un determinado dato o expresión pertenece a un *set* o a una lista y, en caso afirmativo, qué posición ocupa. Esta función tiene tres argumentos: la expresión o dato, el *set* o la lista, y una variable no evaluada en la que se pueda devolver la posición. Considérese el siguiente ejemplo:

```
> set3; member(verde, set3, 'pos'); pos;  
  
{ amarillo, rojo, verde, azul, negro }  
  
true  
  
3  
  
> member(blanco, set3, 'pos');  
  
false
```

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). 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:

```
> lista2 := [op(set3)];  
  
lista2 := [ amarillo, rojo, verde, azul, negro ]  
  
> sort(lista2);  
  
[ amarillo, azul, negro, rojo, verde ]
```

Un elemento de una lista no se puede cambiar por medio de un operador de asignación. Por ejemplo, para cambiar “negro” por “blanco” en *lista2* no se puede hacer:

```
> lista2[3] := blanco;
```

sino que hay que utilizar la función *subsop* que realiza la sustitución de un elemento por otro en la forma:

```
> subsop(3=blanco, lista2);
```

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

Si en vez de reemplazar por la posición se desea reemplazar por el propio valor, puede utilizarse la función **subs**, como en el ejemplo siguiente:

```
> lista3 := [op(lista2), negro];
      lista3 := [amarillo, rojo, blanco, azul, negro, negro]

> subs(negro=blanco, lista3);
      [amarillo, rojo, blanco, azul, blanco, blanco]
```

donde los dos elementos “negro” han sido sustituidos por “blanco”. Para más información sobre las funciones introducidas en este apartado puede consultarse el **Help**.

## 2.6. FUNCIONES CONVERT Y MAP

En muchas ocasiones hay que convertir datos de un tipo a otro tipo distinto. La función **convert** permite convertir unos tipos de datos en otros, y en concreto permite realizar conversiones entre sets y listas. 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**.

Para convertir una lista en un *set* por medio de la función **convert** hay que hacer lo siguiente:

```
> convert(lista3, set);
      {amarillo, rojo, azul, negro, blanco}
```

mientras que convertir un *set* en una lista se puede hacer de la forma:

```
> set4 := set3 union {violeta, naranja, verde};
      set4 := {amarillo, rojo, verde, azul, negro, violeta, naranja}

> convert(set4, list);
      [amarillo, rojo, verde, azul, negro, violeta, naranja]
```

La función **map** permite aplicar una misma función a todos los elementos o expresiones de una lista o conjunto. Por ejemplo, en el caso siguiente se crea una lista de dos funciones a las que luego se aplica la función derivada **diff** mediante la función **map**. En este caso es necesario pasar como argumento adicional la variable respecto a la que se quiere derivar, necesaria para aplicar la función **diff**.

```
> lista4 := [x^2-1, x^3+2*x^2+5];
      lista4 := [x^2 - 1, x^3 + 2 x^2 + 5]

> map(diff, lista4, x);
      [2 x, 3 x^2 + 4 x]
```

## 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 (o bien 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;
```

$$\text{polinomio} := 9x^3 - 37x^2 + 47x - 19$$

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;
```

$$x := \text{variable}$$

$$9 \text{ variable}^3 - 37 \text{ variable}^2 + 47 \text{ variable} - 19$$

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

$$\text{variable} := 10$$

$$10$$

$$5751$$

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

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

$$x := x$$

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

Los apóstrofes '**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 apóstrofes 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: i tiene que valer 1
```

$$a_1 := a_i$$

```
> a[i] := evaln(a[i]); a[i]; # ahora sí 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 evaluado 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 que no se asignan. 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<sup>1</sup>:

```
> 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);
```

$$b$$

$$c$$

$$3$$

---

<sup>1</sup> 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.

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

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

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

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

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

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

```
> i:=0: sum(ithprime(i), i=1..5);
Error, (in ithprime) argument must be a positive integer
> sum('ithprime(i)', i=1..5); # esto sólo no arregla el problema
Error, (in sum) summation variable previously assigned,
second argument evaluates to, 0 = 1 .. 5
> sum('ithprime(i)', 'i'=1..5); # ahora sí funciona
28
```

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

```
> x:=1; x+1;
x := 1
2
> 'x'+1; 'x+1';
1 + x
1 + x
```

```
> ''x'+1''; %; %; %; # cada "último resultado" requiere una evaluación
      'x' + 1
      'x' + 1
      1 + x
      2
```

### 3. POLINOMIOS Y FRACCIONES RACIONALES

#### 3.1. POLINOMIOS DE UNA Y MÁS VARIABLES

##### 3.1.1. Polinomios de una variable

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

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

donde **n** es el grado, **a<sub>n</sub>** el primer coeficiente y **a<sub>0</sub>** 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 := -3 x + 7 x^2 - 3 x^3 + 7 x^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' );
      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 := 5 x^5 + 3 x^3 + x^2 - 2 x + 1
> 2*p1 + 4*p2 + 3;
      -14 x + 18 x^2 + 6 x^3 + 14 x^4 + 20 x^5 + 7
> p1 * p2;
      (-3 x + 7 x^2 - 3 x^3 + 7 x^4)(5 x^5 + 3 x^3 + x^2 - 2 x + 1)
```



```
> 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 caso anterior
```

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

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

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

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
```

$$(x - 1)(x^3 + x^2 + x + 1)$$

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 mostrar 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
```

```
> lcoeff(p2), tcoeff(p2); # para obtener los coeficientes de los términos de mayor (leading) y menor grado (trailing)
```

```
> 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
```

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

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

```
> teste(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 [ [r<sub>i</sub>,m<sub>i</sub>], ..., [r<sub>n</sub>,m<sub>n</sub>] ], donde r<sub>i</sub> es la raíz y m<sub>i</sub> 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]]
```

### 3.1.2. 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);
```

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

o *según las potencias de y*:

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

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

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. 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;
```

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

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

```
> ff/gg;
```

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

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);
```

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

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);
```

### 3.3. TRANSFORMACIONES DE POLINOMIOS Y EXPRESIONES RACIONALES

A continuación se verá cómo se puede transformar un polinomio. En primer lugar se va a ver lo que se puede hacer con la **regla de Horner**. Se utilizará un polinomio definido anteriormente:

```
> p1, readlib(cost)(p1); # para evaluar el n° de operaciones aritméticas
```

$$-3x + 7x^2 - 3x^3 + 7x^4, 3 \text{ additions} + 10 \text{ multiplications}$$

```
> convert(p1, 'horner');
```

$$(-3 + (7 + (-3 + 7x)x)x)x$$

```
> cost(%);
```

$$3 \text{ additions} + 4 \text{ multiplications}$$

Las expresiones racionales se pueden transformar en *fracciones continuas*:

```
> fraccion := (x^3 + x^2 - x + 1)/p1;
```

$$\text{fraccion} := \frac{x^3 + x^2 - x + 1}{-3x + 7x^2 - 3x^3 + 7x^4}$$

```
> cost(fraccion);
```

*6 additions + 13 multiplications + divisions*

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

$$\frac{1}{x - \frac{10}{7} + \frac{24}{7} \frac{1}{x + \frac{11}{6} + \frac{1}{9} \frac{1}{x - \frac{71}{24} + \frac{429}{64} \frac{1}{x + \frac{17}{8}}}}$$

```
> cost(%);
```

*7 additions + 4 divisions*

También se puede realizar la transformación en *fracciones parciales* (no se incluyen los resultados):

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

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

Ahora, después de realizar la transformación, es mas fácil realizar ciertas operaciones matemáticas, como por ejemplo la integración indefinida:

```
> integrate(%, x); # integrate es sinónimo de int
```

Un último ejemplo con más de una variable o constante indeterminada es el siguiente:

```
> ratfun := (x-a)/(x^5 + b*x^4 - c*x^2 - b*c*x);
```

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

### 3.4. OTRA FORMA DE OPERAR CON POLINOMIOS Y FRACCIONES RACIONALES

Maple V Release 5 ofrece la posibilidad de realizar multitud de operaciones sobre cualquier objeto de manera muy sencilla. Para ello no hay más que colocar el ratón sobre el objeto y clicar con el botón derecho, de manera que aparece una lista de acciones a realizar llamada **menú contextual**. Se utilizará la fracción definida anteriormente :

```
> racexp;
```

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

El menú desplegado ofrece una amplia gama de acciones a realizar, sin más que seleccionando con el ratón. El lector puede intentar cualquiera de ellas. Observará que Maple escribe automáticamente la sintaxis correspondiente y lo ejecuta.

Si decidimos por ejemplo simplificar clicamos sobre **Normal**, y obtenemos:

```
> R0 :=
normal((161*y^3+333*y^2*x+184*y^2+162*y*x^2+144*x*y+77
*y+99*x+88)/(49*y^2+28*y*x^2+63*x*y+147*y+36*x^3+32*x^
2+117*x+104));
```

$$R0 := \frac{23y^2 + 18xy + 11}{7y + 4x^2 + 13}$$

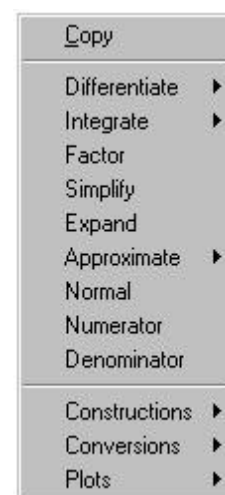


Figura 6. Menú contextual

## 4. ECUACIONES Y SISTEMAS DE ECUACIONES. INECUACIONES.

### 4.1. 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 expresiones con las incógnitas despejadas.

Es posible controlar el número de soluciones (en el caso de soluciones múltiples), mediante el parámetro ***MaxSols***. Por otra parte, si el comando de resolución espera que se le envíe una ecuación, pero lo que se introduce es una expresión ***expr***, Maple supone una ecuación en la forma ***expr=0***.

Si Maple es incapaz de encontrar la solución o ésta no existe, se devuelve el valor NULL. Se pueden asignar las soluciones a variables mediante el comando de asignación ***assign***.

A continuación se presentan algunos ejemplos de ecuaciones e inecuaciones con su solución correspondiente:

```
> solve(cos(x) + y = 9, x); # despeja la variable x en función de y
      pi - arccos(y - 9)
> solve({cos(x) + y = 9}, {x}); x; # "sets" de ecuaciones e incógnitas
      {x = pi - arccos(y - 9)}
      x
```

En el ejemplo anterior, se despeja la variable ***x*** en función de ***y***, y - aparentemente- se asigna la solución a la misma variable ***x***. En realidad no se ha producido tal asignación, como se comprueba al escribir la variable ***x***. Para que se produzca la asignación a ***x*** hay que utilizar el operador ***assign***:

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

> assign(res): x;
                        pi - arccos(y - 9)
```

Recuerde que las llaves indican conjuntos o *sets* de ecuaciones o de variables.

```
> restart;solve(x^3 - 6*x^2 + 11*x - 6, x); # solución exacta
                        {x = 1}, {x = 2}, {x = 3}
```

Maple es capaz de resolver ecuaciones en valor absoluto.

```
> solve( abs( (z+abs(z+2))^2-1 )^2 = 9, {z});
                        {x < -1/2 - 1/2*sqrt(21)}, {-1/2 + 1/2*sqrt(21) < x}
```

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

```
> solve( {x^2+x>5}, {x} );
                        {x < -1/2 - 1/2*sqrt(21)}, {-1/2 + 1/2*sqrt(21) < x}

> solve(({x-1)*(x-2)*(x-3) < 0}, {x});
                        {x < 1}, {2 < x, x < 3}
```

