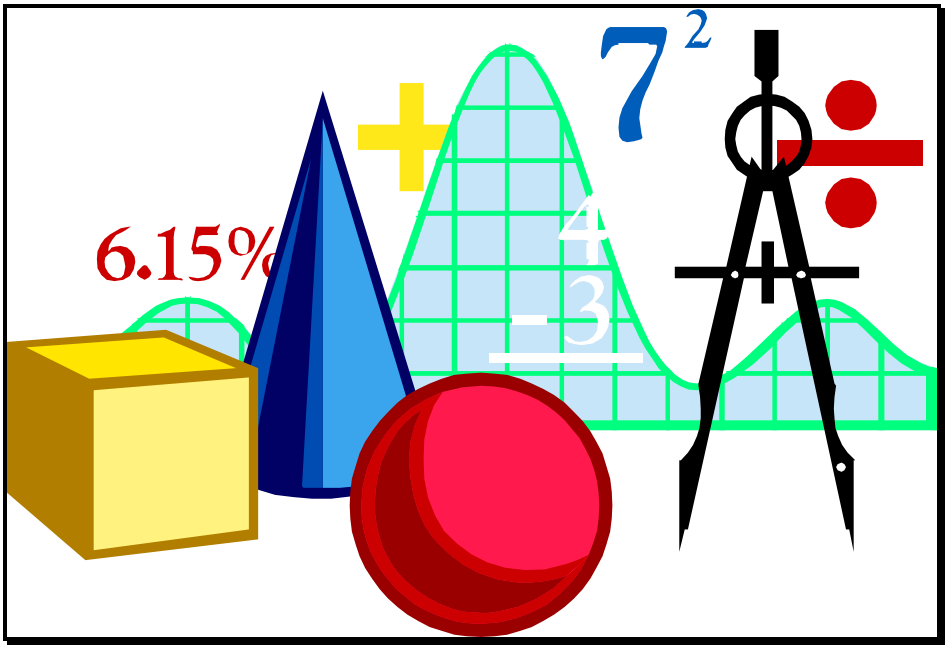
ESCUELA SUPERIOR DE INGENIEROS INDUSTRIALES
UNIVERSIDAD DE NAVARRA

INDUSTRI INJINERUEN GOIMAiLAKO ESKOLA
NAFARROAKO UNIBERTSITATEA



Aprenda Matlab 4.2

como si estuviera en Primero

San Sebastián, 1 de Septiembre de 1997

Colección preparada por:

Javier García de Jalón de la Fuente
Rufino Goñi Lasheras
José María Sarriegui Domínguez
Iñigo Girón Legorburu
Ignacio Rodríguez Garrido
Alfonso Brazález Guerra
Patxi Funes Martínez

ÍNDICE

1. INTRODUCCIÓN 1
1.1 El programa MATLAB 1

2. OPERACIONES CON MATRICES Y VECTORES 2
2.1 Definición de matrices desde teclado 2
2.2 Operaciones con matrices 4
2.3 Control de los formatos de salida 7
2.4 Tipos de datos 7
2.5 Variables y expresiones matriciales 9
2.6 Otras formas de definir matrices 10
2.6.1 Tipos de matrices predefinidos 10
2.6.2 Formación de una matriz a partir de otras 11
2.6.3 Direccionamiento de vectores y matrices a partir de vectores 12
2.6.4 Operador dos puntos (:) 13
2.6.5 Matriz vacía A[ ] 15
2.6.6 Definición de vectores y matrices a partir de un fichero 15
2.6.7 Definición de vectores y matrices mediante funciones y declaraciones 16
2.7 Operadores relacionales 16
2.8 Operadores lógicos 16
2.9 Bifurcaciones y bucles 17
2.9.1 Sentencia *if* 17
2.9.2 Sentencia *for* 18
2.9.3 Sentencia *while* 18
2.9.4 Sentencia *break* 19
2.10 Uso del *Help* 19

3. FUNCIONES DE LIBRERÍA 19
3.1 Camino de búsqueda (*search path*) de MATLAB 19
3.2 Características generales de las funciones de MATLAB 20
3.3 Funciones matemáticas elementales que operan de modo escalar 22
3.4 Funciones que actúan sobre vectores 23
3.5 Funciones que actúan sobre matrices 23
3.5.1 Funciones matriciales elementales: 23
3.5.2 Funciones matriciales especiales 23
3.5.3 Funciones de factorización y/o descomposición matricial 24
3.6 Más sobre operadores relacionales con vectores y matrices 25
3.7 Otras funciones que actúan sobre vectores y matrices 27

4. OTROS ASPECTOS DE MATLAB 27
4.1 Guardar valores de variables y estados de una sesión 27
4.2 Guardar sesión y copiar salidas 28
4.3 Líneas de comentarios 28
4.4 Cadenas de texto 28
4.5 Funciones para cálculos con polinomios 29
4.6 Medida de tiempos y de esfuerzo de cálculo 30
4.7 Llamada a comandos del sistema operativo 30
4.8 Funciones de función 31
4.8.1 Integración numérica 31
4.8.2 Ecuaciones no lineales y optimización 32
4.8.3 Integración numérica de ecuaciones diferenciales 32

5. GRÁFICOS BIDIMENSIONALES 34
5.1 Funciones gráficas 2D elementales 34
5.1.1 Función *plot* 36
5.1.2 Estilos de línea y marcadores en la función *plot* 37
5.1.3 Añadir líneas a un gráfico ya existente 38
5.1.4 Comando *subplot* 38

5.1.5 Control de los ejes 39
5.2 Control de ventanas gráficas: función *figure* 39
5.3 Otras funciones gráficas 2-D 40
5.3.1 Función *fplot* 40
5.3.2 Función *fill* para polígonos 41
5.4 Entrada de puntos con el ratón 42
5.5 Preparación de películas o "movies" 42
5.6 Impresión de las figuras en impresora láser 43

6. GRÁFICOS TRIDIMENSIONALES 44
6.1 Tipos de funciones gráficas tridimensionales 44
6.1.1 Dibujo de líneas: función *plot3* 45
6.1.2 Dibujo de mallados: funciones meshgrid, mesh y surf 46
6.1.3 Dibujo de líneas de contorno: funciones *contour* y *contour3* 47
6.2 Utilización del color en gráficos 3-D 48
6.2.1 Mapas de colores 48
6.2.2 Imágenes y gráficos en pseudocolor 48
6.2.3 Dibujo de superficies faceteadas 49
6.2.4 Otras formas de las funciones *mesh* y *surf* 49
6.2.5 Formas paramétricas de las funciones *mesh*, *surf* y *pcolor* 50
6.2.6 Otras funciones gráficas 3D 50
6.2.7 Elementos generales: ejes, puntos de vista, líneas ocultas, ... 50

7. PROGRAMACIÓN DE MATLAB 51
7.1 Bifurcaciones y bucles 51
7.2 Lectura y escritura interactiva de variables 51
7.2.1 función *input* 51
7.2.2 función *disp* 51
7.3 Ficheros *.m 52
7.3.1 Ficheros de comandos 52
7.3.2 Definición de funciones 53
7.3.3 Help para las funciones de usuario 53
7.3.4 Variables globales 54
7.4 Cadenas de caracteres 54
7.4.1 Funciones para manejo de cadenas de caracteres 54
7.4.2 Las funciones *eval* y *feval* 55
7.5 Entrada y salida de datos 55
7.5.1 Importar datos de otras aplicaciones 55
7.5.2 Exportar datos a otras aplicaciones 56
7.6 Lectura y escritura de ficheros 56
7.6.1 Funciones *fopen* y *fclose* 56
7.6.2 Funciones *fscanf, sscanf, fprintf* y *sprintf* 57
7.6.3 Funciones *fread* y *fwrite* 58
7.6.4 Ficheros de acceso directo 58
7.7 Recomendaciones generales de programación 58

8. CONSTRUCCIÓN DE INTERFACES GRÁFICAS CON MATLAB 58
8.1 Estructura de los gráficos de MATLAB 58
8.1.1 Objetos gráficos de MATLAB 59
8.1.2 Identificadores (*Handles*) 59
8.2 Propiedades de los objetos 60
8.2.1 Funciones *set()* y *get()* 60
8.2.2 Propiedades por defecto 62
8.2.3 Funciones de utilidad 62

9. CREACIÓN DE CONTROLES GRÁFICOS 63
9.1 Opciones del comando *uicontrol* 63
9.1.1 Color del objeto (*BackGroundColor)* 63
9.1.2 Acción a efectuar por el comando (*CallBack*) 63
9.1.3 Control Activado/Desactivado (*Enable)* 63

9.1.4 Alineamiento Horizontal del titulo (*HorizontalAlignment*) 63
9.1.5 Valor Máximo (*Max*) 63
9.1.6 Valor Mínimo (*Min*) 64
9.1.7 Control del objeto padre (*Parent*) 64
9.1.8 Posición del Objeto (*Position*) 64
9.1.9 Nombre del Objeto (*String*) 64
9.1.10 Tipo de Control (S*tyle*) 64
9.1.11 Unidades (*Units*) 64
9.1.12 Valor (*Value*) 64
9.1.13 Visible (*Visible*) 64
9.2 Tipos de *uicontrol* **65**
9.2.1 Botones (*push buttons*) 65
9.2.2 Botones de selección (*check boxes*) 65
9.2.3 Botones de opción (*radio buttons*) 66
9.2.4 Barras de desplazamiento (*scrolling bars o sliders*) 67
9.2.5 Cajas de selección desplegables (*pop-up menus*) 68
9.2.6 Cajas de texto (*static text boxes*) 69
9.2.7 Cajas de texto editables (*editable text boxes*) 69
9.2.8 Marcos (*frames*) 69

10. CREACIÓN DE MENÚS 70
10.1 Descripción de las propiedades **71**
10.1.1 Acelerador (*Accelerator*) 71
10.1.2 Acción a efectuar por el menú (*CallBack*) 71
10.1.3 Creación de submenus (*Children*) 71
10.1.4 Menú activado/desactivado (*Enable*) 71
10.1.5 Nombre del menú (*Label*) 71
10.1.6 Control del objeto padre (*Parent*) 72
10.1.7 Posición del Menú (*Position*) 72
10.1.8 Separador (*Separator*) 72
10.1.9 Visible (*Visible*) 72
10.2 Ejemplo de utilización del comando *uimenu* **73**

11. DEBUGGER 74
11.1 ¿Qué es un *debugger*? **74**
11.2 Comandos del *debugger*. **75**
11.3 Utilización del *debugger*. **75**
11.4 Ejemplo de utilización del *debugger*. **76**
11.4.1 Inicio de una sesión de *debugger* 76
11.4.2 Ejecución el fichero *.m 76
11.4.3 Comprobación del valor de las variables. 77
11.4.4 Cambio de espacio de trabajo. 77
11.4.5 Creacion de una nueva variable. 77
11.4.6 Fin del proceso de *debug*. 77

1. Introducción

1.1 El programa MATLAB

MATLAB es el nombre abreviado de “MATrix LABoratory”. MATLAB es un programa para realizar cálculos numéricos con vectores y matrices. Como caso particular puede también trabajar con números escalares, tanto reales como complejos. Una de las capacidades más atractivas es la de realizar una amplia variedad de gráficos en dos y tres dimensiones. MATLAB tiene también un lenguaje de programación propio.

MATLAB se puede arrancar como cualquier otra aplicación de ***Windows***, clicando dos veces en el icono correspondiente (en ***Windows 95***, se arranca por medio del menú ***Start***). Al arrancar MATLAB se abre una ventana del tipo de la indicada en la figura 1.

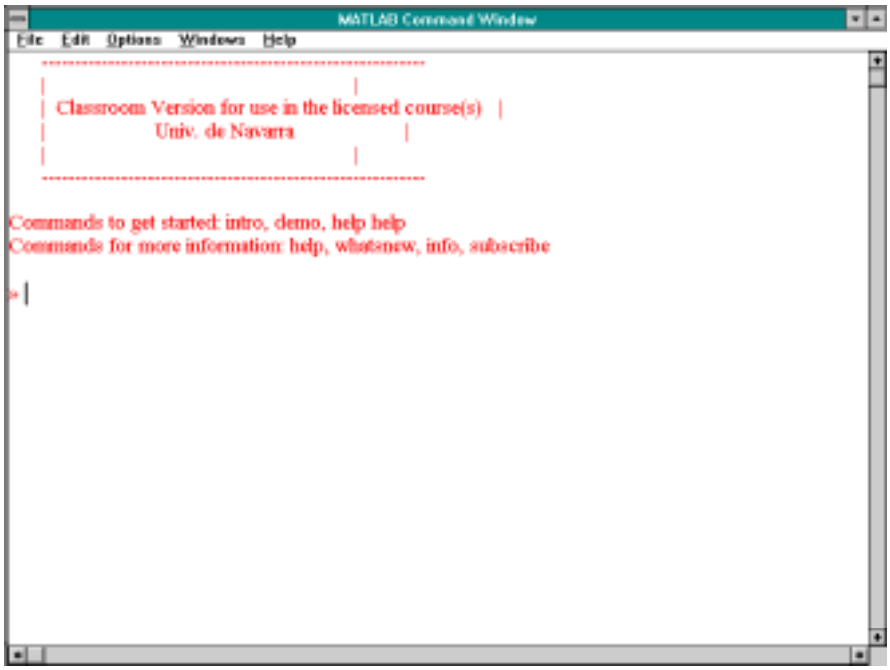


Figura 1. Ventana inicial de MATLAB.