Los sistemas de ecuaciones e inecuaciones se resuelven también de manera sencilla.

```
> solve({x+y=1, 2*x+y=3}, {x,y}); # solución única
                        {y = -1, x = 2}

> solve({a*x^2*y^2, x-y-1}, {x,y}); # soluciones múltiples
                        {y = 0, x = 1}, {x = 0, y = -1}

> _MaxSols := 1:
> solve({a*x^2*y^2, x-y-1}, {x,y}); # encuentra una única solución
                        {y = 0, x = 1}

> solve( {x^2<1, y^2<=1, x+y<1/2}, {x,y} );
                        {x < 1, x + y - 1/2 < 0, -1 <= y, y <= 1, -1 < x}
```

## 4.2. RESOLUCIÓN NUMÉRICA.

En algunos casos puede interesar (o incluso no haber más remedio) resolver las ecuaciones o los sistemas de ecuaciones numéricamente, desechando la posibilidad de hacerlo simbólicamente. Esto es posible con la función ***fsolve***. La función ***fsolve*** resuelve numéricamente únicamente ecuaciones. Esta función intenta encontrar una raíz real en una ecuación no lineal de tipo general, pero en el caso de las ecuaciones polinómicas es capaz de encontrar todas las posibles soluciones. Si se desea obtener también las *raíces complejas*, hay que utilizar el parámetro ***complex***, como en el siguiente ejemplo:

```
> fsolve(x^2+1, x, complex);
```

$$-1. I, 1. I$$

A continuación se pueden ver algunos ejemplos del uso y soluciones de esta función:

```
> fsolve(tan(sin(x))=1, x);
```

$$2.238253543$$

```
> poly := 23*x^5+105*x^4-10*x^2+17*x;
```

$$poly := 23x^5 + 105x^4 - 10x^2 + 17x$$

```
> fsolve(poly, x, -1..1); # halla las raíces en el intervalo dado.
```

$$0, -.6371813185$$

```
> fsolve(poly, x, maxsols=3); # 3 soluciones
```

$$0, -4.536168981, -.6371813185$$

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

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

```
> fsolve(q, x, complex); # halla todas las raíces
```

$$-.6623589786 - .5622795121 I, -.6623589786 + .5622795121 I, 1.324717957, 5.333333333$$

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});
```

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

## 5. PROBLEMAS DE CÁLCULO DIFERENCIAL E INTEGRAL

### 5.1. 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);
-∞

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

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

```

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. Para pasar de una notación a otra seleccione su opción con el comando **Options/Input Display**, o clicando sobre el icono correspondiente (ver al margen).



El último ejemplo, empleando la notación *Standard Math*:

```

> lim tan(x); lim tan(x)
x → (π/2)⁻ x → (π/2)⁺
∞
-∞

```

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 ?
? → ?

```

Ahora sólo tiene que sustituir las interrogaciones ? para *construir* su propio límite.

## 5.2. DERIVACIÓN DE EXPRESIONES

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.

He aquí algunos ejemplos:

```

> diff(x^2, x); # se deriva una función de una sola variable.
2 x

```

```
> diff(x^3, x, x); # se deriva dos veces una función.
```

$$6x$$

```
> diff(x^3*y^2, y); # derivación parcial de una función de 2 variables.
```

$$2x^3y$$

Para derivadas de orden superior se puede utilizar el **carácter de repetición** \$, tal como se muestra en los siguientes ejemplos:

```
> diff(x^6/6!, x$6); # se deriva 6 veces respecto de x.
```

$$1$$

```
> diff((s^3+2*s-5)/(t^2-3*t), s$2, t); # se deriva 2 veces respecto de s y una respecto de t.
```

$$-6 \frac{s(2t-3)}{(t^2-3t)^2}$$

Cuando se deriva una función con el comando **diff** lo que se obtiene es una expresión. Algunas veces interesará que lo que se derive se comporte como una función. Para ello se utiliza el comando **unapply**. Veámoslo con un ejemplo:

```
> f:=x->5*x^3+2*x;
> der := diff(f(x),x);
```

$$der := 15x^2 + 2$$

```
> f_prima:=x->der: #así no lo convertimos en función
> f_prima(2);
```

$$15x^2 + 2$$

```
> f_prima:=unapply(der,x): #para que f_prima sea función hay que aplicar unapply
> f_prima(2);
```

$$62$$

La derivación **parcial** de *funciones de varias variables* puede hacerse del siguiente modo (hay que incluir los argumentos a continuación del nombre de la función):

```
> F:=(x,y)->1+x^2-x*y^3;
```

$$F := (x, y) \rightarrow 1 + x^2 - xy^3$$

```
> diff(F(x,y),x,y);
```

$$-3y^2$$

También pueden derivarse funciones por tramos mediante el comando **piecewise**. Este comando tiene tantos argumentos como condiciones tengamos: **piecewise** (cond\_1,f\_1, cond\_2,f\_2, ..., cond\_n,f\_n, f\_si\_no), y significa que para la condición 1 tenemos f\_1, y así sucesivamente hasta la condición n. Para todos los demás casos la función queda definida por f\_si\_no.

```
> p := x -> piecewise( x<0, -1, x>1, 2*x, x^2 );
```

$$p := x \rightarrow \text{piecewise}(x < 0, -1, 1 < x, 2x, x^2)$$

```
> p(x);
```

$$\begin{cases} -1 & x < 0 \\ 2x & 1 < x \\ x^2 & \text{otherwise} \end{cases}$$

Es interesante conocer dos funciones estrechamente relacionadas con *diff*, *Diff* y *D*. La función *Diff*, cuando se quiere imprimir, devuelve los signos de derivación y hace que la presentación resulte más elegante:

```
> Diff(ln(x),x)= diff(ln(x),x);
```

$$\frac{\partial}{\partial x} \ln(x) = \frac{1}{x}$$

Por su parte, el operador *D* no necesita de variables respecto de las que derivar y, por esa razón, sólo puede aplicarse a funciones. En funciones de varias variables, hay que especificar entre corchetes detrás de la *D* el número de orden de la posición de la variable, respecto de la que se quiere derivar, dentro de la función. Cuando utilizamos el operador *D* para diferenciar, lo que obtenemos es una función.

```
> D(exp+ln+Pi+tan);
```

$$\exp + \left( a \rightarrow \frac{1}{a} \right) + 1 + \tan^2$$

```
> f:=(x,y,z,t)->cos(x)*exp(x*y)*t*cosh(z*t): # definición de una función de
cuatro variables (véase la sección 9)
```

```
> D[3](f); # cálculo de la derivada parcial respecto a la tercera variable
de la función f, es decir respecto de la z
```

$$(x, y, z, t) \rightarrow \cos(x) e^{(x y)} t^2 \sinh(z t)$$

Como se recordará por apartados anteriores, puede utilizar la notación *Standard Math*, clicando sobre el icono correspondiente en la paleta *Expression Palette*.

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

### 5.3. INTEGRACIÓN DE EXPRESIONES

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);
```

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

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-x^2)^(172), x=0..1); # resultado exacto
```

```
35835915874844867368919076489095108449946327955754392558399825615420669938882575126094039 \
```

$$892345713852416 \frac{\Gamma(173)^2}{\Gamma(346)}$$

```
> int(1/(1+x^2), x=0..infinity);
```

$$\frac{1}{2}\pi$$

En el caso de *integrales definidas* se puede añadir una opción "*continous*" 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
```

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

La función **Int** es interesante a la hora de imprimir, ya que devuelve los signos de integración y hace que la presentación resulte más elegante:

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

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

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**.

## 5.4. DESARROLLOS EN SERIE

La función **taylor** permite calcular el desarrollo en serie de una función en un punto determinado y con una precisión también determinada. La forma general de esta función es la siguiente:

```
> taylor(expresion, variable=punto, n);
```

donde **punto** es el valor de la variable en torno al que se hace el desarrollo en serie, y **n** es el grado hasta el que se desea calcular los términos.

Considérese un ejemplo de cómo se usa esta función:

```
> taylor(1/x, x=1, 7);
```

$$1 - (x - 1) + (x - 1)^2 - (x - 1)^3 + (x - 1)^4 - (x - 1)^5 + (x - 1)^6 + O((x - 1)^7)$$

El resultado del desarrollo en serie puede convertirse en polinomio con la función **convert**. A continuación se presenta un nuevo ejemplo:

```
> p := taylor(sin(x), x, 9);
```

$$p := x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + O(x^9)$$

Para convertirlo en polinomio puede usar la función **convert** o clicar con el botón derecho en la salida anterior y elegir **Truncate Series to Polynomial**.

```
> p := convert(p, polynom);
```

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

Cuanto mayor sea el número de términos, mejor será la aproximación en serie de Taylor. A continuación se puede observar la diferencia entre una expresión y su aproximada mediante una gráfica.

```
> expr := sin(4*x)*cos(x):
> aprox := taylor( expr, x=0,10):
> poli := convert( aprox, polynom );
```

$$poli := 4x - \frac{38}{3}x^3 + \frac{421}{30}x^5 - \frac{10039}{1260}x^7 + \frac{246601}{90720}x^9$$

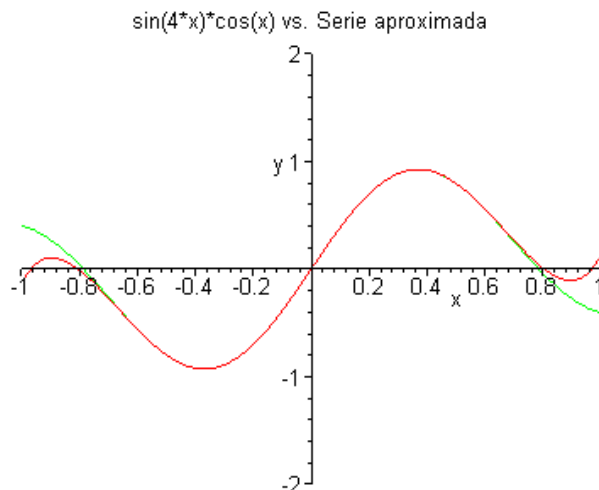


Figura 7. Función real vs. Aproximada

## 6. 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 otra 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.

### 6.1. SIMPLIFICACIÓN DE EXPRESIONES

#### 6.1.1. Función expand

En general, el comando *expand* hace honor a su nombre y expande la función a la forma de suma o producto de otras funciones más sencillas.

```
> cos(2*x): % = expand(%); # expansión de una función trigonométrica
```

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

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

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

Al trabajar con logaritmos hay que especificar el signo de las variables  $x$  e  $y$  mediante el comando *assume*, para garantizar su existencia.

```
> assume(x>0,y>0):ln(x/y): % = expand(%); # expansión de una función
logarítmica
```

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

with assumptions on  $x$  and  $y$

```
> (x^y)^z: % = expand(%); # expansión de una potencia
```

$$(x^y)^z = x^{(yz)}$$

```
> x:='x':y:='y':(n+1)!: % = expand (%); # expansión de un factorial
      (n+1)! = n! (n+1)
```

También se pueden expandir expresiones de un modo parcial, dando como argumento la parte de la expresión que no se quiere expandir. Observe la diferencia que hay entre los resultados de la función *expand* en el siguiente ejemplo:

```
> expresion := sin(x+y) + exp(x+y);
      expresion := sin(x+y) + e(x+y)
> expand(expresion), expand(expresion, sin);
      sin(x) cos(y) + cos(x) sin(y) + ex ey, sin(x+y) + ex ey
```

### 6.1.2. 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. 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) = yn exp(x), para n ∈ Z
(exp x)y = exp (x*y)
a ln x = ln(xa)
ln x + ln y = ln (x*y)
```

*powers*:

```
xy*xz= xy+z
(xy)z = xyz
```

A continuación se presentan algunos ejemplos de aplicación práctica de la función *combine*:

```
> sin(x)^2: " = combine(" , 'trig');
      sin(x)2 = 1/2 - 1/2 cos(2 x)
> xy/x(2/3): " = combine(" , 'power');
      xy / x2/3 = x(y - 2/3)
```

En el caso de compactar expresiones con logaritmos es necesario especificar la naturaleza de los términos (en el ejemplo *x* e *y*) para que el logaritmo exista. Otra posibilidad es añadir la opción *Symbolic* como tercer argumento de la función *combine*.

```
> ln(x)-ln(y): % = combine(%, 'ln'); # no compacta
```

$$\ln(x) - \ln(y) = \ln(x) - \ln(y)$$

```
> assume(x>0,y>0);ln(x)-ln(y): % = combine(%, 'ln');
```

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

with assumptions on  $x$  and  $y$

```
> x:='x':y:='y':ln(x)-ln(y): % = combine(%, 'ln','symbolic');
```

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

### 6.1.3. 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. 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.

```
> sin(x)^2: % = simplify(%, 'trig');
```

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

```
> sinh(x)^3: % = simplify(%);
```

$$\sinh(x)^3 = \sinh(x) \cosh(x)^2 - \sinh(x)$$

Según sea la naturaleza de los términos (introducidos mediante la función **assume**), la simplificación será de una forma o de otra.

```
> assume(x>0):(-x)^y: % = simplify(%); # x>0
```

$$(-x)^y = x^y (-1)^y$$

with assumptions on  $x$

```
> assume(y/2, integer, x>0):(-x)^y: % = simplify(%); #x>0 e y es par ya que y/2 es un número entero
```

$$(-x)^y = x^y$$

with assumptions on  $x$  and  $y$

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

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



## 6.2. MANIPULACIÓN DE EXPRESIONES

Se verán ahora los comandos que al principio de la sección se denominaban *manipulaciones*.

### 6.2.1. Función normal

El comando **normal** permite simplificar numerador y denominador de expresiones algebraicas, hasta lo que se denominan *formas normales factorizadas*, que son polinomios primos (indivisibles) con coeficientes enteros.

```
> normal((x^2-y^2)/(x-y)^3);
```

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

```
> normal((ft(x)^2-1)/(ft(x)-1));
```

$$ft(x) + 1$$

### 6.2.2. Función factor

Este comando permite descomponer un polinomio en factores. Observe los siguientes ejemplos:

```
> p := x^5-3*x^4-x^3+3*x^2-2*x+6;
```

$$p := x^5 - 3x^4 - x^3 + 3x^2 - 2x + 6$$

```
> factor(p); # no consigue hacer nada
```

$$x^5 - 3x^4 - x^3 + 3x^2 - 2x + 6$$

```
> factor((x^3-y^3)/(x^4-y^4));
```

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

El comando **factor** no descompone un número entero en factores primos. Para ello hay que utilizar el comando **ifactor**.

```
> ifactor(90288);
```

$$(2)^4 (3)^3 (11) (19)$$

```
> ifactor(324/952);
```

$$\frac{(3)^4}{(2)(7)(17)}$$

### 6.2.3. 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. 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	rational	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, que se utiliza para transformar desarrollos en serie en funciones polinómicas o de otro tipo, también sirve para convertir funciones trigonométricas o hiperbólicas a formas diversas:

```
> 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);
```

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

#### 6.2.4. 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 le indica lo contrario, Maple realiza la suma de exponentes antes de la ordenación.

```
> p := y^3+y^2*x^2+x^3+x^5;
```

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

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

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

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

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

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

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

## 7. FUNCIONES DE ÁLGEBRA LINEAL

### 7.1. LIBRERÍA LINALG

Casi todas las funciones de Álgebra Lineal están en una librería que se llama ***linalg***. 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 (si se pone **:** se evitará ver una nutrida lista de funciones):

```
> with(linalg);
```

[BlockDiagonal, GramSchmidt, JordanBlock, LUdecomp, QRdecomp, Wronskian, addcol, addrow, adj, adjoint, angle, augment, backsub, band, basis, bezout, blockmatrix, charmat, charpoly, cholesky, col, coldim, colspace, colspan, companion, concat, cond, copyint, crossprod, curl, definite, delcols, delrows, det, diag, diverge, dotprod, eigenvals, eigenvalues, eigenvectors, eigenvecs, entermatrix, equal, exponential, extend, ffgausselim, fibonacci, forwardsub, frobenius, gausselim, gaussjord, geneqns, genmatrix, grad, hadamard, hermite, hessian, hilbert, htranspose, ihermite, indexfunc, innerprod, intbasis, inverse, ismith, issimilar, iszero, jacobian, jordan, kernel, laplacian, leastsqrs, linsolve, matadd, matrix, minor, minpoly, mulcol, mulrow, multiply, norm, normalize, nullspace, orthog, permanent, pivot, potential, randmatrix, randvector, rank, ratform, row, rowdim, rowspace, rowspan, rref, scalarmul, singularvals, smith, stackmatrix, submatrix, subvector, sumbasis, swapcol, swaprow, sylveste, toeplitz, trace, transpose, vandermonde, vecpotent, vectdim, vector, wronskian]

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:

```
> ?linalg;
```

se obtiene información general sobre ***linalg***. Para obtener información sobre una función particular:

```
> ?function_name;
```

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:

```
> linalg[funcion](argumentos);
```

Por ejemplo, para calcular el determinante de una matriz **A**, basta teclear:

```
> linalg[det](A);
```

### 7.2. VECTORES Y MATRICES

Es importante tener en cuenta que todos los argumentos (de datos, numéricos y/o simbólicos) de las funciones de ***linalg*** deben ser ***matrices*** y/o ***vectores***. Se pueden también utilizar ***arrays*** definidos de la forma habitual, por medio de ***listas***. Téngase en cuenta que los ***arrays*** son una estructura de datos algo más general que los ***vectores*** y ***matrices***, pues pueden tener subíndices negativos. Para que un ***array*** sea equivalente a una matriz, los índices tienen que empezar en uno. Véase un ejemplo de ***array***, en el que se ve que los elementos de la matriz se forman a partir de ***listas*** de números:

```
> A := array([[1,2,3],[4,5,6],[7,8,9]]);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Es posible también definir una *matriz* o *array* sin asignar ningún valor a sus elementos, o asignando sólo algunos elementos (esto se hace incluyendo los índices entre paréntesis, como se muestra a continuación en la matriz **C**):

```
> B:=array(1..2, 1..3); C:=array(1..2,1..2,[(1,1)=s,(2,1)=3]);
```

```
B:=array(1..2, 1..3, [ ])
```

$$C := \begin{bmatrix} s & C_{1,2} \\ 3 & C_{2,2} \end{bmatrix}$$

Se puede acceder a los elementos de un *array* con sus índices de fila y columna separados por una coma, y encerrados entre corchetes. Por ejemplo:

```
> B[1,2]:=1; B[2,1]:=0;
```

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

$$B_{2,1} := 0$$

Ahora se puede tratar de ver los elementos de la matriz **B**. Obsérvese la salida de los siguientes comandos:

```
> B; print(B); evalm(B);
```

$B$

$$\begin{bmatrix} B_{1,1} & 1 & B_{1,3} \\ 0 & B_{2,2} & B_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} B_{1,1} & 1 & B_{1,3} \\ 0 & B_{2,2} & B_{2,3} \end{bmatrix}$$

Se puede ver que el nombre de las matrices (y de los vectores) no siguen las reglas normales en Maple. El nombre de una matriz se evalúa a sí mismo, y no a sus elementos. Para ver los elementos de la matriz se puede recurrir a las funciones **print** y **evalm**. Aunque en este caso el resultado haya sido el mismo, dichas funciones son diferentes. Como se verá más adelante, la función **evalm** tiene una gran importancia en cálculo matricial.

Las reglas para definir *vectores* en Maple son similares a las de las matrices, pero teniendo en cuenta que hay un solo subíndice. El siguiente ejemplo forma un vector a partir de una *lista* de tres valores:

```
> u:= array([1,2,3]);
```

$$u := [1 \quad 2 \quad 3]$$

```
> v:= array(1..5);
```

```
v:= array(1 .. 5, [ ])
```

En Maple los **vectores fila** son *arrays* de 1 fila y  $n$  columnas, mientras que los **vectores columna** son *arrays* de  $n$  filas y 1 columna. Esto debe ser tenido en cuenta a la hora de ciertas operaciones vectoriales y matriciales.

Los *arrays* de una o dos dimensiones constituyen una forma general de definir *vectores* y *matrices*. Sin embargo, en la librería **linalg** existen unas funciones llamadas **vector** y **matrix** que son algo más sencillas de utilizar con dicha finalidad, como se ve por ejemplo en los casos siguientes:

```
> a:=vector([5,4,6,3]);
```

```
a:= [5 4 6 3]
```

```
> V:=matrix(2, 3, [1, 2, 3, 4, 5, 6]); # una unica lista de valores
```

```
V:=  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ 
```

Los vectores definidos con la función **vector** son siempre **vectores columna**, y lo mismo sucede con los *arrays* de una sola dimensión. Obsérvese la salida de los siguientes comandos:

```
> vcol:=array(1..3,[1,2,3]); type(vcol, 'vector'); type(vcol, 'matrix');
```

```
vcol:= [1 2 3]
```

```
true
```

```
false
```

Los vectores definidos con la función **matrix** pueden ser **vectores fila** (si son  $1 \times n$ ) y **vectores columna** (si son  $n \times 1$ ). Obsérvese los siguientes ejemplos, de los que se saca la conclusión de que sólo los **vectores columna** son verdaderos vectores. Los **vectores fila** son matrices particulares.

```
> vfil:=matrix(1,3,[1, 2, 3]); type(vfil, 'matrix'), type(vfil, 'vector');
```

```
vfil:= [1 2 3]
```

```
true, false
```

```
> vcol:=matrix(3,1,[1, 2, 3]); type(vcol, 'matrix'), type(vcol, 'vector');
```

```
vcol:=  $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ 
```

```
true, false
```

```
> vcol2:=vector([1, 2, 3]); type(vcol2, 'matrix'), type(vcol2, 'vector');
```

```
vcol2:= [1 2 3]
```

```
false, true
```

Los elementos de los vectores y matrices se pueden definir también por medio de una *función* de los índices de fila y/o columna. Véase el siguiente ejemplo:

```
> f := i -> i^2; v := vector(4,f);
```

$$f := i \rightarrow i^2$$

$$v := [1 \quad 4 \quad 9 \quad 16]$$

Así pues, la función **vector(n,f)** produce un vector cuyos **n** elementos son el resultado de aplicar la función **f** al índice del vector. Dicho de otra forma, la función **vector(n,f)** define el vector  $[f(1), f(2), \dots, f(n)]$ .

Los elementos de las matrices pueden definirse por medio de una *función* en la llamada a **matrix**:

```
> f := (i,j) -> x^(i+j-1); A := matrix(2,2,f);
```

$$f := (i,j) \rightarrow x^{(i+j-1)}$$

$$A := \begin{bmatrix} x & x^2 \\ x^2 & x^3 \end{bmatrix}$$

Otra forma de definir matrices es por medio de la función **entermatrix**, que pregunta al usuario por el valor de los elementos de una matriz cuyo tamaño ha sido previamente definido:

```
> B:=array(1..2,1..2): # definición de una matriz (2x2) vacía
> entermatrix(B);
Note: each matrix entry entered MUST BE FOLLOWED BY A SEMICOLON
enter element 1,1 > 10;
enter element 1,2 > 20;
enter element 2,1 > 30;
enter element 2,2 > 40;
```

La función **entermatrix** se puede utilizar también para modificar el valor de algunos elementos de una matriz, o para especificar ciertas propiedades de la matriz (*simétrica*, *antisimétrica*, etc.). Para obtener más información sobre esta función teclee **?entermatrix**.

Finalmente, puede también introducir una matriz utilizando la notación *Standard Math* de Maple. Para ello tiene que abrir la paleta **Matrix Palette** con **View/Palettes**. Únicamente tiene que clicar sobre el icono con el tamaño de matriz que quiere crear y a continuación ir introduciendo los valores.

A continuación se muestran las principales operaciones y funciones que puede realizar sobre matrices. Algunas de ellas aparecen en el menú contextual que aparece al clicar con el botón derecho sobre el *output* o salida de matrices.