En la ventana inicial se sugieren ya algunos comandos para el usuario inexperto que quiere echar un vistazo a la aplicación. En dicha ventana aparece también el ***prompt*** característico de MATLAB (»). Esto quiere decir que el programa está preparado para recibir instrucciones. El saludo inicial personalizado es reflejo de un fichero de comandos personal que se ejecuta cada vez que se entra en el programa (el fichero ***startup.m***, en el directorio *Matlab*)

Para apreciar desde el principio la potencia de MATLAB, se puede comenzar por escribir la siguiente línea a continuación del *prompt*. Al final hay que pulsar ***intro***.

```
» A=rand(6), B=inv(A), B*A
```

En realidad, en la línea anterior se han escrito tres instrucciones diferentes, separadas por comas. Como consecuencia, la respuesta del programa tiene tres partes también, cada una de ellas correspondiente a una de las instrucciones. Con la primera instrucción se define una matriz cuadrada (6x6) llamada **A**, cuyos elementos son números aleatorios entre cero y uno

(aunque aparezcan sólo 4 cifras, han sido calculados con 16 cifras). En la segunda instrucción se define una matriz **B** que es igual a la inversa de **A**. Finalmente se han multiplicado **B** por **A** y se comprueba que el resultado es la matriz unidad.

Otro de los puntos fuertes de MATLAB son los gráficos, que se verán con más detalle en una sección posterior. A título de ejemplo, se puede teclear la siguiente línea y pulsar ***intro***:

```
» x=-4:.01:4; y=sin(x); plot(x,y), grid, title('Función seno(x)')
```

Se puede observar que se abre una nueva ventana en la que aparece representada la función ***sin(x)***. Esta figura tiene un título "Función seno(x)" y una cuadrícula o "grid". En realidad la línea anterior contiene también varias instrucciones separadas por comas o puntos y comas. En la primera se crea un vector **x** con valores entre -4 y 4 separados por una centésima. A continuación se crea un vector **y,** cada uno de cuyos elementos es el seno del correspondiente elemento del vector **x**. Después se dibujan los valores de **y** en ordenadas frente a los de **x** en abscisas. Las dos últimas instrucciones establecen la cuadrícula y el título.

Un pequeño aviso antes de seguir adelante. Es posible recuperar comandos anteriores de MATLAB y moverse por dichos comandos con las teclas-flechas ↑ y ↓. Al pulsar la primera de dichas flechas aparecerá el comando que se había introducido inmediatamente antes. De modo análogo es posible moverse sobre la línea de un comando con las teclas ← y →.

Si se desea salir del programa, basta teclear los comandos ***quit*** o ***exit***, o bien elegir ***Exit*** MATLAB en el menú ***File*** (también se puede utilizar el ***Alt+F4*** de toda la vida).

2. Operaciones con matrices y vectores

Ya se ha comentado que MATLAB es fundamentalmente un programa para cálculo matricial. Inicialmente se utilizará MATLAB como *programa interactivo*, en el que se irán definiendo las matrices, los vectores y las expresiones que los combinan y obteniendo los resultados sobre la marcha. Si estos resultados son asignados a otras variables podrán ser utilizados posteriormente en otras expresiones. En este sentido MATLAB sería como una potente calculadora matricial (ya se verá que en realidad es esto y mucho más...).

Antes de tratar de hacer cálculos complicados, la primera tarea será aprender a introducir matrices y vectores desde el teclado. Más adelante se verán otras formas más potentes de definir matrices y vectores.

2.1 Definición de matrices desde teclado

Como en el caso de todos los lenguajes de programación, en MATLAB las matrices y vectores son variables que tienen *nombres*. Ya se verá luego con más detalle las reglas que deben cumplir estos nombres. Por el momento se sugiere que se utilicen letras mayúsculas para matrices y minúsculas para vectores (MATLAB no exige esto, pero puede resultar útil).

Para definir una matriz *no hace falta establecer de antemano su tamaño* (de hecho, se puede definir un tamaño y cambiarlo posteriormente). MATLAB determina el número de filas y de columnas en función del número de elementos que se proporcionan. *Las matrices se definen por filas*; los elementos de una misma fila están separados por *blancos* o *comas*, mientras que las filas están separadas por pulsaciones ***intro*** o por caracteres *punto y coma* (;). Por ejemplo, el siguiente comando define una matriz **A** de dimensión (3x3):

```
» A=[1 2 3; 4 5 6; 7 8 9]
```

La respuesta del programa es la siguiente:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

A partir de este momento la matriz **A** está disponible para hacer cualquier tipo de operación con ella (además de valores numéricos, en la definición de una matriz o vector se pueden utilizar expresiones y funciones matemáticas). Por ejemplo, una sencilla operación con **A** es hallar su traspuesta. En MATLAB el apóstrofo (') es el símbolo de trasposición matricial. Para calcular **A'** (traspuesta de **A**) basta teclear lo siguiente (se añade a continuación la respuesta del programa):

```
» A'
ans =
     1     4     7
     2     5     8
     3     6     9
```

Como el resultado de la operación no ha sido asignado a ninguna otra matriz, MATLAB utiliza un nombre de variable por defecto (***ans***, de *answer*), que contiene el resultado de la última operación. La variable ***ans*** puede ser utilizada como operando en la siguiente expresión que se introduzca. También podría haberse asignado el resultado a otra matriz llamada **B**:

```
» B=A'
B =
     1     4     7
     2     5     8
     3     6     9
```

Ahora ya están definidas las matrices **A** y **B**, y es posible seguir operando con ellas. Por ejemplo, se puede hacer el producto **B*A** (deberá resultar una matriz simétrica):

```
» B*A
ans =
    66    78    90
    78    93   108
    90   108   126
```

En MATLAB los elementos de un vector se acceden poniendo el índice entre paréntesis (por ejemplo x(3) ó x(i)). Los elementos de las matrices se acceden poniendo los índices entre paréntesis, separados por una coma (por ejemplo A(1,2) ó A(i,j)). Las matrices *se almacenan por columnas* (aunque *se introduzcan por filas*, como se ha dicho), y teniendo en cuenta esto puede accederse a cualquier elemento de una matriz con un sólo subíndice. Por ejemplo, si **A** es una matriz (3x3) se obtiene el mismo valor escribiendo A(1,2) que escribiendo A(4).

Invertir una matriz es casi tan fácil como trasponerla. A continuación se va a definir una nueva matriz **A** no singular en la forma:

```
» A=[1 4 -3; 2 1 5; -2 5 3]
A =
     1     4    -3
     2     1     5
    -2     5     3
```

Ahora se va a calcular la inversa de **A** y el resultado se asignará a **B**. Para ello basta hacer uso de la función ***inv( )*** (la precisión o número de cifras del resultado se puede cambiar con el menú ***Options/Numeric Format***):

```
B=inv(A)
B =
    0.1803    0.2213   -0.1885
    0.1311    0.0246    0.0902
   -0.0984    0.1066    0.0574
```

Para comprobar que este resultado es correcto basta pre-multiplicar **A** por **B**;

```
» B*A
ans =
    1.0000    0.0000    0.0000
    0.0000    1.0000    0.0000
    0.0000    0.0000    1.0000
```

De forma análoga a las matrices, es posible definir un *vector fila* **x** en la forma siguiente (si los tres números están separados por *blancos* o *comas,* el resultado será un vector fila):

```
» x=[10 20 30]   % vector fila
x =
    10    20    30
```

MATLAB considera comentarios todo lo que va desde el carácter ***%*** hasta el final de la línea. Por el contrario, si los números están separados por *intros* o *puntos y coma* (;) se obtendrá *un vector columna*:

```
» y=[11; 12; 13]   % vector columna
y =
    11
    12
    13
```

MATLAB tiene en cuenta la diferencia entre vectores fila y vectores columna. Por ejemplo, si se intenta sumar los vectores **x** e **y** se obtendrá el siguiente mensaje de error:

```
» x+y
??? Error using ==> +
Matrix dimensions must agree.
```

Estas dificultades desaparecen si se suma **x** con el vector traspuesto de **y**:

```
» x+y'
ans =
    21    32    43
```

Aunque ya se ha visto en los ejemplos anteriores el estilo sencillo e intuitivo con el que MATLAB opera con matrices y vectores, a continuación se va a estudiar este tema con un poco más de detenimiento.

2.2 Operaciones con matrices

MATLAB puede operar con matrices por medio de ***operadores*** y por medio de ***funciones***. Se han visto ya los operadores *suma* (+), *producto* (*) y *traspuesta* ('), así como la función *invertir* ***inv( )***. Los operadores matriciales de MATLAB son los siguientes:

+	adición o suma
–	sustracción o resta
*	multiplicación
'	traspuesta
^	potenciación
\	división-izquierda
/	división-derecha

Estos operadores se aplican también a las variables o valores escalares, aunque con algunas diferencias (en términos de C++ podríamos decir que son operadores *sobrecargados*, es decir, con varios significados distintos dependiendo del contexto). Todos estos operadores son coherentes con las correspondientes operaciones matriciales: no se puede por ejemplo

sumar matrices que no sean del mismo tamaño. Si los operadores no se usan de modo correcto se obtiene un mensaje de error.

Los operadores anteriores se pueden aplicar también de modo *mixto*, es decir con un operador escalar y otro matricial. En este caso la operación con el escalar se aplica a cada uno de los elementos de la matriz. Considérese el siguiente ejemplo:

```
» A=[1 2; 3 4]
A =
     1     2
     3     4
» A*2
ans =
     2     4
     6     8
» A-4
ans =
    -3    -2
    -1     0
```

Los ***operadores de división*** requieren una cierta explicación adicional. Considérese el siguiente sistema de ecuaciones estándar,

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

en donde **x** y **b** son vectores columna, y **A** una matriz cuadrada invertible. La resolución de este sistema de ecuaciones se puede escribir en las 2 formas siguientes (¡Atención a la 2ª forma, basada en la barra invertida \, que puede resultar un poco extraña!):

$$\mathbf{x} = \text{inv}(\mathbf{A})*\mathbf{b} \tag{2a}$$

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{b} \tag{2b}$$

Así pues, el operador *división-izquierda* por una matriz (barra invertida \) equivale a premultiplicar por la inversa de esa matriz. En realidad este operador es más general de lo que aparece en el ejemplo anterior: el operador *división-izquierda* es aplicable aunque la matriz no tenga inversa e incluso no sea cuadrada, en cuyo caso la solución que se obtiene por lo general es la que proporciona el *método de los mínimos cuadrados*. En algunos casos se obtiene la solución de mínima norma sub-1. Por ejemplo, considérese el siguiente ejemplo de matriz (1x2) que conduce a un sistema de infinitas soluciones:

```
» A=[1 2], b=[2]
A =
     1     2
b =
     2
» x=A\b
x =
     0
     1
```

que es la solución cuya suma de valores absolutos de componentes (norma sub-1) es mínima. Por otra parte, en el caso de un sistema de ecuaciones *sobre-determinado* el resultado de MATLAB es el punto más “cercano” en el sentido de los mínimos cuadrados a las ecuaciones dadas (aunque no cumpla exactamente ninguna de ellas). Véase el siguiente ejemplo de tres ecuaciones formadas por una recta que no pasa por el origen y los dos ejes de coordenadas:

```
» A=[1 2; 1 0; 0 1], b=[2 0 0]'
A =
     1     2
     1     0
     0     1
```

```
b =
     2
     0
     0
» x=A\b
x =
    0.3333
    0.6667
» x=A\b, A*x-b
x =
    0.3333
    0.6667
ans =
   -0.3333
    0.3333
    0.6667
```

Aunque no es una forma demasiado habitual, también se puede escribir un sistema de ecuaciones lineales en una forma correspondiente a la traspuesta de la ecuación (1):

$$\mathbf{yB} = \mathbf{c} \tag{3}$$

donde **y** y **c** son vectores пια (**c** conocido). Si la matriz **B** es cuadrada e invertible, la solución de este sistema se puede escribir en las formas siguientes:

$$\mathbf{y} = \mathbf{c}*\text{inv}(\mathbf{B}) \tag{4a}$$

$$\mathbf{y} = \mathbf{c}/\mathbf{B} \tag{4b}$$

En este caso, el operador *división-derecha* por una matriz (/) equivale a postmultiplicar por la inversa de la matriz. Si se traspone la ecuación (3) y se halla la solución aplicando el operador *división-izquierda* de obtiene:

$$\mathbf{y'} = (\mathbf{B'})\backslash\mathbf{c'} \tag{5}$$

Comparando las expresiones (4b) y (5) se obtiene la relación entre los operadores *división-izquierda* y *división-derecha* (MATLAB sólo tiene implementado el operador *división-izquierda*):

$$\mathbf{c}/\mathbf{B} = ((\mathbf{B'})\backslash\mathbf{c'})' \tag{6}$$

En MATLAB existe también la posibilidad de aplicar ***elemento a elemento*** los operadores matriciales (*, ^, \ y /). Para ello basta precederlos por un punto (.). Por ejemplo:

```
» [1 2 3 4]^2
??? Error using ==> ^
Matrix must be square.

» [1 2 3 4].^2
ans =
     1     4     9    16

» [1 2 3 4]*[1 -1 1 -1]
??? Error using ==> *
Inner matrix dimensions must agree.

» [1 2 3 4].*[1 -1 1 -1]
ans =
     1    -2     3    -4
```

2.3 Control de los formatos de salida

Los formatos de salida de MATLAB pueden controlarse fácilmente con el comando ***Numeric Format*** del menú ***Options***. En la figura 2 se ven las opciones que aparecen al elegir ese comando.

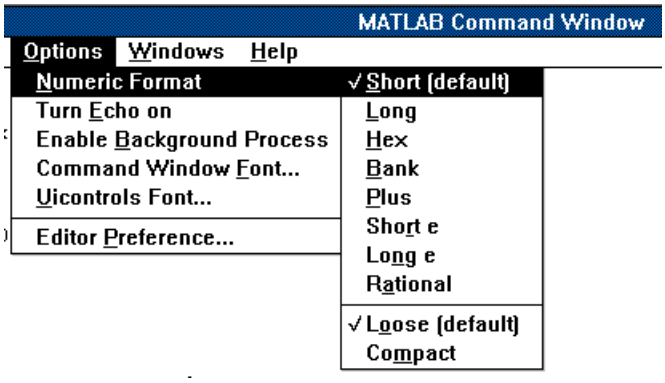


Figura 2. Comando ***Numeric Format*** del menú ***Options***.

Las mismas posibilidades se pueden activar por medio de comandos tecleados en la ventana de comandos de MATLAB. Los más importantes de estos comandos son los siguientes:

format short	coma fija con 4 decimales (defecto)
format long	coma fija con 15 decimales
format short e	notación científica con 4 decimales
format long e	notación científica con 15 decimales
format loose	introduce algunas líneas en blanco en la salida (defecto)
format compact	elimina las líneas en blanco citadas (opción recomendada)

Con ***Commnad Window Font*** el usuario tiene la posibilidad de elegir el tipo de letra –así como el tamaño y el color, tanto de las letras como del fondo– con la que se escribe en la ventana de comandos de MATLAB. Se sugiere al usuario que haga pruebas con estos comandos hasta conseguir el tipo de salida que más le guste.

Hay que añadir que MATLAB trata de mantener el formato de los números que han sido definidos como enteros (sin punto decimal). Si se elige la opción ***Numeric Format/Rational*** el programa trata de expresar los números racionales como cocientes de enteros.

2.4 Tipos de datos

Ya se ha dicho que MATLAB es un programa preparado para trabajar con vectores y matrices. Como caso particular también trabaja con variables escalares (matrices de dimensión 1). MATLAB trabaja siempre en *doble precisión*, es decir guardando cada dato en 8 bytes, con unas 15 cifras decimales exactas. Ya se verá más adelante que también puede trabajar con cadenas de caracteres (*strings*).

MATLAB mantiene una forma especial para los *números muy grandes* (más grandes que los que es capaz de representar), que son considerados como ***infinito***. Por ejemplo, obsérvese cómo responde el programa al ejecutar el siguiente comando:

```
» 1.0/0.0
Warning: Divide by zero
ans =
   Inf
```

Así pues, para MATLAB el ***infinito*** se representa como ***inf*** ó ***Inf***. MATLAB tiene también una representación especial para los resultados que no están definidos como números. Por ejemplo, ejecútense los siguientes comandos y obsérvense las respuestas obtenidas:

```
» 0/0
Warning: Divide by zero
ans =
   NaN
» inf/inf
ans =
   NaN
```

En ambos casos la respuesta es ***NaN***, que es la abreviatura de ***Not a Number***. Este tipo de respuesta, así como la de ***Inf***, son enormemente importantes en MATLAB, pues permiten controlar la fiabilidad de los resultados de los cálculos matriciales. Los ***NaN*** se propagan al realizar con ellos cualquier operación aritmética, en el sentido de que, por ejemplo, cualquier número sumado a un ***NaN*** da otro ***NaN***. MATLAB tiene esto en cuenta. Algo parecido sucede con los ***Inf***.

En muchos cálculos matriciales los datos y/o los resultados no son reales sino ***complejos***. MATLAB trabaja sin ninguna dificultad con números complejos. Para ver como se representan por defecto los números complejos, ejecútense los siguientes comandos:

```
» a=sqrt(-4)
a =
        0 + 2.0000i
» 3 + 4j
ans =
   3.0000 + 4.0000i
```

En la entrada de datos de MATLAB se pueden utilizar indistintamente la **i** y la **j** para representar el *número imaginario unidad* (en la salida, sin embargo, puede verse que siempre aparece la **i**). Si la **i** o la **j** no están definidas como variables, puede intercalarse el signo (*). Esto no es posible en el caso de que sí estén definidas, porque entonces se utiliza el valor de la variable. En general, cuando se está trabajando con números complejos, conviene no utilizar la **i** como variable ordinaria, pues puede dar lugar a errores y confusiones. Por ejemplo, obsérvense los siguientes resultados:

```
» i=2
i =
     2
» 2+3i
ans =
   2.0000 + 3.0000i
» 2+3*i
ans =
     8
» 2+3*j
ans =
   2.0000 + 3.0000i
```

Cuando **i** y **j** son variables utilizadas para otras finalidades, como *unidad imaginaria* puede utilizarse también la función ***sqrt(-1)***, o una variable a la que se haya asignado el resultado de esta función.

La asignación de *valores complejos* a vectores y matrices desde teclado puede hacerse de las dos formas que se muestran en el ejemplo siguiente (conviene hacer antes ***clear i***, para que **i** no esté definida como variable. Este comando se estudiará más adelante):

```
» A = [1+2i 2+3i; -1+i 2-3i]
A =
   1.0000 + 2.0000i   2.0000 + 3.0000i
  -1.0000 + 1.0000i   2.0000 - 3.0000i
» A = [1 2; -1 2] + [2 3; 1 -3]*i
A =
   1.0000 + 2.0000i   2.0000 + 3.0000i
  -1.0000 + 1.0000i   2.0000 - 3.0000i
```

Puede verse que es posible definir las partes reales e imaginarias por separado. En este caso sí es necesario utilizar el operador (*), según se muestra en el ejemplo anterior.

Es importante advertir que el *operador de matriz traspuesta* ('), aplicado a matrices complejas, produce la matriz conjugada y traspuesta. Existe una función que permite hallar simplemente la matriz conjugada (***conj()***) y el operador punto y apóstrofo que calcula simplemente la matriz traspuesta (**.'**).

2.5 Variables y expresiones matriciales

Ya han aparecido algunos ejemplos de ***variables*** y ***expresiones*** matriciales. Ahora se va a tratar de generalizar un poco lo visto hasta ahora.

Una ***variable*** es un nombre que se da a una entidad numérica, que puede ser una matriz, un vector o un escalar. El valor de esa variable, e incluso el tipo de entidad numérica que representa, puede cambiar a lo largo de una sesión de MATLAB o a lo largo de la ejecución de un programa. La forma más normal de cambiar el valor de una variable es colocándola a la izquierda del ***operador de asignación*** (=).

Una expresión de MATLAB puede tener las dos formas siguientes: asignando su resultado a una variable,

```
variable = expresión
```

o simplemente evaluando el resultado del siguiente modo,

```
expresion
```

en cuyo caso el resultado se asigna automáticamente a una variable interna de MATLAB llamada ***ans*** (de *answer*) que almacena el último resultado obtenido. Se considera por defecto que una expresión termina cuando se pulsa ***intro***. Si se desea que una expresión continúe en la línea siguiente, hay que introducir ***tres puntos*** (...) antes de pulsar ***intro***. También se pueden incluir varias expresiones en una misma línea separándolas por *comas* (,) o *puntos y comas* (;).

Si una expresión *termina en punto y coma* (;) su resultado se calcula, pero no se escribe en pantalla. Esta posibilidad es muy interesante, tanto para evitar la escritura de resultados intermedios, como para evitar la impresión de grandes cantidades de números cuando se trabaja con matrices de gran tamaño.

A semejanza de C, MATLAB distingue entre mayúsculas y minúsculas en los nombres de variables. Los ***nombres de variables*** deben empezar siempre por una letra y pueden constar de hasta 19 letras y números (en ANSI C este número era 31). El carácter (_) se considera como una letra. A diferencia del lenguaje C, no hace falta declarar las variables que se vayan a utilizar. Esto hace que se deba tener especial cuidado con no utilizar nombres erróneos en las variables, porque no se recibirá ningún aviso del ordenador.

Cuando se quiere tener una *relación de las variables* que se han utilizado en una sesión de trabajo se puede utilizar el comando ***who***. Existe otro comando llamado ***whos*** que proporciona además información sobre el tamaño, la cantidad de memoria ocupada y el carácter real o complejo de cada variable. Se sugiere utilizar de vez en cuando estos comandos en la sesión de MATLAB que se tiene abierta.

Se puede eliminar por ejemplo la variable **A** con el comando ***clear A***. Si se utiliza sin argumentos, el comando ***clear*** elimina todas las variables y funciones creadas previamente.

2.6 Otras formas de definir matrices

MATLAB dispone de varias formas de definir matrices. El introducirlas por teclado sólo es práctico en casos de pequeño tamaño y cuando no hay que repetir esa operación muchas veces. Recuérdese que en MATLAB no hace falta definir el tamaño de una matriz. Las matrices toman tamaño al ser definidas y este tamaño puede ser modificado por el usuario mediante adición y/o borrado de filas y columnas. A continuación se van a ver otras formas más potentes y generales de definir y/o modificar matrices.

2.6.1 TIPOS DE MATRICES PREDEFINIDOS

Existen en MATLAB varias funciones orientadas a definir con gran facilidad matrices de tipos particulares. Algunas de estas funciones son las siguientes:

eye(4)	forma la matriz unidad de tamaño (4x4)
zeros(3,5)	forma una matriz de ceros de tamaño (3x5)
zeros(4)	idem de tamaño (4x4)
ones(3)	forma una matriz de unos de tamaño (3x3)
ones(2,4)	idem de tamaño (2x4)
linspace(x1,x2,n)	genera un vector con **n** valores igualmente espaciados entre **x1** y **x2**
logspace(d1,d2,n)	genera un vector con **n** valores espaciados logarítmicamente entre 10^d1 y 10^d2. Si d2 es **pi**[1], los puntos se generan entre 10^d1 y **pi**
rand(3)	forma una matriz de números aleatorios entre 0 y 1, con distribución uniforme, de tamaño (3x3)
rand(2,5)	idem de tamaño (2x5)
randn(4)	forma una matriz de números aleatorios de tamaño (4x4), con distribución normal, de valor medio 0 y varianza 1
magic(4)	crea una matriz (4x4) con los números 1, 2, ... 4*4, con la propiedad de que todas las filas y columnas suman lo mismo
hilb(5)	crea una matriz de Hilbert de tamaño (5x5). La matriz de Hilbert es una matriz cuyos elementos (i,j) responden a la expresión (1/(i+j-1)). Esta es una matriz especialmente difícil de manejar por los grandes errores numéricos a los que conduce
invhilb(5)	crea directamente la inversa de la matriz de Hilbert

[1] **pi** es una variable predefinida en MATLAB, que como es fácil suponer representa el número π.

kron(x,y)	produce una matriz con todos los productos de los elementos del vector **x** por los elementos del vector **y**. Equivalente a x*y, donde **x** e **y** son vectores fila
compan(pol)	construye una matriz cuyo polinomio característico tiene como coeficientes los elementos del vector **pol** (ordenados de mayor grado a menor)
vander(v)	construye la matriz de Vandermonde a partir del vector **v** (las columnas son las potencias de los elementos de dicho vector)

Existen otras funciones para crear matrices de tipos particulares. Con ***Help/Index*** se puede obtener información sobre todas las funciones disponibles en MATLAB. Con ***Help/Table of Contents*** y ***ELMAT*** se obtiene información de algunas de estas funciones.

2.6.2 FORMACIÓN DE UNA MATRIZ A PARTIR DE OTRAS

MATLAB ofrece también la posibilidad de crear una matriz a partir de matrices previas ya definidas, por varios posibles caminos:

– recibiendo alguna de sus propiedades (como por ejemplo el tamaño),

– por composición de varias submatrices más pequeñas,

– modificándola de alguna forma.