### 7.3. 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(2,2,[x,y,z,t]): B:=matrix(2,2,[1,2,3,4]):
> A+B, A*B, A&*B; evalm(A+B), evalm(A&*B);
```

$$A+B, A \cdot B, A \&*B$$

$$\begin{bmatrix} x+1 & y+2 \\ z+3 & t+4 \end{bmatrix}, \begin{bmatrix} x+3y & 2x+4y \\ z+3t & 2z+4t \end{bmatrix}$$

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 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(2*A),evalm(A+2);
```

$$\begin{bmatrix} 2x & 2y \\ 2z & 2t \end{bmatrix}, \begin{bmatrix} x+2 & y \\ z & t+2 \end{bmatrix}$$

#### 7.4. 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(2,2,[x,y,z,t]); A2:=evalm(A^2); AINV:=evalm(A^(-1));
```

$$A := \begin{bmatrix} x & y \\ z & t \end{bmatrix}$$

$$A2 := \begin{bmatrix} x^2 + yz & xy + yt \\ zx + tz & yz + t^2 \end{bmatrix}$$

$$AINV := \begin{bmatrix} -\frac{t}{-xt + yz} & \frac{y}{-xt + yz} \\ \frac{z}{-xt + yz} & -\frac{x}{-xt + yz} \end{bmatrix}$$

#### 7.5. 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:

```
> A:=matrix(2,2,[x,y,z,t]); B:=A; print(B);
```

$$B := A$$

$$\begin{bmatrix} x & y \\ z & t \end{bmatrix}$$

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[1,1]:=0; evalm(B), evalm(A);
```

$$B_{1,1} := 0$$

$$\begin{bmatrix} 0 & y \\ z & t \end{bmatrix}, \begin{bmatrix} 0 & y \\ z & t \end{bmatrix}$$

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.

## 7.6. FUNCIONES BÁSICAS DE ÁLGEBRA LINEAL.

A continuación se describen algunas de las funciones más importantes de la librería **linalg**. Esta librería dispone de un gran número de funciones para operar con matrices, algunas de las cuáles 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.

### 7.6.1. Función matadd

Esta función permite **sumar matrices** de las mismas dimensiones (no se pueden sumar con el operador +). Las matrices pueden también estar multiplicadas por sendos valores escalares, que se dan como parámetros adicionales de la función. Defina las matrices **A** y **B** (si no lo están) e introduzca el siguiente ejemplo:

```
> A:= array([[1,2,3],[2,3,4],[3,4,5]]):
> B:= array([[1,0,0],[0,1,0],[0,0,1]]):
> matadd(A, B);
```

$$\begin{bmatrix} 2 & 2 & 3 \\ 2 & 4 & 4 \\ 3 & 4 & 6 \end{bmatrix}$$

Para calcular la suma de **3\*A** y **4\*B**, hay que utilizar el comando **matadd** en la forma siguiente:

```
> matadd(A, B, 3, 4):
```

Como último ejemplo de esta función, teclee las líneas siguientes:

```
> matadd(A, B, 1, 10):
```

$$\begin{bmatrix} 11 & 2 & 3 \\ 2 & 13 & 4 \\ 3 & 4 & 15 \end{bmatrix}$$



### 7.6.2. Función charmat

Esta función permite construir la **matriz característica** de la matriz **A** (es decir, **lambda\*I-A**, siendo **I** la matriz identidad). Por ejemplo, teclee los comandos siguientes y observe la respuesta de Maple:

```
> charmat(A, lambda); charmat(A, x);
```

$$\begin{bmatrix} \lambda - 1 & -2 & -3 \\ -2 & \lambda - 3 & -4 \\ -3 & -4 & \lambda - 5 \end{bmatrix}$$

$$\begin{bmatrix} x - 1 & -2 & -3 \\ -2 & x - 3 & -4 \\ -3 & -4 & x - 5 \end{bmatrix}$$

### 7.6.3. Función charpoly

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**). Considérese el siguiente ejemplo:

```
> X := array([[1,2,3],[1,2,3],[1,5,6]]);
```

$$X := \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 5 & 6 \end{bmatrix}$$

```
> eqn:=charpoly(X, y); # se utiliza y como variable
```

$$eqn := y^3 - 9y^2$$

Ahora se podría utilizar la función **solve** para calcular los valores propios de la matriz.

### 7.6.4. Funciones colspace y rowspace

Estas funciones calculan, respectivamente, una base del subespacio de columnas y de filas de la matriz, que es pasada como argumento. Los elementos de la matriz tienen que estar definidos numéricamente. Véase un ejemplo y la respuesta que da Maple:

```
> C := array([[1,2,3],[2,3,4],[3,4,5]]);
```

$$C := \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

```
> colspace(C), rowspace(C);
```

$$\{[0 \ 1 \ 2], [1 \ 0 \ -1]\}, \{[0 \ 1 \ 2], [1 \ 0 \ -1]\}$$

### 7.6.5. Función crossprod

Esta función calcula el **producto vectorial** de dos vectores:

```
> crossprod(u,v):
```

### 7.6.6. Función det

Esta función calcula el **determinante** de una matriz definida de forma numérica o simbólica. Se utiliza en la forma siguiente:

```
> det(A):
```

### 7.6.7. Función dotprod

Esta función calcula el *producto escalar* de dos vectores, que pueden ser reales o complejos. Se utiliza en la forma siguiente:

```
> dotprod(u,v):
```

### 7.6.8. Función eigenvals

Esta función calcula los valores propios de una matriz cuadrada, calculando las raíces del polinomio característico  $\det(\mathbf{A}-\lambda\mathbf{I})=0$ . Por ejemplo, con la matriz  $\mathbf{A}$  definida anteriormente:

```
> eigenvals(A);
```

$$0, \frac{9}{2} + \frac{1}{2}\sqrt{105}, \frac{9}{2} - \frac{1}{2}\sqrt{105}$$

Si la dimensión de la matriz es mayor que 4, Maple puede no ser capaz de calcular los valores propios simbólicamente. Sin embargo, siempre se pueden intentar calcular de modo numérico mediante la función *evalf(eigenvals(A))*, que utiliza un algoritmo numérico más general.

### 7.6.9. Función eigenvects

Esta función calcula los vectores propios resolviendo el sistema de ecuaciones lineales:

$$(\mathbf{A}-\lambda\mathbf{I})\mathbf{c} = 0$$

en el que Maple calcula el vector  $\mathbf{c}$  para cada valor propio de  $\mathbf{A}$ . 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. (Para más información teclear *?eigenvects*). Considérese a continuación el siguiente ejemplo:

```
> mat:= array([[1,-3,3], [3,-5,3], [6,-6,4]]):
> eigenvects(mat);
```

```
[-2, 2, {[1 1 0], [-1 0 1]}], [4, 1, {[1 1 2]}]
```

### 7.6.10. Función gausselim

Esta función realiza la *triangularización* de una matriz  $m$  por  $n$  con pivotamiento por filas. El resultado es una matriz triangular superior:

```
> A1:= array([[x,1,0],[0,0,1],[1,y,1]]);
```

$$A1 := \begin{bmatrix} x & 1 & 0 \\ 0 & 0 & 1 \\ 1 & y & 1 \end{bmatrix}$$

```
> gausselim(A1);
```

$$\begin{bmatrix} x & 1 & 0 \\ 0 & \frac{xy-1}{x} & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

### 7.6.11. Función inverse

Esta función invierte una matriz cuadrada. Si la inversa no existe, se imprime un mensaje de error. Su forma es la siguiente:

```
> inverse(A);
```

### 7.6.12. Función iszero

La función *iszero(A)* comprueba que **todos los elementos** de **A** son cero. Si es así, el resultado es *true* y en caso contrario *false*.

### 7.6.13. Función linsolve

Esta función sirve para **resolver sistemas de ecuaciones lineales**, con una matriz que puede ser rectangular. Se llama de la forma siguiente:

```
> linsolve(A, b);
```

donde **A** es una matriz  $m \times n$  y **b** un vector de  $m$  elementos. Si no existe solución, Maple devuelve un NULL. Si hay muchas soluciones, se imprimen de forma paramétrica utilizando los parámetros  $t1$ ,  $t2$ ,  $t3$ , etc.

Si **b** es también una matriz, se supone que se trata de un sistema de ecuaciones con varios segundos miembros.

### 7.6.14. Función multiply

Sirve para **multiplicar** varias matrices y vectores (en el orden en que aparecen como argumentos). Las dimensiones tienen que ser coherentes. Veamos un ejemplo:

```
> A2:= array([[1,2],[3,4]]):
> B2:= array([[0,1],[1,0]]):
> C2:= array([[1,2],[4,5]]):
> multiply(A2,B2,C2);
```

### 7.6.15. Función randmatrix

Esta función genera una matriz de números aleatorios entre -99 y +99. Por ejemplo, la sentencia:

```
> randmatrix(4,5);
```

genera una matriz de tamaño (4x5).

### 7.6.16. Función rank

Esta función devuelve el rango de una matriz definida de forma numérica.

```
> rank(A);
```

3

### 7.6.17. Función trace

Esta función calcula la traza de una matriz cuadrada definida de forma numérica o simbólica, esto es, la suma de los elementos de la diagonal..

```
> trace(A);
```

## 8. GRÁFICOS EN 2 Y 3 DIMENSIONES

La visualización de resultados es una de las capacidades más utilizadas del álgebra computacional. Poder ver en gráficos 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.



Figura 8. Botones adicionales para opciones gráficas 2-D

Estos botones permiten modificar las características del dibujo. Por ejemplo, puede hacerse que la función aparezca representada con trazo continuo o por medio puntos, se pueden dibujar ejes de distinto tipo, y se puede obligar a que la

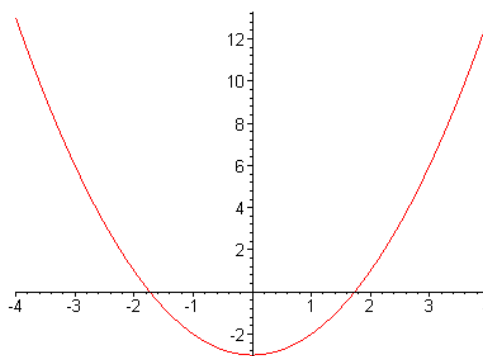
escala sea la misma en ambos ejes. Asimismo, Maple devuelve la posición  $(x,y)$  de cualquier punto clicando sobre la gráfica. Estas opciones se detallan en el **apartado 8.1.5**.

### 8.1. GRÁFICOS BIDIMENSIONALES

#### 8.1.1. Expresiones de una variable

Las *expresiones de una sola variable* se pueden representar mediante el comando **plot**. Para poder visualizar una función en la forma más sencilla, es suficiente con informar a **plot** de la expresión de la función y del rango de la variable independiente que se quiere representar. He aquí un primer ejemplo, cuyo resultado se puede ver en la figura 6:

```
> plot(x^2-3, x=-4..4);
```



El ejemplo anterior puede resolverse también de otra forma, definiendo una función y utilizándola en **plot** en vez de la expresión anterior. Esto se haría en la forma (obsérvese que en este caso no hace falta poner la variable **x**, ni en la función ni en la definición del rango):

```
> f:=(x)->x^2-3; plot(f, -4..4);
```

Un segundo ejemplo (ya no se incluye la figura) puede ser el siguiente:

```
> plot(x*sin(x), x=-3*Pi..3*Pi);
```

Existe la posibilidad de *imprimir varias funciones* con un solo comando, en una misma gráfica y con unos mismos ejes, de las dos formas que se indican a continuación (atención a las llaves { }, que indican conjuntos o *sets* y que contienen todas las expresiones que se van a dibujar):

```
> plot({x^2, exp(x), x}, x=0..3, y=0..10); # tres funciones
> plot({seq(cos(x*i), i=-1..4)}, x=-Pi..Pi); # cuatro funciones
```

Otra función para representar gráficos bidimensionales es **smartplot** (sólo disponible en Windows 95, Windows NT o superior). De hecho, cuando clicamos sobre **Plots** en el menú contextual Maple escribe y ejecuta automáticamente **smartplot**.

```
> smartplot(cos(x) + sin(x));
```

### 8.1.2. Funciones paramétricas

Se pueden representar también *funciones paramétricas* (dos ecuaciones, función de un mismo parámetro) definiéndolas en forma de *lista* (atención a los corchetes [ ], que engloban tanto a las expresiones como al parámetro y su rango de valores):

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

Si ha ejecutado este último ejemplo habrá comprobado que la circunferencia obtenida se asemeja a una elipse. Para no ver el dibujo distorsionado tiene que añadir la opción **scaling=constrained** (véase el apartado 8.1.5) o bien clicar sobre el icono correspondiente.