A continuación se describen algunas de las funciones que crean una nueva matriz a partir de otra o de otras, comenzando por dos funciones auxiliares:

[m,n]=size(A)	devuelve el número de filas y de columnas de la matriz **A**. Si la matriz es cuadrada basta recoger el primer valor de retorno
n=length(x)	calcula el número de elementos de un vector **x**
zeros(size(A))	forma una matriz de ceros del mismo tamaño que una matriz **A** previamente creada
ones(size(A))	idem con unos
A=diag(x)	forma una matriz diagonal **A** cuyos elementos diagonales son los elementos de un vector ya existente **x**
x=diag(A)	forma un vector **x** a partir de los elementos de la diagonal de una matriz ya existente **A**
diag(diag(A))	crea una matriz diagonal a partir de la diagonal de la matriz **A**
triu(A)	forma una matriz triangular superior a partir de una matriz **A** (no tiene por qué ser cuadrada)
tril(A)	idem con una matriz triangular inferior
rot90(A,k)	Gira k*90 grados la matriz rectangular **A** en sentido antihorario. k es un entero que puede ser negativo. Si se omite, se supone k=1
flipud(A)	halla la matriz simétrica de **A** respecto de un eje horizontal
fliplr(A)	halla la matriz simétrica de **A** respecto de un eje vertical
reshape(A,m,n)	Cambia el tamaño de la matriz **A** devolviendo una matriz de tamaño *mxn* cuyas columnas se obtienen a partir de un vector formado por las columnas de **A** puestas una a continuación de otra. Si la matriz **A** tiene menos de *mxn* elementos se produce un error.

Un caso especialmente interesante es el de crear una nueva matriz *componiendo como submatrices* otras matrices definidas previamente. A modo de ejemplo, ejecútense las siguientes líneas de comandos y obsérvense los resultados obtenidos:

```
» A=rand(3)
» B=diag(diag(A))
» C=[A, eye(3); zeros(3), B]
```

En el ejemplo anterior, la matriz **C** de tamaño (6x6) se forma por composición de cuatro matrices de tamaño (3x3). Al igual que con simples escalares, las submatrices que forman una fila se separan con *blancos* o *comas*, mientras que las diferentes filas se separan entre sí con ***intros*** o *puntos y comas*. Los tamaños de las submatrices deben de ser coherentes.

2.6.3 DIRECCIONAMIENTO DE VECTORES Y MATRICES A PARTIR DE VECTORES

Los elementos de un vector **x** se pueden direccionar a partir de los de otro vector **v**. En este caso, **x(v)** equivale al vector x(v(1)), x(v(2)), ... Considérese el siguiente ejemplo:

```
» v=[1 3 4]
v =
     1     3     4
» x=rand(1,6)
x =
    0.5899    0.4987    0.7351    0.9231    0.1449    0.9719
» x(v)
ans =
    0.5899    0.7351    0.9231
```

De forma análoga, los elementos de una matriz **A** pueden direccionarse a partir de los elementos de dos vectores **f** y **c**. Véase por ejemplo:

```
» f=[2 4]; c=[1 2];
» A=magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
» A(f,c)
ans =
     5    11
     4    14
```

El siguiente ejemplo –continuación del anterior– permite comprobar cómo los elementos de una matriz se pueden direccionar con un sólo índice, considerando que las columnas de la matriz están una a continuación de otra formando un vector:

```
» f=[1 3 5 7];
» A(f), A(5), A(6)
ans =
    16     9     2     7
ans =
     2
ans =
    11
```

Más adelante se verá que esta forma de extraer elementos de un vector y/o de una matriz tiene abundantes aplicaciones, por ejemplo la de modificar selectivamente esos elementos.

2.6.4 OPERADOR DOS PUNTOS (:)

Este operador es muy importante en MATLAB y puede usarse de varias formas. Se sugiere al lector que practique mucho sobre los ejemplos contenidos en este apartado, introduciendo todas las modificaciones que se le ocurran y haciendo pruebas abundantes (¡Probar es la mejor forma de aprender!).

Para empezar, defínase un vector **x** con el siguiente comando:

```
» x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
```

En cierta forma se podría decir que el operador (:) representa un *rango*: en este caso, los números enteros entre el 1 y el 10. Por defecto el incremento es 1, pero este operador puede también utilizarse con otros valores enteros y reales, positivos o negativos. En este caso el incremento va entre el valor inferior y el superior, en las formas que se muestran a continuación:

```
» x=1:2:10
x =
     1     3     5     7     9
» x=1:1.5:10
x =
    1.0000    2.5000    4.0000    5.5000    7.0000    8.5000   10.0000
» x=10:-1:1
x =
    10     9     8     7     6     5     4     3     2     1
```

Puede verse que por defecto este operador produce vectores fila. Si se desea obtener un vector columna basta trasponer el resultado. El siguiente ejemplo genera una tabla de funciones seno y coseno. Ejecútese y obsérvese el resultado (recuérdese que con (;) después de un comando el resultado no aparece en pantalla).

```
» x=[0.0:pi/50:2*pi]';
» y=sin(x); z=cos(x);
» [x y z]
```

El operador dos puntos (:) es aún más útil y potente –y también más complicado– con matrices. A continuación se va a definir una matriz **A** de tamaño 6x6 y después se realizarán diversas operaciones sobre ella con el operador (:).

```
» A=magic(6)
A =
    35     1     6    26    19    24
     3    32     7    21    23    25
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
     4    36    29    13    18    11
```

A diferencia de C, MATLAB accede a los elementos de una matriz por medio de los índices de fila y de columna encerrados entre paréntesis y separados por una coma. Por ejemplo:

```
» A(2,3)
ans =
     7
```

El siguiente comando extrae los 4 primeros elementos de la 6ª fila:

```
» A(6, 1:4)
ans =
     4    36    29    13
```

Los dos puntos aislados representan "todos los elementos". Por ejemplo, el siguiente comando extrae todos los elementos de la 3ª fila:

```
» A(3, :)
ans =
    31     9     2    22    27    20
```

mientras que el siguiente extrae todos los elementos de las filas 3, 4 y 5:

```
» A(3:5,:)
ans =
    31     9     2    22    27    20
     8    28    33    17    10    15
    30     5    34    12    14    16
```

Se pueden extraer conjuntos disjuntos de filas utilizando *corchetes* [ ]. Por ejemplo, el siguiente comando extrae las filas 1, 2 y 5:

```
» A([1 2 5],:)
ans =
    35     1     6    26    19    24
     3    32     7    21    23    25
    30     5    34    12    14    16
```

En los ejemplos anteriores se han extraído filas y no columnas por motivos del espacio ocupado por el resultado en la hoja de papel. Es evidente que todo lo que se dice para filas vale para columnas y viceversa, basta cambiar el orden de los índices.

El operador dos puntos (:) puede utilizarse en ambos lados del operador (=). Por ejemplo, a continuación se va a definir una matriz identidad **B** de tamaño 6x6 y se van a reemplazar filas de **B** por filas de **A**. Obsérvese que la siguiente secuencia de comandos sustituye las filas 2, 4 y 5 de **B** por las filas 1, 2 y 3 de **A**,

```
» B=eye(size(A));
» B([2 4 5],:)=A(1:3,:)
B =
     1     0     0     0     0     0
    35     1     6    26    19    24
     0     0     1     0     0     0
     3    32     7    21    23    25
    31     9     2    22    27    20
     0     0     0     0     0     1
```

Se pueden realizar operaciones aún más complicadas como la siguiente[2]:

```
» B=eye(size(A));
» B(1:2,:)=[0 1; 1 0]*B(1:2,:)
```

Como nuevo ejemplo, se va a ver la forma de invertir el orden de los elementos de un vector:

```
» x=rand(1,5)
x =
    0.9103    0.7622    0.2625    0.0475    0.7361
» x=x(5:-1:1)
x =
    0.7361    0.0475    0.2625    0.7622    0.9103
```

Obsérvese que por haber utilizado paréntesis –en vez de corchetes– los valores generados por el operador (:) afectan a los índices del vector y no al valor de sus elementos.

Para invertir el orden de las columnas de una matriz se puede hacer lo siguiente:

---

[2] Se sustituyen las dos primeras filas de **B** por el producto de dichas filas por una matriz de permutación.

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
» A(:,3:-1:1)
ans =
     6     1     8
     7     5     3
     2     9     4
```

aunque hubiera sido más fácil utilizar la función ***fliplr(A)***.

Finalmente, hay que decir que ***A(:)*** representa un vector columna con las columnas de **A** una detrás de otra.

2.6.5 MATRIZ VACÍA A[ ]

Para MATLAB una matriz definida sin ningún elemento entre los corchetes es una matriz que *existe*, pero que está *vacía*, o lo que es lo mismo que tiene *dimensión cero*. Considérense los siguientes ejemplos de aplicación de las matrices vacías:

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
» B=[]
B =
     []
» exist(B)
ans =
     []
» isempty(B)
ans =
     1
» A(:,3)=[]
A =
     8     1
     3     5
     4     9
```

Las funciones ***exist()*** e ***isempty()*** permiten chequear si una variable existe y si está vacía. En el último ejemplo se ha eliminado la 3ª columna de **A** asignándole la matriz vacía.

2.6.6 DEFINICIÓN DE VECTORES Y MATRICES A PARTIR DE UN FICHERO

MATLAB acepta como entrada un fichero ***nombre.m*** (siempre con extensión ***.m***) que contiene instrucciones y/o funciones. Dicho fichero se llama tecleando simplemente su nombre sin la extensión. A su vez, un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso puede llamarse a sí mismo (funciones recursivas). Las variables definidas dentro de un fichero de comandos ****.m*** son ***variables globales***, esto es, pueden ser accedidas desde fuera de dicho fichero; no sucede lo mismo si el fichero ****.m*** corresponde a una función.

Como ejemplo se puede crear un fichero llamado ***unidad.m*** que construya una matriz unidad de tamaño 3x3 llamada ***U33*** en un directorio llamado *g:\matlab*. Este fichero deberá contener la línea siguiente:

```
U33=eye(3)
```

Desde MATLAB llámese al comando ***unidad*** y obsérvese el resultado. Entre otras razones, es muy importante utilizar ficheros de comandos para evitar teclear muchas veces los mismos datos, sentencias o expresiones.

2.6.7 DEFINICIÓN DE VECTORES Y MATRICES MEDIANTE FUNCIONES Y DECLARACIONES

También se pueden definir las matrices y vectores por medio de *funciones de librería* (las que se verán en la siguiente sección) y de *funciones programadas por el usuario* (que también se verán más adelante).

2.7 Operadores relacionales

El lenguaje de programación de MATLAB dispone de los siguientes operadores relacionales:

<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
==	igual que
~=	distinto que[3]

Obsérvese que, salvo el último de ellos, coinciden con los correspondientes operadores relacionales de C. Sin embargo, ésta es una coincidencia más bien formal. En MATLAB los operadores relacionales pueden aplicarse a vectores y matrices, y eso hace que tengan un significado especial.

Al igual que en C, si una comparación se cumple el resultado es 1 (***true***), mientras que si no se cumple es 0 (***false***). Recíprocamente, cualquier valor distinto de cero es considerado como ***true*** y el cero equivale a ***false***. La diferencia con C está en que cuando los operadores relacionales de MATLAB se aplican a dos matrices o vectores del mismo tamaño, la comparación se realiza elemento a elemento, y el resultado es otra matriz de unos y ceros del mismo tamaño que recoge el resultado de cada comparación entre elementos. Considérese el siguiente ejemplo como ilustración de lo que se acaba de decir:

```
» A=[1 2;0 3]; B=[4 2;1 5];
» A==B
ans =
     0     1
     0     0
» A~=B
ans =
     1     0
     1     1
```

2.8 Operadores lógicos

Los operadores lógicos de MATLAB son los siguientes:

&	and
|	or
~	negación lógica

[3] El carácter (~) se obtiene en los PCs pulsando sucesivamente las teclas 1, 2 y 6 manteniendo ***Alt*** pulsada.

Obsérvese que estos operadores lógicos tienen distinta notación que los correspondientes operadores de C (&&, || y !). Los operadores lógicos se combinan con los relacionales para poder comprobar el cumplimiento de condiciones múltiples. Más adelante se verán otros ejemplos y ciertas funciones de las que dispone MATLAB para facilitar la aplicación de estos operadores a vectores y matrices.

2.9 Bifurcaciones y bucles

Se van a introducir aquí los primeros conceptos de programación. MATLAB posee un lenguaje de programación que –como cualquier otro lenguaje– dispone de sentencias para realizar ***bifurcaciones*** y ***bucles***. Las ***bifurcaciones*** permiten realizar una u otra operación según se cumpla o no una determinada condición. La figura 3 muestra dos posibles formas de bifurcación. Los ***bucles*** permiten realizar las mismas o análogas operaciones sobre datos distintos. Mientras que en C el "cuerpo" de estas sentencias se determinaba mediante llaves {...}, en MATLAB se utiliza la palabra ***end*** con análoga finalidad. Existen también algunas otras diferencias de sintaxis. Hay que señalar que en MATLAB no existen las construcciones análogas a ***do ... while*** y ***switch***.

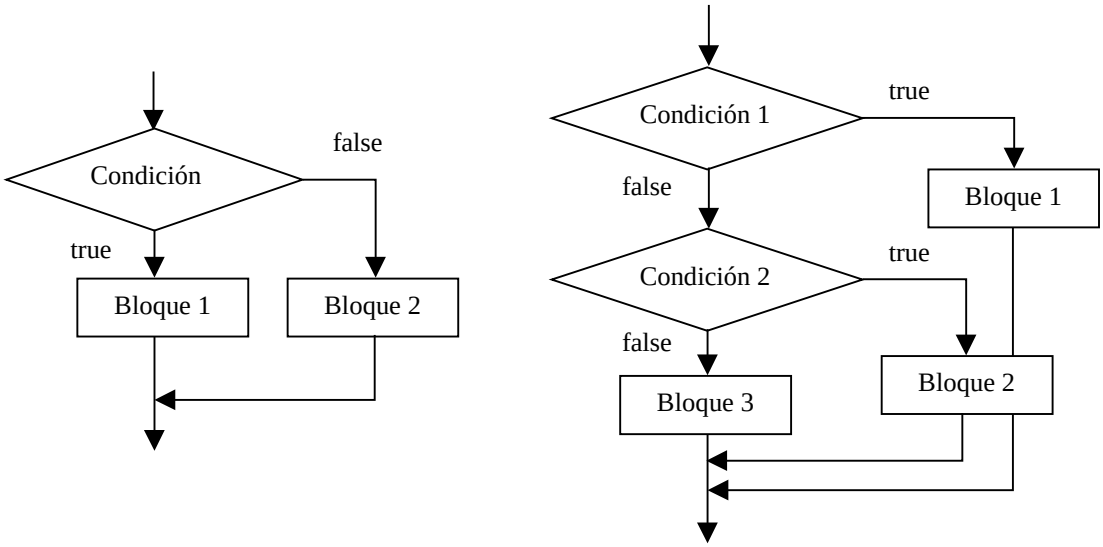


Figura 3. Ejemplos gráficos de bifurcaciones.

Las bifurcaciones y bucles no sólo son útiles en la preparación de programas o de ficheros ****.m***. También se aplican con frecuencia en el uso interactivo de MATLAB, como se verá más adelante en algunos ejemplos.

2.9.1 SENTENCIA *IF*

En su forma más simple, la sentencia ***if*** se escribe en la forma siguiente (obsérvese que –a diferencia de C– la condición no va entre paréntesis, aunque se pueden poner si se desea)[4]:

```
if condicion
     sentencias
end
```

Existe también la *bifurcación múltiple*, que tiene la forma:

[4] En los ejemplos siguientes las ***sentencias*** aparecen desplazadas hacia la derecha respecto al ***if***, ***else*** o ***end***. Esto se hace así para que el programa resulte más legible, resultando más fácil ver dónde empieza y termina la bifurcación o el bucle. Es muy recomendable seguir esta práctica de programación.

```
if condicion1
     bloque1
elseif condicion2
     bloque2
else               % opción por defecto
     bloque3
end
```

Una observación muy importante: la condición del ***if*** puede ser una *condición matricial*, del tipo **A**==**B**, donde **A** y **B** son matrices del mismo tamaño. Para que se considere que la *condición* se cumple es necesario que sean *iguales dos a dos* ***todos*** *los elementos* de las matrices **A** y **B**. Basta que haya dos elementos diferentes para que las matrices no sean iguales, y por tanto las sentencias del ***if*** no se ejecuten. Por ejemplo:

if A==B	exige que ***todos*** los elementos sean ***iguales*** dos a dos
if A~=B	exige que ***todos*** los elementos sean ***diferentes*** dos a dos

Como se ha dicho MATLAB dispone de funciones especiales para ayudar en el chequeo de condiciones matriciales.

2.9.2 SENTENCIA *FOR*

La sentencia ***for*** repite un bucle un número predeterminado de veces. La sentencia ***for*** de MATLAB es muy diferente y no tiene la generalidad de la sentencia ***for*** de C. La siguiente construcción ejecuta *sentencias* con valores de **i** de **1** a **n**, variando de uno en uno.

```
for i=1:n
     sentencias
end
```

En el siguiente ejemplo el bucle se ejecuta por primera vez con **i=n**, y luego **i** se va reduciendo en 0.2 hasta que llega a ser menor que 1, en cuyo caso el bucle se termina:

```
for i=n:-0.2:1
     sentencias
end
```

En el siguiente ejemplo se presenta una estructura correspondiente a dos bucles anidados. La variable **j** es la que varía más rápidamente (por cada valor de **i**, **j** toma todos sus posibles valores):

```
for i=1:m
     for j=1:n
          sentencias
     end
end
```

2.9.3 SENTENCIA *WHILE*

La estructura del bucle ***while*** es muy similar a la de C. Su sintaxis es la siguiente

```
while condicion
     sentencias
end
```

donde ***condicion*** puede ser una expresión vectorial o matricial. Las *sentencias* se siguen ejecutando mientras haya elementos distintos de cero en ***condicion***, es decir, mientras haya algún o algunos elementos ***true***. El bucle se termina cuando *todos los elementos* de ***condicion*** son ***false*** (es decir, cero).

2.9.4 SENTENCIA *BREAK*

Al igual que en C, la sentencia ***break*** hace que se termine la ejecución del bucle más interno de los que comprenden a dicha sentencia.

2.10 Uso del *Help*

MATLAB dispone de un buen ***Help*** con el que se puede encontrar la información que se desee. En el menú ***Help*** se tienen tres opciones: ***Table of Contents***, ***Index*** y ***Help Selected***. Con ***Table of Contents*** aparece una relación de temas sobre los que MATLAB tiene ayuda; con ***Index*** aparecen los nombres de todas las funciones disponibles, ordenadas alfabéticamente; si en la ventana de comandos se ha seleccionado algún texto, con ***Help Selected*** se obtiene la información disponible sobre ese texto seleccionado.

Además, se puede recurrir al ***Help*** desde la línea de comandos. Se aconseja hacer algunas prácticas al respecto. Por ejemplo, obsérvese la respuesta a los siguientes comandos:

```
» help
» help lang
» help matfun
» help eig
» help who, help whos
» help what, help which
```

3. Funciones de librería

MATLAB tiene un gran número de funciones incorporadas. Algunas son ***funciones intrínsecas***, esto es, funciones incorporadas en el propio código ejecutable del programa. Estas funciones son particularmente rápidas y eficientes. Existen además funciones definidas en ficheros ****.m*** y ****.mex***[5] que vienen con el propio programa o que han sido aportadas por usuarios del mismo. Estas funciones extienden en gran manera las posibilidades del programa.

3.1 Camino de búsqueda (*search path*) de MATLAB

MATLAB puede llamar a una gran variedad de funciones. A veces puede incluso haber funciones distintas que tienen el mismo nombre. Interesa saber cuáles son las reglas que determinan qué función o qué fichero ****.m*** es el que se va a ejecutar cuando su nombre aparezca en una línea de comandos del programa. Esto queda determinado por el *camino de búsqueda* (***search path***) que el programa utiliza cuando encuentra el nombre de una función.

Supóngase que se utiliza la palabra ***nombre1*** en un comando. El proceso que sigue el programa para tratar de conocer qué es ***nombre1*** es el siguiente:

1.- Comprueba si ***nombre1*** es una variable previamente definida por el usuario.

2.- Comprueba si ***nombre1*** es una función interna o intrínseca.

3.- Comprueba si hay un fichero llamado ***nombre1.mex***, ***nombre1.dll*** o ***nombre1.m*** en el directorio actual (el directorio cuyo contenido se obtiene con el comando ***» dir***).

4.- Comprueba si hay ficheros llamados ***nombre1.mex***, ***nombre1.dll*** o ***nombre1.m*** en los directorios incluidos en el *search path* de MATLAB.

[5] Los ficheros ****.mex*** son ficheros de código ejecutable.

Estos cuatro pasos se realizan por el orden indicado. En cuanto se encuentra lo que se está buscando se para la búsqueda y se utiliza el fichero que se ha encontrado. Conviene saber que, a igualdad de nombre, los ficheros ****.mex*** tienen precedencia sobre los ficheros ****.m*** que están en el mismo directorio.

El siguiente comando hace que se escriba el *search path* de MATLAB:

```
» path

         MATLABPATH

    c:\matlab\toolbox\local
    c:\matlab\toolbox\matlab\datafun
    c:\matlab\toolbox\matlab\elfun
    .... (por brevedad se omiten muchas de las líneas de salida)
    c:\matlab\toolbox\matlab\dde
    c:\matlab\toolbox\matlab\demos
    c:\matlab\toolbox\wintools
```

Para incluir un directorio nuevo al comienzo de la lista se utiliza el comando siguiente (sólo se deben utilizar directorios que realmente existan en el PC), como por ejemplo:

```
» path('c:\inf1\matlab', path)
```

o bien,

```
» path('g:\matlab', path)
```

mientras que para añadirlo al final de la lista, se utiliza:

```
» path(path, 'c:\inf1\prac9798')
```

Después de ejecutar estos comandos se puede comprobar cómo ha quedado modificado el *search path*. No es difícil borrar las líneas que se han introducido: los cambios no son permanentes y dejarán de surtir efecto al salir de MATLAB y volver a entrar. El *search path* de MATLAB está contenido en un fichero llamado ***matlabrc.m***, en el directorio principal del programa. En las instalaciones de red (como la de las salas de PCs de la Escuela), éste es un fichero que se ejecuta automáticamente al arrancar MATLAB y es un fichero controlado por el administrador del sistema. Para cambios permanentes y personalizados en el propio *search path* se debe utilizar otro fichero llamado ***startup.m***, que también se ejecuta automáticamente al arrancar el programa. Un posible contenido de este fichero puede ser el siguiente:

```
» format compact
» disp('¡Hola!')
```

3.2 Características generales de las funciones de MATLAB

El concepto de función en MATLAB es semejante al de C y al de otros lenguajes de programación, aunque con algunas diferencias importantes. Al igual que en C, una función tiene ***nombre***, ***valor de retorno*** y ***argumentos***. Una función *se llama* utilizando su nombre en una expresión o utilizándolo como un comando más. Las funciones se definen en ficheros de texto ****.m*** en la forma que se verá más adelante. Considérense los siguientes ejemplos de llamada a funciones:

```
» [maximo, posmax] = max(x);
» r = sqrt(x^2+y^2) + eps;
» a = cos(alfa) - sin(alfa);
```

donde se han utilizado algunas funciones matemáticas bien conocidas como el cálculo del valor máximo, el seno, el coseno y la raíz cuadrada. Los ***nombres*** de las funciones se han puesto en negrita. Los ***argumentos*** de cada función van a continuación del nombre entre

paréntesis (y separados por comas si hay más de uno). Los ***valores de retorno*** son el resultado de la función y sustituyen a ésta en la expresión donde la función aparece.

Una diferencia importante con otros lenguajes es que en MATLAB las funciones pueden tener valores de retorno matriciales múltiples (ya se verá que pueden recogerse en variables *ad hoc* todos o sólo parte de estos valores de retorno), como en el primero de los ejemplos anteriores. En este caso se calcula el elemento de máximo valor en un vector y se devuelven dos valores: el valor máximo y la posición que ocupa en el vector. Obsérvese que los 2 valores de retorno se recogen entre corchetes, separados por comas.

Una característica de MATLAB es que las funciones que no tienen argumentos argumentos no llevan paréntesis, por lo que a simple vista no siempre son fáciles de distinguir de las simples variables. En lo sucesivo el nombre de la función irá seguido de paréntesis si interesa resaltar que la función espera que se le pase uno o más argumentos.

MATLAB tiene diversos tipos de funciones. A continuación se enumeran los tipos de funciones más importantes, clasificadas según su finalidad:

1.- Funciones matemáticas elementales.

2.- Funciones especiales.

3.- Funciones matriciales elementales.

4.- Funciones matriciales específicas.

5.- Funciones para la descomposición y/o factorización de matrices.

6.- Funciones para análisis estadístico de datos.

7.- Funciones para análisis de polinomios.

8.- Funciones para integración de ecuaciones diferenciales ordinarias.

9.- Resolución de ecuaciones no-lineales y optimización.

10.- Integración numérica.

11.- Funciones para procesamiento de señal.

A continuación se enumeran algunas características generales de las funciones de MATLAB:

– Los *argumentos actuales*[6] de estas funciones pueden ser expresiones y también llamadas a otra función.

– MATLAB nunca modifica las variables que se pasan como argumentos. Si el usuario las modifica dentro de la función, previamente se sacan copias de esas variables (se modifican las copias, no las variables originales).

– MATLAB admite valores de retorno matriciales múltiples. Por ejemplo, en el comando:

```
[V, D] = eig(A)
```

la función ***eig()*** calcula los valores y vectores propios de la matriz cuadrada **A**. Los vectores propios se devuelven como columnas de la matriz **V**, mientras que los valores propios son los elementos de la matriz diagonal **D**. En los ejemplos siguientes:

```
[xmax, imax] = max(x)
xmax = max(x)
```

[6] Los argumentos actuales son los que se utilizan en la llamada de la función

puede verse que la misma función ***max()*** puede ser llamada recogiendo dos valores de retorno (el máximo elemento de un vector y la posición que ocupa) o un sólo valor de retorno (el máximo elemento).

– Las operaciones de suma y/o resta de una matriz con un escalar consisten en sumar y/o restar el escalar a todos los elementos de la matriz.

– Recuérdese que tecleando ***help nombre_funcion*** se obtiene de inmediato información sobre la función de ese nombre. Eligiendo ***Index*** en el menú ***Help*** aparece una relación completa de las funciones disponibles en MATLAB.