### 8.1.3. Dibujo de líneas poligonales

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]]);
```

### 8.1.4. Otras funciones de la librería plots

La librería **plots** contiene funciones para realizar otros gráficos que no son tan habituales como los anteriores. Aunque no sea necesario probar todas, sí se recomienda intentar utilizar algunas capacidades más de las que aquí se han explicado. Para ello se puede recurrir al **help** tecleando *?plots[nombre de la función que interese]*. La lista completa de funciones es la siguiente (se incluyen tanto gráficos planos como tridimensionales):

animate	animate3d	animatecurve	changecoords	complexplot
complexplot3d	conformal	contourplot	contourplot3d	coordplot
coordplot3d	cylinderplot	densityplot	display	display3d
fieldplot	fieldplot3d	gradplot	gradplot3d	implicitplot
implicitplot3d	inequal	listcontplot	listcontplot3d	listdensityplot
listplot	listplot3d	loglogplot	logplot	matrixplot
odeplot	pareto	pointplot	pointplot3d	polarplot
polygonplot	polygonplot3d	polyhedra_supported	polyhedraplot	replot
rootlocus	semilogplot	setoptions	setoptions3d	spacecurve
sparsematrixplot	sphereplot	surfdata	textplot	textplot3d
tubeplot				

Se pueden cargar todas estas funciones de una vez con el comando:

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

o bien se puede utilizar (es decir, ejecutar) cada una de estas funciones por separado (sin cargarlas todas) en la forma:

```
> plots[nombre_funcion](argumentos);
```

A continuación se incluyen algunos ejemplos de los usos y utilidades de algunas de las funciones de la lista anterior. Recuerde que las *coordenadas polares* son aquellas que definen un punto mediante su distancia  $r$  al origen de los ejes de coordenadas, y el ángulo  $\theta$  entre la línea que une el punto con dicho origen y la horizontal. En el siguiente ejemplo (en coordenadas polares) se representa, para cada ángulo, un radio de longitud doble al valor del ángulo en radianes:

```
> plots[polarplot](2*t);
```

donde se supone que  $t$  es el ángulo, que por defecto varía entre  $-\pi$  y  $\pi$ . Se pueden dibujar también *curvas polares paramétricas*, con el radio, el ángulo y el parámetro encerrados entre corchetes, en forma de *lista*, como en el siguiente ejemplo:

```
> polarplot([cos(t), sin(t), t=0..4*Pi], color=blue);
```

Para imprimir un *campo vectorial* hay que hacer lo siguiente:

```
> plots[fieldplot]([sin(y), cos(x)], x=-10..10, y=-10..10, arrows=SLIM);
```

donde se han representado del orden de 400 vectores. La primera componente de cada uno de ellos es el seno de su coordenada  $y$  (en radianes) y su componente en  $y$  es el coseno de su coordenada  $x$ .

También se pueden representar *funciones implícitas*, esto es, expresiones en las que ninguna de sus variables está despejada. Obsérvese como se dibuja una elipse:

```
> implicitplot(x^2/25+y^2/9=1, x=-6..6, y=-6..6, scaling=CONSTRAINED);
```

Maple ofrece la posibilidad de representar sistemas de inecuaciones en 2 variables mediante la función *inequal*. La gráfica obtenida se compone de 4 partes:

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.

Ejecute el siguiente ejemplo:

```
> inequal( { x+y>0, x-y<=1, y=2 }, x=-3..3, y=-3..3,
optionsfeasible=(color=red), optionsopen=(color=blue, thickness=2), optionsclosed=(color=green, thickness=3), optionsexcluded=(color=yellow) );
```

### 8.1.5. Colores y otras opciones de plot

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. Seguidamente se proponen dos ejemplos para practicar con estas opciones:

```
> plot([sin(2*x), cos(x), x=0..2*Pi], color=Blue, title='Azul');
> plot(x^3+2*x^2-3*x-1, x=-3..3, axes =FRAME, style=POINT);
```

## 8.2. GRÁFICOS TRIDIMENSIONALES

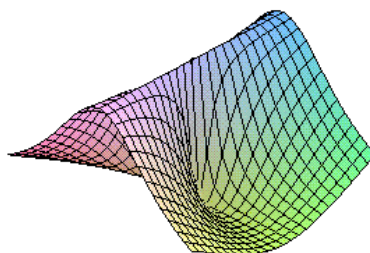
### 8.2.1. Expresiones de dos variables

Los gráficos tridimensionales no presentan grandes diferencias respecto a los bidimensionales. Al existir, en este caso; dos variables independientes, hay que indicar a la función gráfica **plot3d** el rango de ambas variables:

```
> plot3d((x^2-y^2)/(x^2+y^2), x=-2..2, y=-3..3);
> plot3d(exp(x+y), x=-0..2, y=-2..2);
```

También se pueden dibujar con Maple funciones propiamente dichas (en lugar de expresiones), y en ese caso no es necesario poner las variables en la función y en los rangos). Por ejemplo, la primera de las funciones anteriores se puede dibujar también en la forma:

```
> f3 := (x,y)->(x^2-y^2)/(x^2+y^2); plot3d(f3, -2..2, -3..3);
```



Si clicamos sobre la gráfica anterior aparecerán unos botones en la barra de herramientas:



Figura10: Botones adicionales para opciones gráficas tridimensionales

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 menú 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.

```
> smartplot3d(sin(x^2 + y^2));
```

### 8.2.2. Otros tipos de gráficos 3-D

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);
```

También se puede representar *varias funciones* en la misma gráfica y con los mismos ejes, definiéndolas como *set* entre llaves { }:

```
> plot3d({x+y^2, -x-y^2}, x=0..3, y=0..3);
```

Algunas de las funciones de la librería *plots* citadas en el apartado anterior están especialmente indicadas para las funciones tridimensionales. Así, para utilizar *coordenadas esféricas* hay que hacer lo siguiente (en las coordenadas esféricas se utilizan dos ángulos y la distancia al origen de coordenadas, para definir la posición de un punto en el espacio):

```
> sphereplot((1.3)^z*sin(theta), z=-1..2*Pi, theta=0..Pi);
```

Para dibujar una *curva* en el espacio (una hélice de radio creciente):

```
> spacecurve([t*cos(t), t*sin(t), t], t=0..7*Pi);
```

y para dibujar una *función implícita* (un elipsoide):

```
> implicitplot3d(x^2+2*y^2+3*z^2=1, x=-1..1, y=-1..1, z=-1..1);
```

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]);
```

### 8.3. ANIMACIONES

Maple realiza *animaciones* con gran facilidad. En las animaciones se representa una función que varía en el tiempo o con algún parámetro. Este parámetro es una nueva variable que hace falta introducir. Las animaciones bidimensionales tienen una variable espacial y otra variable temporal, y ambas son independientes. Para obtener una animación hay que definir los rangos de esas dos variables. Para realizar animaciones en 2D se dispone de la función **animate**, en la librería **plots**.

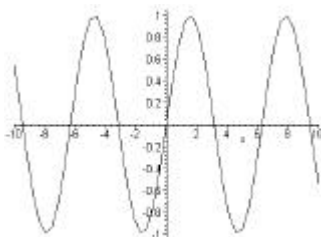
Las animaciones de Maple quedan insertadas, al igual que las gráficas, en la hoja de Maple. Si clicamos sobre ella, queda seleccionada y aparecen unos botones en la barra de herramientas, junto con unos menús adicionales en la barra de menús.



Figura 13: Botones adicionales para el control de animaciones tridimensionales

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.

```
> animate(sin(x*t), x=-10..10, t=1..2, frames=50);
```



Para que la animación comience hay que pulsar el botón **Start**. Considérese otro ejemplo:

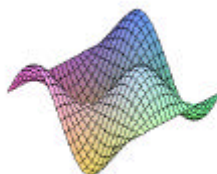
```
> animate([sin(x*t), x, x=-4..4], t=1..4, coords=polar, numpoints=100,
frames=100);
```

Con estas dos representaciones se ha podido ver cómo varía la función *seno* en coordenadas cartesianas y polares. Por defecto el número de imágenes (*frames*) que se genera es 16.

Las animaciones tridimensionales son aún más espectaculares. Se realizan con la función **animate3d**. Ahora, las variables espaciales independientes son dos y por eso hay que indicar el rango

de tres variables independientes, las dos espaciales y la temporal. Aquí se presentan algunos ejemplos para poder practicar (por defecto, el número de frames es 8):

```
> animate3d(cos(t*x)*sin(t*y), x=-Pi..Pi, y=-Pi..Pi, t=1..2);
```



```
> animate3d((1.3)^x*sin(u*y), x=-1..2*Pi, y=0..Pi, u=1..8,
coords=spherical);
```

```
> animate3d(sin(x)*cos(t*u), x=1..3, t=1..4, u=1/4..7/2,
coords=cylindrical);
```

Recuerde que en coordenadas cilíndricas se utilizan, para definir la posición de un punto, la coordenada cartesiana  $z$ , la distancia  $r$  al eje  $z$  y el ángulo  $\theta$  respecto al eje  $x$ .

## 9. FUNCIONES DEFINIDAS MEDIANTE EL OPERADOR FLECHA (->)

### 9.1. FUNCIONES DE UNA VARIABLE

Con Maple, las relaciones funcionales se pueden definir de dos modos:

- mediante una *expresión* o fórmula.
- mediante una *función* matemática propiamente dicha.

A continuación se verá, con un ejemplo definido de ambas formas, la diferencia entre una expresión y una función. En primer lugar, se va a definir la temperatura  $T$  como una expresión en la que interviene la variable tiempo  $t$ :

```
> T := T0*exp(-c*t);
```

$$T := T0 e^{(-c t)}$$

En la expresión anterior, a la variable  $t$  se le puede dar otro valor, por ejemplo asignando a  $t$  otra expresión en dicha ecuación. Para ello hay que utilizar el comando *subs*, o bien cambiar el valor de las variables que intervienen en esa expresión. Obsérvense los resultados de los siguientes comandos:

```
> subs(t=1/c, T);
```

$$T0 e^{(-1)}$$

```
> t:=1/c; T;
```

$$t := \frac{1}{c}$$

$$T0 e^{(-1)}$$

```
> t:='t'; c:=2/t; T;
```

$$t := t$$

$$c := 2 \frac{1}{t}$$

$$T0 e^{(-2)}$$

Aunque el método anterior ha funcionado de modo correcto,  $T$  no es una verdadera **función del tiempo** ( $t$  interviene en la misma forma que  $T0$  o  $c$ ). Ahora se va a definir  $T$  como verdadera función del tiempo  $t$ . Para ello se va a utilizar el **operador flecha** ( $\rightarrow$ ). Observe bien la sintaxis de este ejemplo (los espacios en blanco son opcionales):

```
> T := t -> T0*exp(-c*t);
```

$$T := t \rightarrow T0 e^{(-c t)}$$

Ahora es mucho más fácil obtener el valor de  $T$  para cualquier valor de la variable  $t$  (que ya interviene en la función de un modo diferente que  $c$  o  $T0$ ). Por ejemplo:

```
> T(1/c), T(0);
```

$$T0 e^{(-1)}, T0$$

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:

```
> T, T(t);
```

$$T, T0 e^{(-c t)}$$

La representación gráfica de las funciones definidas de esta forma es también más sencilla (ver apartado de Gráficos). Otra ventaja importante de las funciones definidas mediante el operador flecha es que se les puede aplicar la función **solve**:

```
> c:='c'; solve(T(t)=100, t);
```

$$-\frac{\ln\left(100 \frac{1}{T0}\right)}{c}$$

## 9.2. FUNCIONES DE DOS VARIABLES

Las funciones de dos o más variables se pueden definir utilizando de nuevo el **operador flecha**  $\rightarrow$ . Véase un ejemplo, con gráfico incluido (se omite la figura):

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

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

```
> f(3,2); # evaluación de la función para valores de las variables
```

$$-9$$

```
> plot3d(f, -1..1, -1..1, numpoints= 500, style=PATCH, axes=FRAME);
```

Se va a definir ahora una función llamada **multipl** que acepta una entrada (primer argumento), y la repite **n** veces (segundo argumento):

```
> multipl := (x,n) -> seq(x, j=1..n); # repetir x n veces
      multipl := (x,n) -> local j; seq(x,j = 1 .. n)
> multipl(estudia,9); # repetir "estudia" 9 veces
      estudia, estudia, estudia, estudia, estudia, estudia, estudia, estudia, estudia
> multipl(no,9);
      no, no, no, no, no, no, no, no, no
```

En la función anterior, la variable **j** ha sido implícitamente definida como **local** (existe de modo independiente en la función) y por eso está *desasignada*. Esto se puede comprobar ejecutando los siguientes comandos:

```
> j; n;
> n := 4; multipl(x,n);
```

### 9.3. CONVERSIÓN DE EXPRESIONES EN FUNCIONES

Ya se ha 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 ventajas en muchos casos. En Maple existe el operador **unapply**, que puede convertir una expresión o fórmula en una función. Considérese cómo se comporta Maple con un ejemplo concreto:

```
> formula := (b^2*x^2*sin(b*x)-2*sin(b*x)+2*b*x*cos(b*x)*a*t)/b^3;
```

$$formula := \frac{b^2 x^2 \sin(b x) - 2 \sin(b x) + 2 b x \cos(b x) a t}{b^3}$$

```
> F := unapply(formula, x, t);
```

$$F := (x, t) \rightarrow \frac{b^2 x^2 \sin(b x) - 2 \sin(b x) + 2 b x \cos(b x) a t}{b^3}$$

```
> F(0, 1), F(Pi/b, 5);
```

$$0, -10 \frac{\pi a}{b^3}$$

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) -> formula;
```

$$G := (x, t) \rightarrow formula$$

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

$$\frac{b^2 u^2 \sin(b u) - 2 \sin(b u) + 2 b u \cos(b u) a v}{b^3}, \frac{b^2 x^2 \sin(b x) - 2 \sin(b x) + 2 b x \cos(b x) a t}{b^3}$$

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=formula, (x, t) -> body);
```

$$H := (x, t) \rightarrow \frac{b^2 x^2 \sin(b x) - 2 \sin(b x) + 2 b x \cos(b x) a t}{b^3}$$

```
> H(u, v); # ahora sí funciona
```

$$\frac{b^2 u^2 \sin(b u) - 2 \sin(b u) + 2 b u \cos(b u) a v}{b^3}$$

#### 9.4. 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;
```

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

$$g := y \rightarrow e^y - 1$$

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

$$h := f + g$$

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

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

$$h := f g$$

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

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);
```

$$h := f@g$$

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

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));
```

## 9.5. FUNCIONES ANÓNIMAS

Las funciones de Maple pueden no tener nombre, es decir, ser anónimas. Estas funciones son útiles cuando hay algo que se quiere ejecutar una sola vez y no se quiere desperdiciar un nombre para ello. Considérese el siguiente ejemplo:

```
> map( x -> x^2, a+b+c);
```

$$a^2 + b^2 + c^2$$

Ahora se calcula el logaritmo del 2º elemento de cada entrada:

```
> data := [[1,1.0],[2, 3.8],[3,5.1]];
data := [[1, 1.0], [2, 3.8], [3, 5.1]]
```

```
> map( x -> subsop(2=ln(op(2,x)), x), data);
[[1, 0], [2, 1.335001067], [3, 1.629240540]]
> sum('x^i', 'i'=0..6);
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6$$

```
> select(t -> degree(t)<3, %);
```

$$1 + x + x^2$$

## 10. ECUACIONES DIFERENCIALES

### 10.1. INTEGRACIÓN DE ECUACIONES DIFERENCIALES ORDINARIAS

Maple permite resolver ecuaciones diferenciales ordinarias con el comando *dsolve*. La forma de utilizar este comando es la siguiente:

```
> dsolve(ecuacion, variable, opciones);
```

Una de las claves de la utilización de este comando es la forma en la que se escribe la ecuación diferencial. Considérese la siguiente ecuación diferencial:

$$y'(x) = a * y(x)$$

Con Maple, esta ecuación diferencial se escribe del siguiente modo:

```
> restart; ec := diff(y(x), x) = a*y(x);
```

$$ec := \frac{\partial}{\partial x} y(x) = a y(x)$$

Ahora se puede llamar a la función *dsolve*:

```
> dsolve(ec, y(x));
```

$$y(x) = e^{(a x)} \_C1$$

La constante de integración debe ser calculada en función de las condiciones iniciales. Éstas pueden ser incluidas en la función **dsolve** entre llaves { }, en la forma:

```
> dsolve({ec, y(0)=1}, y(x));
```

$$y(x) = e^{(a x)}$$

Con este comando también se pueden resolver ecuaciones diferenciales ordinarias de orden superior:

Considerese la ecuación de segundo orden  $\text{diff}(y(x), x, x) + 5*\text{diff}(y(x), x) + 6*y(x) = 0$ . En Maple:

```
> diff_eq1 := D(D(y))(x)+5*D(y)(x)+6*y(x) = 0;
```

$$\text{diff\_eq1} := (D^{(2)})(y)(x) + 5 D(y)(x) + 6 y(x) = 0$$

Obsérvese que se utiliza el operador *D*. Este operador halla únicamente derivadas de **funciones** (en este caso  $y(x)$ ) y su salida son siempre funciones. El comando *diff*, en cambio, opera con **funciones** y también con **expresiones**, y su salida son siempre **expresiones**.

El resto del proceso es análogo al ejemplo anterior. Se definen las condiciones de contorno, se resuelve la ecuación y se representa gráficamente (sólo se va a mostrar parte de la salida):

```
> init_con := y(0)=0, D(y)(0)=1;
```

$$\text{init\_con} := y(0) = 0, D(y)(0) = 1$$

```
> sol := dsolve( {diff_eq1, init_con} , {y(x)} );
```

$$\text{sol} := y(x) = -e^{(-3 x)} + e^{(-2 x)}$$

```
> expr := subs(sol, y(x));
```

```
> plot( expr, x=0..5, axes=BOXED );
```

Maple también puede resolver sistemas de ecuaciones diferenciales ordinarias:

```
> diff(y(x), x, x)=z(x), diff(z(x), x, x)=y(x);
```

```
> sys := (D@@2)(y)(x) = z(x), (D@@2)(z)(x) = y(x);
```

$$\text{sys} := (D^{(2)})(y)(x) = z(x), (D^{(2)})(z)(x) = y(x)$$

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*.

## 10.2. INTEGRACIÓN DE ECUACIONES DIFERENCIALES EN DERIVADAS PARCIALES

### 10.2.1. Integración de ecuaciones diferenciales en derivadas parciales homogéneas

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

```
> pde := D[1, 1, 2, 2, 2](U)(x, y) = 0; # recuerdese que la notación
D[1](U) se refiere a la derivada de U respecto de la primera variable y,
por tanto, D[1,1,2,2,2](U) deriva dos veces respecto de la primera variable
y tres veces respecto de la segunda
```

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

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

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

### 10.2.2. Integración de ecuaciones diferenciales en derivadas parciales no homogéneas

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

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

### 10.2.3. Representación de las soluciones

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

Sea la E.D.:

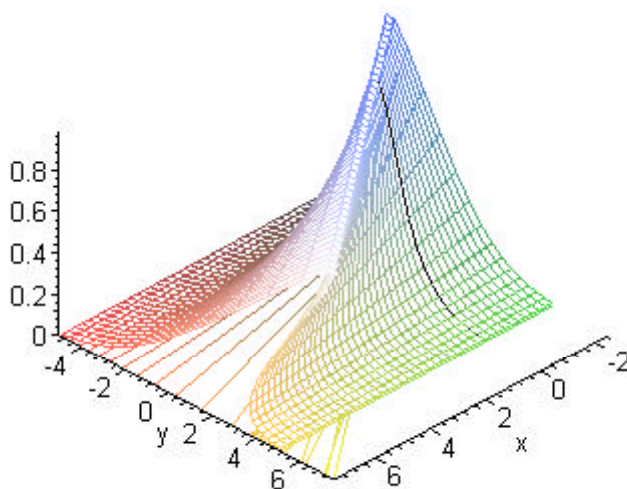
```
> pde := D[1](z)(x, y) + z(x, y)*D[2](z)(x, y) = 0;
```

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

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

```
> ini := [0, s, sech(s)], s=-5..5:
> PDEtools[PDEplot]( pde, z(x, y), ini, numsteps=[10, 30], numchar=30,
basechar=true, method=internal, title='A PDE Plot-Internal Method',
style=hidden );
```

A PDE Plot-Internal Method





## 11. PROGRAMACIÓN

### 11.1. ESTRUCTURAS DE PROGRAMACIÓN

Maple dispone de un lenguaje de programación propio con bifurcaciones y bucles similares a los de otros lenguajes. A continuación se revisará brevemente estas construcciones, que tienen una sintaxis algo diferente a la de C y MATLAB. A semejanza de este último programa, Maple dispone de *palabras clave* para indicar la terminación del bloque, en lugar de las llaves { } de C.

#### 11.1.1. Bifurcaciones: sentencia if

La sentencia condicional *if* tiene cuatro formas. Todas ellas comienzan con la palabra *if* y terminan con *fi* ; , donde *fi* marca el final del bloque y ; la terminación de línea.

La forma más sencilla es la siguiente:

```
if condicion then sentencias fi;
```

Ante esta selección Maple evalúa *condicion*: si el resultado es el valor booleano *true*, se ejecutan las *sentencias*. Si la *condicion* es *false* o **FAIL**, se saltan y se pasa directamente a la siguiente orden.

La segunda forma de *if* es:

```
if condicion then sentencias1 else sentencias2 fi;
```

La diferencia que tiene con la primera forma es que en caso de obtener una evaluación de la condición *false* o **FAIL**, en vez de saltar a la siguiente orden, como en el primer caso, Maple ejecuta las sentencias que siguen a *else*, es decir, *sentencias2*.

En todas las formas de la sentencia *if* la *condición* debe evaluar uno de los valores booleanos *true*, *false* o **FAIL**; en caso contrario el programa mandará un mensaje de error.