3.3 Funciones matemáticas elementales que operan de modo escalar

Estas funciones, que comprenden las funciones matemáticas trascendentales y otras funciones básicas, actúan sobre cada elemento de la matriz como si se tratase de un escalar. Se aplican de la misma forma a escalares, vectores y matrices. Algunas de las funciones de este grupo son las siguientes:

sin(x)	seno
cos(x)	coseno
tan(x)	tangente
asin(x)	arco seno
acos(x)	arco coseno
atan(x)	arco tangente (devuelve un ángulo entre $-\pi/2$ y $+\pi/2$)
atan2(x)	arco tangente (devuelve un ángulo entre $-\pi$ y $+\pi$); se le pasan 2 argumentos, proporcionales al seno y al coseno
sinh(x)	seno hiperbólico
cosh(x)	coseno hiperbólico
tanh(x)	tangente hiperbólica
asinh(x)	arco seno hiperbólico
acosh(x)	arco coseno hiperbólico
atanh(x)	arco tangente hiperbólica
log(x)	logaritmo natural
log10(x)	logaritmo decimal
exp(x)	función exponencial
sqrt(x)	raíz cuadrada
sign(x)	devuelve -1 si <0, 0 si =0 y 1 si >0. Aplicada a un número complejo, devuelve un vector unitario en la misma dirección
rem(x)	resto de la división (2 argumentos que no tienen que ser enteros)
round(x)	redondeo hacia el entero más próximo
fix(x)	redondea hacia el entero más próximo a 0
floor(x)	valor entero más próximo hacia $-\infty$
ceil(x)	valor entero más próximo hacia $+\infty$
gcd(x)	máximo común divisor
lcm(x)	mínimo común múltiplo
real(x)	partes reales
imag(x)	partes imaginarias
abs(x)	valores absolutos
angle(x)	ángulos de fase

3.4 Funciones que actúan sobre vectores

Las siguientes funciones actúan sobre vectores (no sobre matrices ni sobre escalares)

[xm,im]=max(x)	máximo elemento de un vector. Devuelve el valor máximo **xm** y la posición que ocupa **im**
min(x)	mínimo elemento de un vector. Devuelve el valor mínimo y la posición que ocupa
sum(x)	suma de los elementos de un vector
mean(x)	valor medio de los elementos de un vector
std(x)	desviación típica
prod(x)	producto de los elementos de un vector
[y,i]=sort(x)	ordenación de menor a mayor de los elementos de un vector **x**. Devuelve el vector ordenado **y**, y un vector **i** con las posiciones iniciales en **x** de los elementos en el vector ordenado **y**.

En realidad estas funciones *se pueden aplicar también a matrices*, pero en ese caso se aplican por separado a cada columna de la matriz, dando como valor de retorno un vector resultado de aplicar la función a cada columna de la matriz considerada como vector. Si estas funciones se quieren aplicar a las filas de la matriz basta aplicar dichas funciones a la matriz traspuesta.

3.5 Funciones que actúan sobre matrices

Las siguientes funciones exigen que el/los argumento/s sean matrices. En este grupo aparecen algunas de las funciones más útiles y potentes de MATLAB. Se clasificarán en varios subgrupos:

3.5.1 FUNCIONES MATRICIALES ELEMENTALES:

B = A'	calcula la traspuesta (conjugada) de la matriz **A**
B = A.'	calcula la traspuesta (sin conjugar) de la matriz **A**
v = poly(A)	devuelve un vector **v** con los coeficientes del polinomio característico de la matriz cuadrada **A**
t = trace(A)	devuelve la traza **t** (suma de los elementos de la diagonal) de una matriz cuadrada **A**
[m,n] = size(A)	devuelve el número de filas **m** y de columnas **n** de una matriz rectangular **A**
n = size(A)	devuelve el tamaño de una matriz cuadrada **A**

3.5.2 FUNCIONES MATRICIALES ESPECIALES

Las funciones ***exp()***, ***sqrt()*** y ***log()*** se aplican elemento a elemento a las matrices y/o vectores que se les pasan como argumentos. Existen otras funciones similares que tienen también sentido cuando se aplican a una matriz como una única entidad. Estas funciones son las siguientes (se distinguen porque llevan una "m" adicional en el nombre):

expm(A)	si A=XDX', expm(A) = X*diag(exp(diag(D)))*X'
sqrtm(A)	devuelve una matriz que mutiplicada por sí misma da la matriz **A**
logm()	es la función recíproca de expm(A)

Aunque no pertenece a esta familia de funciones, se puede considerar que el ***operador potencia*** (^) está emparentado con ellas. Así, es posible decir que:

A^n	está definida si **A** es cuadrada y **n** un número real. Si **n** es entero, el resultado se calcula por multiplicaciones sucesivas. Si **n** es real, el resultado se calcula como: A^n=X*D.^n*X' siendo [X,D]=eig(A)

3.5.3 FUNCIONES DE FACTORIZACIÓN Y/O DESCOMPOSICIÓN MATRICIAL

A su vez este grupo de funciones se puede subdividir en 4 subgrupos:

– Funciones basadas en la factorización triangular (eliminación de Gauss):

[L,U] = lu(A)	descomposición de Crout (A = LU) de una matriz. La matriz **L** es una permutación de una matriz triangular inferior (dicha permutación es consecuencia del pivotamiento por columnas utilizado en la factorización)
B = inv(A)	calcula la inversa de **A**. Equivale a B=inv(U)*inv(L)
d = det(A)	devuelve el determinante **d** de la matriz cuadrada **A**. Equivale a d=det(L)*det(U)
E = rref(A)	reducción a forma de escalón (mediante la eliminación de Gauss con pivotamiento por columnas) de una matriz rectangular **A**
U = chol(A)	descomposición de Cholesky de matriz simétrica y positivo-definida. Sólo se utiliza la diagonal y la parte triangular superior de **A**. El resultado es una matriz triangular superior tal que A = U'*U
c = rcond(A)	devuelve una estimación del recíproco de la condición numérica de la matriz **A** basada en la norma sub-1. Si el resultado es próximo a 1 la matriz **A** está bien condicionada; si es próximo a 0 no lo está.

– Funciones basadas en el cálculo de valores y vectores propios:

[X,D] = eig(A)	valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de una matriz cuadrada **A**. Con frecuencia el resultado es complejo (si **A** no es simétrica)
[X,D] = eig(A,B)	valores propios (diagonal de **D**) y vectores propios (columnas de **X**) de dos matrices cuadradas **A** y **B** (**Ax** = λ**Bx**).

– Funciones basadas en la descomposición QR:

[Q,R] = qr()	descomposición QR de una matriz rectangular. Se utiliza para sistemas con más ecuaciones que incógnitas.
B = null(A)	devuelve una base ortonormal del subespacio nulo (kernel, o conjunto de vectores **x** tales que **Ax** = 0) de la matriz rectangular **A**
Q = orth(A)	las columnas de **Q** son una base ortonormal del espacio de columnas de **A**. El número de columnas de **Q** es el rango de **A**

– Funciones basadas en la descomposición de valor singular

[U,D,V] = svd(A)	descomposición de valor singular de una matriz rectangular (A=U*D*V'). **U** y **V** son matrices ortonormales. **D** es diagonal y contiene los valores singulares

B = pinv(A)	calcula la pseudo-inversa de una matriz rectangular **A**
r = rank(A)	calcula el rango **r** de una matriz rectangular **A**
nor = norm(A)	calcula la norma sub-2 de una matriz (el mayor valor singular)
nor = norm(A,2)	lo mismo que la anterior
c = cond(A)	condición numérica sub-2 de la matriz **A**. Es el cociente entre el máximo y el mínimo valor singular. La condición numérica da una idea de los errores que se obtienen al resolver un sistema de ecuaciones lineales con dicha matriz: su logaritmo indica el número de cifras significativas que se pierden.

– Cálculo del rango, normas y condición numérica:

Existen varias formas de realizar estos cálculos, con distintos niveles de esfuerzo de cálculo y de precisión en el resultado.

El rango se calcula implícitamente (sin que el usuario lo pida) al ejecutar las funciones ***rref(A)***, ***orth(A)***, ***null(A)*** y ***pinv(A)***. Con ***rref(A)*** el rango se calcula como el número de filas diferentes de cero; con ***orth(A)*** y ***null(A)*** –basadas ambas en la descomposición QR– el rango es el número de columnas del resultado (o **n** menos el número de columnas del resultado). Con ***pinv(A)*** se utiliza la descomposición de valor singular, que es el método más fiable y más caro en tiempo de *cpu*. La función ***rank(A)*** está basada en ***pinv(A)***.

Normas de matrices:

norm(A)	norma sub-2, es decir, máximo valor singular de **A**, ***max(svd(A))***.
norm(A,2)	lo mismo que ***norm(A)***
norm(A,1)	norma sub-1 de **A**, máxima suma de valores absolutos por columnas, es decir: ***max(sum(abs((A))))***
norm(A,inf)	norma sub-∞ de **A**, máxima suma de valores absolutos por filas, es decir: ***max(sum(abs((A'))))***

Normas de vectores:

norm(x,p)	norma sub-p, es decir ***sum(abs(x)^p)^(1/p)***.
norm(x)	norma euclídea; equivale al módulo o ***norm(x,2)***.
norm(x,inf)	norma sub-∞, es decir ***max(abs(x))***.
norm(x,1)	norma sub-1, es decir ***sum(abs(x))***.

3.6 Más sobre operadores relacionales con vectores y matrices

Cuando alguno de los operadores relacionales vistos previamente (<, >, <=, >=, == y ~=) actúa entre dos matrices (vectores) del mismo tamaño, el resultado es otra matriz (vector) de ese mismo tamaño conteniendo unos y ceros, según los resultados de cada comparación entre elementos hayan sido ***true*** o ***false***, respectivamente.

Por ejemplo, supóngase que se define una matriz *magic* **A** de tamaño 3x3 y a continuación se forma una matriz binaria **M** basada en la condición de que los elementos de **A** sean mayores que 4 (MATLAB convierte este cuatro en una matriz de cuatros de modo automático). Obsérvese con atención el resultado:

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
```

```
» M=A>4
M =
     1     0     1
     0     1     1
     0     1     0
```

De ordinario, las matrices "binarias" que se obtienen de la aplicación de los operadores relacionales no se almacenan en memoria ni se asignan a variables, sino que se procesan sobre la marcha. MATLAB dispone de varias funciones para ello. Recuérdese que cualquier valor distinto de cero equivale a ***true***, mientras que un valor cero equivale a ***false***. Algunas de estas funciones son:

any(x)	función vectorial; chequea si *alguno* de los elementos del vector **x** cumple una determinada condición (en este caso ser distinto de cero). Devuelve un uno ó un cero
any(A)	se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros
all(x)	función vectorial; chequea si *todos* los elementos del vector **x** cumplen una condición. Devuelve un uno ó un cero
all(A)	se aplica por separado a cada columna de la matriz **A**. El resultado es un vector de unos y ceros
find(x)	busca índices correspondientes a elementos de vectores que cumplen una determinada condición. El resultado es un vector con los índices de los elementos que cumplen la condición
find(A)	cuando esta función se aplica a una matriz la considera como un vector con una columna detrás de otra, de la 1ª a la última.

A continuación se verán algunos ejemplos de utilización de estas funciones.

```
» A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
» m=find(A>4)
m =
     1
     5
     6
     7
     8
```

Ahora se van a sustituir los elementos que cumplen la condición anterior por valores de 10. Obsérvese cómo se hace y qué resultado se obtiene:

```
» A(m)=10*ones(size(m))
A =
    10     1    10
     3    10    10
     4    10     2
```

donde ha sido necesario convertir el 10 en un vector del mismo tamaño que **m**. Para chequear si hay algún elemento de un determinado valor –por ejemplo 3– puede hacerse lo siguiente:

```
» any(A==3)
ans =
     1     0     0
» any(ans)
ans =
     1
```

mientras que para comprobar que todos los elementos de **A** son mayores que cero:

```
» all(all(A))
ans =
     1
```

En este caso no ha hecho falta utilizar el operador relacional porque cualquier elemento distinto de cero equivale a ***true***.

3.7 Otras funciones que actúan sobre vectores y matrices

Las siguientes funciones pueden actuar sobre vectores y matrices, y sirven para chequear ciertas condiciones:

exist(var)	comprueba si la variable **var** existe
isnan()	chequea si hay valores ***NaN***, devolviendo una matriz de unos y ceros
isinf()	chequea si hay valores ***Inf***, devolviendo una matriz de unos y ceros
finite()	chequea si los valores son finitos
isempty()	chequea si un vector o matriz está vacío
isstr()	chequea si una variable es una cadena de caracteres (*string*)
isglobal()	chequea si una variable es global
issparse()	chequea si una matriz es dispersa (*sparse*, es decir, con un gran número de elementos cero)

A continuación se presentan algunos ejemplos de uso de estas funciones en combinación con otras vistas previamente. Se define un vector **x** con un ***NaN***, que se elimina en la forma:

```
» x=[1 2 3 4 0/0 6]
Warning: Divide by zero
x =
     1     2     3     4   NaN     6
» i=find(isnan(x))
i =
     5
» x=x(find(~isnan(x)))
x =
     1     2     3     4     6
```

Otras posibles formas de eliminarlo serían las siguientes:

```
» x=x(~isnan(x))
» x(isnan(x))=[]
```

La siguiente sentencia elimina las filas de una matriz que contienen algún ***NaN***:

```
A(any(isnan(A)'), :)=[]
```

4. Otros aspectos de MATLAB

4.1 Guardar valores de variables y estados de una sesión

En muchas ocasiones puede resultar interesante interrumpir el trabajo con MATLAB y poderlo recuperar más tarde en el mismo punto en el que se dejó (con las mismas variables definidas, con los mismos resultados intermedios, etc.). Hay que tener en cuenta que al salir del programa todo el contenido de la memoria se borra automáticamente.

Para guardar el estado de una sesión de trabajo en el directorio actual existe el comando ***save***. Si se teclea:

```
» save
```

antes de abandonar el programa, se crea un fichero binario llamado ***matlab.mat*** con el estado de la sesión (excepto los gráficos, que por ocupar mucha memoria hay que guardar aparte). Dicho estado puede recuperarse la siguiente vez que se arranque el programa con el comando:

```
» load
```

Esta es la forma más básica de los comandos ***save*** y ***load***. Se pueden guardar también matrices y vectores de forma selectiva y en ficheros con nombre especificado por el usuario. Por ejemplo, el comando:

```
» save filename A, x, y
```

guarda las variables **A**, **x** e **y** en un fichero binario llamado ***filename.mat***. Para recuperarlas en otra sesión basta teclear:

```
» load filename
```

Si no se indica ningún nombre de variable, se guardan todas las variables creadas en esa sesión.

4.2 Guardar sesión y copiar salidas

Los comandos ***save*** y ***load*** crean ficheros binarios con el estado de la sesión. Existe otra forma más sencilla de almacenar en un fichero un texto que describa lo que el programa va haciendo (la entrada y salida de los comandos utilizados). Esto se hace con el comando ***diary*** en la forma siguiente:

```
diary filename.txt
...
diary off
...
diary on
...
```

El comando ***diary off*** suspende la ejecución de ***diary*** y ***diary on*** la reanuda. El simple comando ***diary*** pasa de *on* a *off* y viceversa. Para poder acceder al fichero ***filename.txt*** con ***Notepad*** es necesario que ***diary*** esté en *off*.

4.3 Líneas de comentarios

Ya se ha indicado que para MATLAB el carácter tanto por ciento (%) indica comienzo de comentario. Cuando aparece en una línea de comandos, el programa supone que todo lo que va desde ese carácter hasta el fin de la línea es un comentario.

Más adelante se verá que los comentarios de los ficheros ****.m*** tienen algunas peculiaridades importantes, pues pueden servir para definir ***help***'s personalizados de las funciones que el usuario vaya creando.

4.4 Cadenas de texto

MATLAB puede definir variables que contengan cadenas de caracteres, aunque quizás con menos potencia y generalidad que C. En MATLAB las cadenas de texto van entre apóstrofos o comillas simples (recuérdese que en C iban entre comillas dobles: "cadena"). Por ejemplo:

```
s = 'cadena de caracteres'
```

Las cadenas de texto tienen su más clara utilidad en temas que se verán más adelante y por eso se difiere hasta entonces una explicación más detallada.

4.5 Funciones para cálculos con polinomios

Para MATLAB un polinomio se puede definir mediante un vector de coeficientes. Por ejemplo, el polinomio:

$$x^4 - 8x^2 + 6x - 10 = 0$$

se puede representar mediante el vector [1, 0, -8, 6, -10]. MATLAB puede realizar diversas operaciones sobre él, como por ejemplo evaluarlo para un determinado valor de **x** (función ***polyval()***) y calcular las raíces (función ***roots()***):

```
» pol=[1 0 -8 6 -10]
pol =
     1     0    -8     6   -10
» roots(pol)
ans =
  -3.2800
   2.6748
   0.3026 + 1.0238i
   0.3026 - 1.0238i
» polyval(pol,1)
ans =
   -11
```

Para calcular producto de polinomios MATLAB utiliza una función llamada ***conv()*** (de *producto de convolución*). En el siguiente ejemplo se va a ver cómo se multiplica un polinomio de segundo grado por otro de tercer grado:

```
» pol1=[1 -2 4]
pol1 =
     1    -2     4
» pol2=[1 0 3 -4]
pol2 =
     1     0     3    -4
» pol3=conv(pol1,pol2)
pol3 =
     1    -2     7   -10    20   -16
```

Para dividir polinomios existe otra función llamada ***deconv()***. Las funciones orientadas al cálculo con polinomios son las siguientes:

poly(A)	polinomio característico de la matriz **A**
roots(pol)	raíces del polinomio **pol**
polyval(pol,x)	evaluación del polinomio **pol** para el valor de **x**. Si **x** es un vector, **pol** se evalúa para cada elemento de **x**
polyvalm(pol,A)	evaluación del polinomio **pol** de la matriz **A**
conv(p1,p2)	producto de convolución de dos polinomios **p1** y **p2**
[c,r]=deconv(p,q)	división del polinomio **p** por el polinomio **q**. En **c** se devuelve el cociente y en **r** el resto de la división
residue(p1,p2)	descompone el cociente entre **p1** y **p2** en suma de fracciones simples (ver ***»help residue***)
polyder(pol)	calcula la derivada de un polinomio
polyder(p1,p2)	calcula la derivada de producto de polinomios

polyfit(x,y,n)	calcula los coeficientes de un polinomio **p(x)** de grado **n** que se ajusta a los datos **p(x(i)) ~= y(i)**, en el sentido de mínimo error cuadrático medio.

4.6 Medida de tiempos y de esfuerzo de cálculo

MATLAB dispone de algunas funciones con las cuales se pueden medir los tiempos de cálculo y el número de operaciones matemáticas realizadas. Estas funciones pueden ser útiles para comparar la eficiencia de algoritmos distintos que resuelven un mismo problema. Algunas de estas funciones son:

flops(0)	inicializa a cero el contador de número de operaciones aritméticas de punto flotante (flops)
flops	devuelve el número de flops realizados hasta ese momento
clock	hora actual con precisión de centésimas de segundo
etime(t2,t1)	tiempo transcurrido entre t1 y t2 (¡atención al orden!)

Por ejemplo, el siguiente código mide el tiempo necesario para resolver un sistema de 100 ecuaciones con 100 incógnitas.

```
A=rand(100); b=rand(100,1); x=zeros(100,1);
tiempo=clock; x=A\b; tiempo=etime(clock, tiempo)
```

donde se han puesto varias sentencias en la misma línea para que se ejecuten todas sin tiempos muertos al pulsar ***intro***. Esto es especialmente importante en la segunda línea de comandos, ya que es en ella en la que se quiere medir los tiempos. Todas las sentencias –excepto la última– van seguidas de punto y coma (;) con objeto de evitar la impresión de resultados.

4.7 Llamada a comandos del sistema operativo

Estando en la ventana de comandos de MATLAB, se pueden ejecutar comandos del MS-DOS precediéndolos por el carácter (!)

```
» !edit fichero1.m
```

Si el comando va seguido por el carácter ampersand (&) el comando se ejecuta en "background", es decir, se recupera el control del programa sin esperar que el comando termine de ejecutarse. Por ejemplo, para arrancar ***Notepad*** en background,

```
» !notepad &
```

Existe también la posibilidad de arrancar una aplicación y dejarla iconizada. Esto se hace postponiendo el carácter (|), como por ejemplo en el comando:

```
» !notepad |
```

Algunos comandos de MATLAB realizan la misma función que los comandos análogos del sistema operativo MS-DOS, con lo que se puede evitar utilizar el operador (!)

dir	contenido del directorio actual
what	ficheros ****.m*** en el directorio actual
delete filename	borra el fichero llamado ***filename***
type file.txt	imprime por la pantalla el contenido del fichero de texto file.txt
chdir	cambiar de directorio activo
which func	localiza una función llamada ***func***
lookfor palabra	busca ***palabra*** en todas las primeras líneas de los ficheros ****.m***

4.8 Funciones de función

En MATLAB existen funciones a las que hay que pasar como argumento otras funciones. Así sucede por ejemplo si se desea calcular la integral definida de una función, resolver una ecuación no lineal, o integrar numéricamente una ecuación diferencial ordinaria (problema de valor inicial). Estos serán los tres casos –de gran importancia práctica– que se van a ver a continuación. Se hará por medio de un ejemplo, utilizando una función llamada ***prueba*** que se va a definir en un fichero llamado ***prueba.m***.

Para definir esta función, se debe elegir ***FILE/New/M-File*** en el menú de MATLAB. Si las cosas están "en orden" se abrirá ***Notepad*** para que se pueda editar ese fichero. Si no se abre ***Notepad***, elegir en el menú el comando ***Options/Editor Preference*** y teclear ***notepad.exe*** en el cuadro de texto correspondiente. Una vez que se haya conseguido tener abierto ***Notepad***, se deben teclear las 2 líneas siguientes:

```
function y=prueba(x)
y = 1./((x-.3).^2+.01)+1./((x-.9).^2+.04)-6;
```

salvándolo después con el nombre de ***prueba.m***. Aunque la definición de funciones se verá con más detalle en un capítulo posterior, no ha habido más remedio que adelantar acontecimientos para poder explicar esto de las *funciones de función*. El fichero anterior ha definido una nueva función que puede ser utilizada como cualquier otra de las funciones de MATLAB. Antes de seguir adelante, conviene que se vea un poco el aspecto que tiene esta función que se acaba de crear. Para ello se va a dibujar la función ***prueba***. Tecléense los siguientes comandos:

```
» x=-1:0.1:2;
» plot(x,prueba(x))
```

El resultado aparece en la figura 4. Ya se está en condiciones de intentar hacer cálculos y pruebas con esta función.

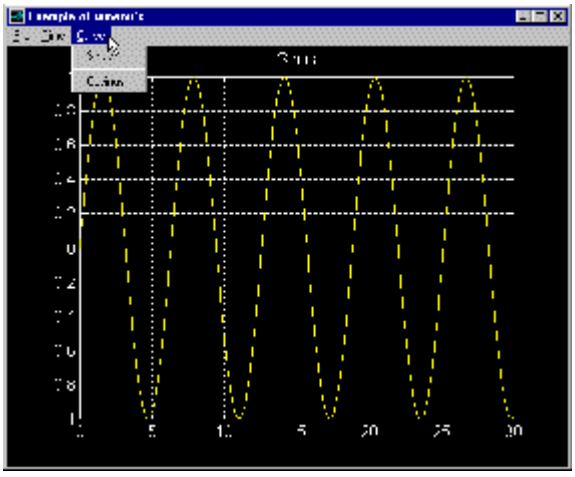


Figura 4. Función “prueba”.

4.8.1 INTEGRACIÓN NUMÉRICA

Lo primero que se va a hacer es calcular la integral definida de esta función entre dos valores de la abscisa **x**. En inglés, al cálculo numérico de integrales definidas se le llama *quadrature*. Sabiendo eso, no resulta extraño el comando con el cual se calcula el área comprendida bajo

la función entre los puntos 0 y 1 (obsérvese que el nombre de la función a integrar se pasa entre apóstrofos, como cadena de caracteres):

```
» area = quad('prueba',0,1)
area =
   29.8583
```

Si se teclea ***help quad*** se puede obtener más de información sobre esta función, incluyendo el método utilizado (*Simpson*) y la forma de controlar el error de la integración.

4.8.2 ECUACIONES NO LINEALES Y OPTIMIZACIÓN

Después de todo calcular integrales definidas no es tan difícil. Más difícil es desde luego calcular las raíces de ecuaciones no lineales y el mínimo o los mínimos de una función. MATLAB dispone de las tres funciones siguientes:

fzero	calcula un cero o una raíz de una función de una variable
fmin	calcula el mínimo de una función de una variable
fmins	calcula el mínimo de una función de varias variables

Se empezará con el cálculo de raíces. Del gráfico de la función ***prueba*** entre -1 y 2 resulta evidente que dicha función tiene dos raíces en ese intervalo. La función ***fzero*** calcula una y se conforma: ¿Cuál es la que calcula? Pues depende de un parámetro o argumento que indica un punto de partida para buscar la raíz. Véanse los siguientes comandos y resultados:

```
» fzero('prueba',-.5)
ans =
   -0.1316
» fzero('prueba',2)
ans =
    1.2995
```

En el primer caso se ha dicho al programa que empiece a buscar en el punto -0.5 y la solución encontrada ha sido -0.1316. En el segundo caso ha empezado a buscar en el punto de abscisa 2 y ha encontrado otra raíz en el punto 1.2995. Se ven claras las limitaciones de esta función.

Ahora se va a calcular el mínimo de la función ***prueba***. Defínase una función llamada ***prueba2*** que sea ***prueba*** cambiada de signo, y trátese de reproducir en su PC los siguientes comandos y resultados (para calcular máximos con ***fmin*** hay que cambiar el signo de la función):