```
> x:=2;
> if x then 5 else 3 fi;
Error, invalid boolean expression
```

En el ejemplo *x* no es ninguna condición, no corresponde *true*, *false* o **FAIL**, es simplemente *x*, y por eso el programa no lo acepta.

La sentencia de la cláusula *then* o *else* puede ser cualquier tipo de sentencia, incluyendo otra *if*:

```
> if x>5 then 5
>   else if x=1 then 1
>   else 0 fi
> fi;
```

Obsérvese que *x>5* y *x=1* sí son condiciones, que pueden cumplirse (*true*) o no (*false*). Cuando sea *x>5* el programa retiene un 5; si es *x=1* un 1 y para cualquier otro valor de *x* un 0. En este ejemplo existen dos sentencias *if* con sus correspondientes *fi*, que cierran cada uno de los bloques.

Cuando existen muchos casos a considerar, el uso de sentencias *if* anidadas resulta incómodo. Por esto se recomienda utilizar una de las dos alternativas siguientes:

```
if condicion1 then sentencias1 elif condicion2 then sentencias2 fi;
if condicion1 then sentencias1 elif condicion2 then sentencias2 else
sentencias3 fi;
```

donde se pueden incluir tantas etapas *elif* (*else if*) como se desee. Sólo hay una sentencia *fi*, que lleva el carácter de terminación de línea (; o :), ya que se trata de un sólo bloque.

```
> if x>5 then 5
>   elif x=4 then 4
>   elif x=3 then 2
>   elif x=2 then 1
>   else ERROR("bad argument",x)
> fi;
```

### 11.1.2. Bucles: sentencia for

La forma general de la sentencia de repetición *for* es la siguiente:

```
for var from inicio by salto to final while expresion do sentencias od;
```

La variable *var* comienza tomando el valor *inicio*, y luego el bucle se repite incrementando *var* de *salto* en *salto* hasta que llega a *final*. Si se cumple *expresión*, en cada uno de esos escalones, se ejecutan las *sentencias*. El final del bucle se indica con la palabra *od*. Se utiliza un valor de *salto* negativo cuando se quiere que *var* tome valores hacia abajo, es decir, que realice una cuenta atrás. Excepto por la cláusula *for*, que debe aparecer siempre en primer lugar, se puede variar el orden. Además se puede omitir las cláusulas *for var*, *from inicio*, *by salto*, *to final* o *while expresion*. Si se omite una cláusula, esta toma un valor por defecto, que sería el que se indica en la siguiente tabla:

Claúsula	Valor por defecto
for	Variable auxiliar
from	1
by	1
to	Infinito
while	true

```
> for i from 1 to 4 do i+1 od;
```

2

3

4

5

En el ejemplo, cuando comienza el bucle, *i* vale 1. Como no se incluye la cláusula *while*, no se debe cumplir ninguna condición para continuar. Así, se ejecuta la orden *i+1* y el programa devuelve un 2. Tampoco se ha especificado el valor de *by*, así que al llegar a *od*, la variable *i* se incrementa en 1, indicado por defecto. Esto mismo se repite mientras *i* vale 2, 3 y 4, cifra que marca el final del bucle.

El siguiente ejemplo es muy parecido. Tan sólo cambia la variación *i*, que en este caso va desde 4 hasta 1, de 1 en 1. Por esto ahora debe aparecer *by* con el valor  $-1$ .

```
> for i from 4 to 1 by -1 do i+1 od;
5
4
3
2
```

### 11.1.3. Bucles: sentencia while

El bucle ***while*** es un bucle ***for*** en el que se han omitido todas las cláusulas excepto la *while*. Su forma es la siguiente:

```
while condicion do sentencias od;
```

Maple evalúa la *condicion*. Mientras sea ***true***, se ejecutan las *sentencias* y se repite hasta que la condición deje de cumplirse. La palabra clave *od* indica el fin del bucle, y es la única que lleva el carácter de terminación (; o :).

```
> x:=2;
> while x<15 do x:=x^2 od;

x := 4
x := 16
```

En el ejemplo se comienza dándole a *x* el valor 2. Mientras sea  $x < 15$ , en cada vuelta se actualiza su valor a  $x^2$ . El bucle se parará cuando  $x = 16$  ( $4^2$ ), ya que deja de cumplirse la condición.

### 11.1.4. Bucles: sentencia for-in

La particularidad de este bucle es que se aplica a cada uno de los componentes de un objeto, que puede ser una lista, un set, una suma de términos, un producto de factores o los caracteres de una cadena.

La expresión general del bucle ***for-in*** es :

```
for var in expr while cond do sentencias od;
```

Este bucle aplica las *sentencias* a cada uno de los elementos del objeto si se cumple la condición. En el ejemplo no aparece la cláusula *while* así que las *sentencias*, que en este caso es un *if*, se ejecutan siempre.

```
> L:=[8,5,1,3,9,6];

L := [8, 5, 1, 3, 9, 6]

> for i in L do
>     if i>4 then print(i) fi;
> od;

8
5
9
6
```

### 11.1.5. Sentencias break y next

La sentencia **break** hace que Maple salga inmediatamente del bucle en el que está.

```
> L:=[8,5,1,3,9,6];
                                     L:=[8, 5, 1, 3, 9, 6]

> for i in L do
>     print(i);
>     if i=3 then break fi;
> od;
                                     8
                                     5
                                     1
                                     3
```

Cuando  $i = 3$  Maple sale del bucle **for-in**.

La sentencia **next** hace que Maple salte a la siguiente iteración.

```
> L:=[8,5,1,3,9,6];
                                     L:=[8, 5, 1, 3, 9, 6]

> for i in L do
>     if i=3 then next fi;
>     print(i);
> od;
                                     8
                                     5
                                     1
                                     9
                                     6
```

Cuando  $i = 3$  Maple salta la sentencia `print (i)` y pasa directamente a  $i = 4$ .

Si **break** o **next** se utilizan fuera de una sentencia de repetición el programa devolverá un mensaje de error.

## 11.2. PROCEDIMIENTOS: DEFINICIÓN

Un **procedimiento** (*procedure*) es un grupo de comandos que Maple ejecuta conjuntamente. Es lo equivalente a las subrutinas de Fortran, o las funciones de C y MATLAB. Se define de la siguiente forma:

```
Proc (P)
Local L;
Global G;
Options O;
Description D;
B
end
```

B es la secuencia de sentencias que forma el cuerpo del procedimiento. Los parámetros P, las variables locales L, globales G, las opciones O y la descripción D son opcionales.

### 11.2.1. 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.

### 11.2.2. 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 null.

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*.

### 11.2.3. 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 := proc(expr:anything, x:name) ... end*

### 11.2.4. El campo de descripción

Es la última cláusula 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`.

```
description string;
```

### 11.3. 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.
- a través de un *return explícito*.
- mediante un *return de error*.

#### 11.3.1. 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

#### 11.3.2. 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.

#### 11.3.3. 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);
```

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

```
> 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.

#### 11.4. GUARDAR Y RECUPERAR PROCEDIMIENTOS

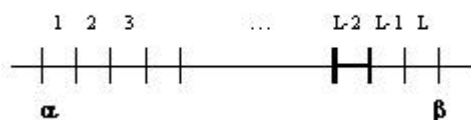
Mientras se está desarrollando un procedimiento se puede salvar el trabajo grabando la hoja de Maple entera. Una vez que se está satisfecho con cómo funciona el procedimiento se puede guardar en un archivo \*.m. Estos archivos forman parte del formato interno de Maple, lo hace que se pueda trabajar con 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";
```

#### 11.5. EJEMPLO DE PROGRAMACIÓN CON PROCEDIMIENTOS

A continuación se va a realizar un ejercicio que consiste en la obtención de unos coeficientes  $a_0, a_1, \dots, a_M$  que servirán para la determinación de unos polinomios. El enunciado del problema es el siguiente:

Sea un intervalo perteneciente a la recta de los reales de extremos  $\alpha$  y  $\beta$  que se divide en  $L$  subintervalos de igual longitud, que de aquí en adelante se llamarán *elementos*.



Cada uno de los  $L$  *elementos* se divide a su vez en  $M$  intervalos de igual longitud. Se definen, por tanto,  $M+1$  puntos en cada elemento, considerando los extremos del elemento  $L$ .

Se realiza un cambio de coordenadas en cada elemento de modo que su extremo izquierdo se hace corresponder al valor 0 y el derecho al valor  $(\beta - \alpha)/L$ . Por consiguiente, los  $M+1$  puntos del elemento, anteriormente definidos, son los puntos de coordenada  $\{x_1=0, x_2=(\beta - \alpha)/(L \cdot M), x_3=2 \cdot (\beta - \alpha)/(L \cdot M), \dots, x_{M+1}=M \cdot (\beta - \alpha)/(L \cdot M)\}$ , que les llamaremos *nodos*.

Queremos, ahora, definir en el elemento  $L$ ,  $M+1$  polinomios de grado  $M$  de modo que su valor sea nulo en todos los nodos salvo en uno de ellos en el que ha de valer 1.

La colección de polinomios se obtendrá haciendo

$$\begin{aligned} N_j(x_i) &= 0 \text{ si } i \neq j \\ N_j(x_j) &= 1 \end{aligned}$$

Variando  $i$  y  $j$  desde 1 hasta  $M+1$ .

Se trata de hallar los coeficientes de estos polinomios.

$$N(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_M x^M$$

Como son de grado  $M$ , cada polinomio tendrá  $M+1$  coeficientes, que son  $a_0, a_1, \dots, a_M$ . De esta manera se plantearán  $M+1$  ecuaciones por cada polinomio correspondientes al valor que éste toma en los  $M+1$  puntos.

```
> restart;
> with(linalg):
```

Warning, new definition for norm

Warning, new definition for trace

Grado de los polinomios:  $M$

```
> M:=2;
```

$$M := 2$$

Número de elementos

```
> L:=20;
```

$$L := 20$$

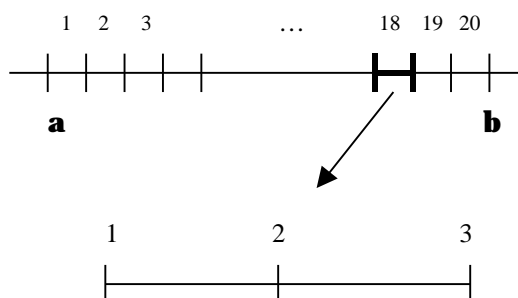
```
> H:=vector(L,0.1);
```

$$H := [.1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1, .1]$$

Se define  $H$  como un vector de  $L$  elementos, todos de valor 0.1. En este caso se ha dividido el intervalo  $[\alpha, \beta]$  en 20 subintervalos, elementos, de longitud 0.1. A continuación se determina el valor de los extremos del intervalo:

```
> alpha:=0.;beta:=sum(H[j],j=1..L);
```

$$\alpha := 0$$

$$\beta := 2.0$$


Se define un procedimiento  $h$  que para, cada elemento  $j$ , define la posición de los nodos del elemento:

```
> h:= proc(j,i)
> H[j]*i/M
> end;
```

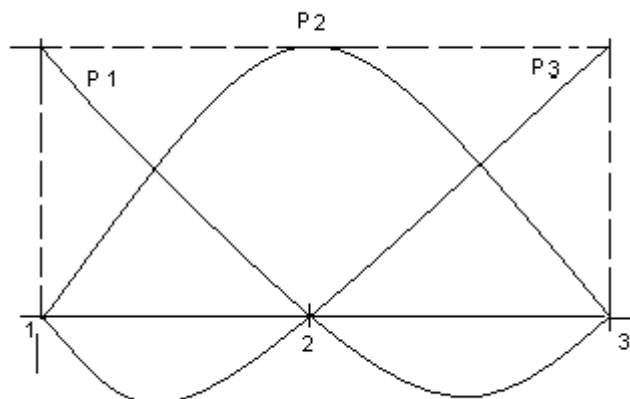
En este ejemplo  $M = 2$  y  $H = 0.1$ ; si se opera, se obtiene  $0.05*i$  y si  $i$  toma valores entre 0 y 2, para cada elemento:

- cuando  $i = 0 \Rightarrow h = 0$ ; estamos en el primer nodo del elemento
- cuando  $i = 1 \Rightarrow h = 0.05$  estamos en el segundo nodo
- cuando  $i = 2 \Rightarrow h = 0.1$  estamos en el tercer nodo

Seguidamente se definen los polinomios - repárase que en el ejemplo, en cada elemento hay  $M+1$  polinomios, es decir 3 - :

```
> N:=proc(j,i,x) local r;
> a[j][i,1]+sum(a[j][i,r]*x^(r-1), r=2..M+1)
> end;
```

$a[j][i,r]$  es el coeficiente del polinomio  $i$  del elemento  $j$ , correspondiente al término de grado  $(r-1)$ ;  $x$  es la variable del polinomio.



Por último, se calculan los coeficientes de los polinomios. Para ello se define un procedimiento *interpol* en el que se crea un vector, *equat*, cuyas componentes se refieren a cada uno de los elementos,  $j$ . Estas componentes contienen todas las ecuaciones necesarias para definir los valores de los polinomios en cada nudo (1, 2, y 3) del elemento  $j$ . Al resolver estas ecuaciones obtenemos los valores de los coeficientes de estos polinomios.

Por ejemplo, se va a analizar  $\text{seq}(\text{seq}(N(j,ii,h(j,k))), k=0..ii-2), ii=2..M+1)$

- cuando  $ii = 2$  y  $k = 0$  tenemos  $N(j,2,h(j,0) = 0)$ . Para resolver se iguala a cero y así se obtiene que el segundo polinomio (2) de todos los elementos ( $j$ ) en el primer nudo ( $h(j, 0) = 0$ ) toma el valor cero
- cuando  $ii = 3$  y  $k = 0$  tenemos  $N(j,3,0)$ , lo que indica que el polinomio 3 de cualquier elemento en el primer nudo toma valor cero
- cuando  $ii = 3$  y  $k = 1$ ; tenemos  $N(j,3,0.05)$ , lo que indica que el polinomio 3 de cualquier elemento en el nudo 2 ( $h(j, 1) = 0.05$ ) vale cero

```
> interpol:=proc(j) local i,k,ii,iii,l;
> global equat, h, N, DN,x;
> equat[j]:=seq(seq(N(j,ii,h(j,k))),k=0..ii-2),ii=2..M+1),
seq(seq(N(j,iii,h(j,k))), k=iii..M),iii=1..M+1),N(j,1,h(j,0))-1,
seq(N(j,k,h(j,k-1))-1,k=1..M+1) end:
> solve({seq(interpol(l),l=1..L)}):
> assign(%):
```

## 11.6. 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.

### 11.6.1. 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 “;”.

### 11.6.2. 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
```

### 11.6.3. 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 puede 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]
```

### 11.6.4. 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 *debugger*:

```
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’

### 11.6.5. 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 *debugger*.

## 12. COMANDOS DE ENTRADA/SALIDA

### 12.1. CONTROL DE LA INFORMACIÓN DE SALIDA

La información de salida de Maple se puede controlar con el parámetro *prettyprint*, en la función *interface*. De ordinario este parámetro tiene por valor 1. Para que la salida sea en una sola línea hay que hacer *prettyprint=0*. Por ejemplo, ejecute los siguientes comandos y observe el resultado:

```
> interface(prettyprint=0):
> a*x + b**x = c;
  a*x+b^x = c
> interface(prettyprint=2):
```

### 12.2. LAS LIBRERÍAS DE MAPLE

La librería de Maple tiene 4 partes:

- 1.- standard library.
- 2.- miscellaneous library.
- 3.- packages.
- 4.- share library (librería de funciones desarrolladas por usuarios).

Cuando se utiliza una función o rutina de la *standard* library, el sistema la carga automáticamente en la memoria principal. Por el contrario, la *miscellaneous* library contiene funciones menos frecuentemente utilizadas y hay que cargarlas explícitamente con el comando *readlib*.

Hay también *paquetes* de funciones para operaciones más especializadas. Para cargar estas rutinas hay que hacer mención también al paquete correspondiente. Véanse algunos ejemplos:

```
> a:=evaln(a); f := exp(a*z)/(1+exp(z)); # una fórmula cualquiera
> residue(f, z=Pi*I); # para calcular el residuo en un punto
> readlib(residue); # esta rutina está en la miscellaneous library
  proc(f,a) ... end
> residue(f, z=Pi*I); # ahora sí lo va a calcular bien
> residue(1/(z**2+a**2), z=a*I);
```

Por ejemplo, el paquete *orthopoly* contiene funciones para varios tipos de polinomios ortogonales. Para llamar a una de estas funciones:

```
> orthopoly[T](4,x); # polinomios de Chebyshev
> with(orthopoly, T); with(orthopoly);
```

Se puede indicar a Maple que cargue parte o todo el paquete, aunque en realidad se cargan sólo los nombres y no los códigos fuente.

### 12.3. GUARDAR Y RECUPERAR EL ESTADO DE UNA HOJA DE TRABAJO

El estado y los comandos de una hoja de trabajo se puede guardar desde dicha hoja con el comando *save*. Dos formas posibles de este comando son las siguientes:

```
> save `c:\\temp\\mifile.m`; # fichero no legible con Notepad
> save `c:\\linf2\\maple\\mifile.m`;
> save A,B,C,`mifile.txt`; # fichero legible con Notepad
```



Si el nombre del fichero termina en *\*.m* se almacena en formato interno de Maple. En otro caso, se almacena en forma de texto (el formato de la salida del comando *lprint*) con las sentencias de entrada.

Para recuperar el fichero guardado se utiliza el comando *read*:

```
> read `c:\\linf2\\maple\\mifile.m`;
> evaln(A), evaln(B), evaln(C); read `c:\\temp\\mifile.txt`;
```

Si se ejecuta *Save As* en el menú *File* se guarda también todo el estado interno de la hoja de trabajo.

## 12.4. LECTURA Y ESCRITURA DE FICHEROS

En Maple hay que distinguir entre varios tipos de ficheros:

- 1.- Ficheros que contienen código objeto para Maple.
- 2.- Ficheros que contienen código para Maple en formato legible por el usuario.
- 3.- Ficheros de resultados de Maple, pero que no pueden ser reutilizados como entrada del programa en una sesión posterior.
- 4.- Ficheros de input/output formateados, que pueden ser leídos o escritos por otros programas.

Comenzaremos viendo la forma de crear ficheros de resultados de Maple, que no pueden ser reutilizados como entrada. Esto se puede conseguir redireccionando la salida de la pantalla mediante los comandos *writeto* o *appendto* para que dicha salida se escriba en un fichero llamado por ejemplo *outfile*

```
> writeto(outfile): # o bien,
> appendto(outfile):
```

Para que la salida se vuelva a escribir en la pantalla basta ejecutar el comando:

```
> writeto(terminal):
```

Hay que tener en cuenta que el fichero *outfile* se escribirá en el directorio principal de Maple. Para que este fichero se pudiera utilizar como entrada, habría que hacer ciertos retoques. Sin embargo, no hay inconveniente en hacer *copy* y *paste* de ciertas cosas entre este fichero y la hoja de trabajo.

Como ya se ha visto, con el siguiente comando:

```
> save filename;
```

pueden guardarse los cálculos realizados de modo que se pueden utilizar de nuevo en una sesión posterior, o en un momento posterior de la misma sesión. Los ficheros guardados con *save* pueden leerse posteriormente desde Maple por medio de la función *read* en la forma:

```
> read filename;
```

Una vez leído el fichero, sus variables y resultados son utilizables inmediatamente.

Si el fichero guardado con *save* termina en *\*.m* se utiliza el formato objeto de Maple, no legible directamente por el usuario. En caso contrario, el fichero es legible por el usuario. Si el nombre del fichero termina en *\*.m* –y siempre que tenga extensión– hay que encerrarlo entre apóstrofes para evitar que Maple confunda el punto con el operador de concatenación. Si se da el *path* absoluto de un fichero, las barras invertidas hay que ponerlas dos veces (\\).

Los ficheros legibles pueden prepararse con cualquier editor de textos.

El siguiente ejemplo habría que ejecutarlo en una sesión nueva de Maple:

```
> restart; polinomio:= x**2+2*x+1; `numero cuatro` := 4;
> save `c:\\temp\\filepufo.m`;
> restart; polinomio; `numero cuatro`;
> read `c:\\temp\\filepufo.m`; polinomio; `numero cuatro`;
```

Cuando el fichero que se lee está en formato *\*.m* su contenido no se vuelca en la pantalla al cargarse, mientras que sí se vuelca si es un fichero legible que tiene una extensión diferente.

Puede salvarse selectivamente parte del contenido de una sesión, por ejemplo:

```
> save `numero cuatro`, polinomio, `c:\\temp\\filepuf2`;
> `numero cuatro`:=3; polinomio:= x**5; # se cambia el valor de lo salvado
> read `c:\\temp\\filepuf2`; # ahora se recupera el fichero
```

La función **readlib** puede utilizarse para leer una función de una librería. Además, en la misma llamada a **readlib** para leer una función, se le pueden pasar los parámetros a esa función, como en el ejemplo siguiente:

```
> readlib(mttaylor)(sin(x+y), [x, y]);
> sort(simplify(%, {z=x+y}, [x, y, z]));
> subs(z=x+y, %);
```

## 12.5. FICHEROS FORMATEADOS

Se verán ahora los ficheros de Input/Output formateados. Maple puede leer/escribir ficheros creados por, o para ser leídos por otras aplicaciones. En principio escribe en la pantalla, pero se puede redireccionar la salida con **writeto**. Las funciones de entrada/salida que se van a ver ahora son similares a las de C. La forma general de la función de escritura es:

```
> printf(format, arg1, arg2, ...)
```

que imprime sus argumentos con el formato indicado. Véase un ejemplo:

```
> printf(`%015.8f`, 5/3);
000001.66666667
```

Por otra parte, también se pueden leer ficheros con la función **sscanf**, que tiene la siguiente forma general:

```
> sscanf(string, format)
```

Esta sentencia es similar a la de C, pero aquí devuelve una lista de los objetos leídos. Posibles ejemplos serían los siguientes:

```
> sscanf(`x = 123.45, y = 6.7E-8, 9.10`, `x = %f, y = %g, %d.%d`);
> sscanf(`f = x/(1+x**2)`, `f = %a`);
```

## 12.6. GENERACIÓN DE CÓDIGO

Una de las aplicaciones más interesantes de Maple es la generación de ficheros con código C o Fortran (el código de MATLAB se parece bastante al de Fortran) a partir de las expresiones desarrolladas en el programa. Esto se puede hacer directamente utilizando el menú contextual o de la forma tradicional, introduciendo las órdenes por el teclado. Considérense los siguientes ejemplos:

```
> x:= evaln(x); ecuacion := x**3 - 5*a*x**2 = 1;
> soluciones := solve(ecuacion, x);
> sol1 := soluciones[1];
```

```
> precision := double; # se va a generar código Fortran de doble precision
> fortran(soll, 'optimized');
> readlib(cost)(soll); # para saber el costo en operaciones aritméticas
> cost(optimize(soll));
> fortran(soll, 'optimized', filename = `c:\\temp\\filepuf3.f`);
```

La sentencia anterior produce salida a fichero. Si el fichero ya existe lo añade a continuación de lo que hay (*append*).

```
> readlib(C); # para generar código en C
> C(soll, 'optimized');
> C(soll);
```

La forma general de la función C es la siguiente:

```
> C(s, filename=`g:\\nombre.c`)
```

donde **s** es una expresión, un array de expresiones con un nombre, o una lista de ecuaciones en la forma *name = algebraic*, entendiendo *algebraic* como una secuencia de sentencias de asignación.

Puede utilizarse también el argumento adicional *optimized*, y la variable global *precision* puede ser asignada a *double*. Los subíndices se traducen directamente, sin tener en cuenta que en C se empieza a contar por cero.

Considérese un ejemplo adicional:

```
> readlib(C);
> s := ln(x)+2*ln(x)^2-ln(x)^3;
> C(s);
    t0 = log(x)+2.0*pow(log(x),2.0)-pow(log(x),3.0);
> C(s,optimized);
    t1 = log(x);
    t2 = t1*t1;
    t4 = t1+2.0*t2-t2*t1;
```