```
» plot(x,prueba2(x))
» fmin('prueba2',-1,2)
ans =
    0.3004
» fmin('prueba2',0.5,1)
ans =
    0.8927
```

MATLAB tiene un *toolbox* o paquete especial (no disponible por el momento en la Escuela) con muchas más funciones orientadas a la *optimización*, es decir al cálculo de valores mínimos de funciones, con o sin restricciones.

4.8.3 INTEGRACIÓN NUMÉRICA DE ECUACIONES DIFERENCIALES

Este es otro campo en el que las capacidades de MATLAB pueden resultar de gran utilidad. MATLAB es capaz de calcular la evolución en el tiempo de sistemas de ecuaciones

diferenciales ordinarias de primer orden, lineales y no lineales. Se supondrá que las ecuaciones diferenciales se pueden escribir en la forma:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t) \tag{7}$$

donde *t* es la variable escalar, y tanto **y** como su derivada son vectores. Un caso típico puede ser el ***tiro parabólico***, considerando una resistencia del aire proporcional al cuadrado de la velocidad. Se supone que dicha fuerza responde a la expresión vectorial:

$$\begin{Bmatrix} F_x \\ F_y \end{Bmatrix} = -c\sqrt{\left(\dot{x}^2 + \dot{y}^2\right)} \begin{Bmatrix} \dot{x} \\ \dot{y} \end{Bmatrix} \tag{8}$$

donde *c* es una constante conocida. Las ecuaciones diferenciales del movimiento serán:

$$\begin{Bmatrix} \ddot{x} \\ \ddot{y} \end{Bmatrix} = \begin{Bmatrix} 0 \\ -g \end{Bmatrix} - \frac{c}{m}\sqrt{\left(\dot{x}^2 + \dot{y}^2\right)} \begin{Bmatrix} \dot{x} \\ \dot{y} \end{Bmatrix} \tag{9}$$

pero éste es un sistema de 2 ecuaciones diferenciales de orden 2. Para poderlo integrar debe tener la forma del sistema (7), y para ello se va a trasformar en 4 ecuaciones de primer orden, de la forma siguiente:

$$\begin{Bmatrix} \dot{u} \\ \dot{v} \\ \dot{x} \\ \dot{y} \end{Bmatrix} = \begin{Bmatrix} 0 \\ -g \\ u \\ v \end{Bmatrix} - \frac{c}{m}\sqrt{\left(u^2 + v^2\right)} \begin{Bmatrix} u \\ v \\ 0 \\ 0 \end{Bmatrix} \tag{10}$$

MATLAB dispone de dos funciones para integrar sistemas de ecuaciones diferenciales ordinarias de primer orden: ***ode23***, que utiliza el método de *Runge-Kutta* de segundo/tercer orden, y ***ode45***, que utiliza el método de *Runge-Kutta-Fehlberg* de cuarto/quinto orden. Ambas exigen al usuario escribir una función que calcule las derivadas a partir del vector de variables. Cree con ***Notepad*** un fichero llamado ***tiropar.m*** que contenga las siguientes líneas:

```
function deriv=tiropar(t,y)
fac=-(0.001/1.0)*sqrt((y(1)^2+y(2)^2));
deriv=zeros(4,1);
deriv(1)=fac*y(1);
deriv(2)=fac*y(2)-9.8;
deriv(3)=y(1);
deriv(4)=y(2);
```

donde se han supuesto unas constantes con los valores de c=0.001, m=1 y g=9.8. Falta fijar los valores iniciales de posición y velocidad. Se supondrá que el proyectil parte del origen con una velocidad de 100 m/seg y con un ángulo de 30º, lo que conduce a los valores iniciales siguientes: u(0)=100*cos(pi/6), v(0)=100*sin(pi/6), x(0)=0, y(0)=0. Los comandos para realizar la integración son los siguientes:

```
» t0=0; tf=9;
» y0=[100*cos(pi/6) 100*sin(pi/6) 0 0]';
» [t,Y]=ode23('tiropar',t0,tf,y0);
» plot(t,Y(:,4))     % dibujo de la altura en función del tiempo
```

Es muy importante que en la función ***ode23***, el vector **y0** de condiciones iniciales sea un vector columna. El vector **t** contiene los valores del tiempo para los cuales se ha calculado la posición y velocidad. Dichos valores son controlados por la función ode23 y no por el usuario, por lo que de ordinario no estarán igualmente espaciados. La matriz **Y** contiene cuatro columnas (las dos velocidades y las dos coordenadas de cada posición) y tantas filas como

elementos tiene el vector **t**. En la figura 5 se muestra el resultado del ejemplo anterior (posición vertical en función del tiempo).

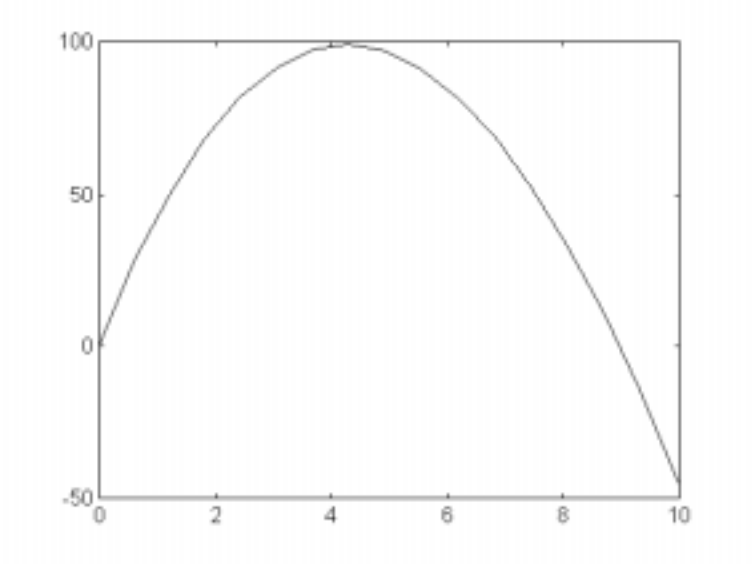


Figura 5. Tiro parabólico (posición vertical en función del tiempo).

5. Gráficos bidimensionales

A estas alturas, después de ver cómo funciona este programa, a nadie le puede resultar extraño que los gráficos 2-D de MATLAB estén fundamentalmente orientados a la representación gráfica de vectores. Los argumentos básicos de la función ***plot*** van a ser vectores. Cuando una matriz aparezca como argumento, se considerará como un conjunto de vectores columna (en algunos casos también de vectores fila).

MATLAB utiliza un tipo especial de ventanas para realizar las operaciones gráficas. Ciertos comandos abren una ventana nueva y otros dibujan sobre la ventana activa, bien sustituyendo lo que hubiera en ella, bien añadiendo nuevos elementos gráficos a un dibujo anterior. Todo esto se verá con más detalle en las siguientes secciones.

5.1 Funciones gráficas 2D elementales

MATLAB dispone de cuatro funciones básicas para crear gráficos 2-D. Estas funciones se diferencian principalmente por el *tipo de escala* que utilizan en los ejes de abscisas y de ordenadas. Estas cuatro funciones son las siguientes:

plot()	crea un gráfico a partir de vectores y/o columnas de matrices, con escalas lineales sobre ambos ejes.
loglog()	idem con escala logarítmica en ambos ejes
semilogx()	idem con escala lineal en el eje de ordenadas y logarítmica en el eje de abscisas
semilogy()	idem con escala lineal en el eje de abscisas y logarítmica en el eje de ordenadas

En lo sucesivo se hará referencia casi exclusiva a la primera de estas funciones (***plot***). Las demás se pueden utilizar de un modo similar.

Existen además otras funciones orientadas a añadir títulos al gráfico, a cada uno de los ejes, a dibujar una cuadrícula auxiliar, a introducir texto, etc. Estas funciones son las siguientes:

title('título')	añade un título al dibujo
xlabel('tal')	añade una etiqueta al eje de abscisas. Con ***xlabel off*** desaparece
ylabel('cual')	añade una etiqueta al eje de ordenadas. Con ***ylabel off*** desaparece
text(x,y,'texto')	introduce 'texto' en el lugar especificado por las coordenadas **x** e **y**. Si **x** e **y** son vectores, el texto se repite por cada par de elementos. Si **texto** es también un vector de cadenas de texto de la misma dimensión, cada elemento se escribe en las coordenadas correspondientes
gtext('texto')	introduce **texto** con ayuda del ratón: el cursor cambia de forma y se espera un clic para introducir el texto en esa posición
grid	activa la inclusión de una cuadrícula en el dibujo. Con ***grid off*** desaparece la cuadrícula

Borrar texto (u otros elementos gráficos) es un poco más complicado; de hecho, hay que preverlo de antemano. Para poder hacerlo hay que recuperar previamente el *valor de retorno* del comando con el cual se ha creado. Después hay que llamar a la función ***delete*** con ese valor como argumento. Considérese el siguiente ejemplo:

```
» v = text(1,.0,'seno')
v =
   76.0001
» delete(v)
```

Los dos grupos de funciones anteriores no actúan de la misma forma. Así, la función ***plot*** dibuja una nueva figura en la ventana activa (en todo momento MATLAB tiene una ventana activa de entre todos las ventanas gráficas abiertas), o abre una nueva figura si no hay ninguna abierta, sustituyendo cualquier cosa que hubiera dibujada anteriormente en esa ventana. Para verlo, se comenzará creando un par de vectores **x** e **y** con los que trabajar:

```
» x=[-10:0.2:10]; y=sin(x);
```

Ahora se deben ejecutar los comandos siguientes (se comienza cerrando la ventana activa, para que al crear la nueva ventana aparezca en primer plano):

```
» close
» grid
» plot(x,y)
```

Se puede observar la diferencia con la secuencia que sigue:

```
» close
» plot(x,y)
» grid
```

En el primer caso MATLAB ha creado la cuadrícula en una ventana nueva y luego la ha borrado al ejecutar la función ***plot***. En el segundo caso, primero ha dibujado la función y luego ha añadido la cuadrícula. Esto es así porque hay funciones como ***plot*** que por defecto crean una nueva figura, y otras funciones como ***grid*** que se aplican a la ventana activa modificándola, y sólo crean una ventana nueva cuando no existe ninguna ya creada. Más adelante se verá que con la función ***hold*** pueden añadirse gráficos a una figura ya existente respetando su contenido.

5.1.1 FUNCIÓN *PLOT*

Esta es la función clave de todos los gráficos 2-D en MATLAB. Ya se ha dicho que el elemento básico de los gráficos bidimensionales es el **vector**. Se utilizan también cadenas de 1, 2 ó 3 caracteres para indicar *colores* y *tipos de línea*. La función ***plot()***, en sus diversas variantes, no hace otra cosa que dibujar vectores. Un ejemplo muy sencillo de esta función, en el que se le pasa un único vector como argumento, es el siguiente:

```
» x=[1 3 2 4 5 3]
x =
     1     3     2     4     5     3
» plot(x)
```

El resultado de este comando es que se abre una ventana mostrando un gráfico similar al de la figura 6. Por defecto, los distintos puntos del gráfico se unen con una línea continua. También por defecto, el color que se utiliza para la primera línea es el amarillo.

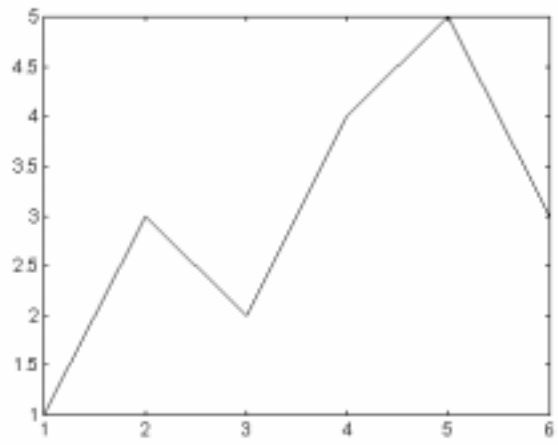


Figura 6. Gráfico 2-D correspondiente al vector x=[1 3 2 4 5 3].

Cuando a la función ***plot()*** se le pasa un único vector –real– como argumento, dibuja en ordenadas el valor de los **n** elementos del vector frente a los índices 1, 2, ... **n** del mismo en abscisas. Más adelante se verá que si el vector es complejo, el funcionamiento es bastante diferente.

En la pantalla de su ordenador se habrá visto (y en la figura 1 quedó reflejado) que MATLAB utiliza por defecto color negro para el fondo de la pantalla y otros colores más claros para los ejes y las gráficas. Estos colores no son los más adecuados para una impresora láser o de chorro de tinta, y se pueden invertir cuando una figura se envía a la impresora o se guarda en un fichero.

Una segunda forma de utilizar la función ***plot()*** es con dos vectores como argumentos. En este caso los elementos del segundo vector se representan en ordenadas frente a los valores del primero, que se representan en abscisas. Véase por ejemplo cómo se puede dibujar un cuadrilátero de esta forma (obsérvese que para dibujar un polígono cerrado el último punto debe coincidir con el primero):

```
» x=[1 6 5 2 1]; y=[1 0 4 3 1];
» plot(x,y)
```

La función ***plot()*** permite también dibujar múltiples curvas introduciendo varias parejas de vectores como argumentos. En este caso, cada uno de los segundos vectores se dibujan en

ordenadas como función de los valores del primer vector de la pareja, que se representan en abscisas. Obsérvese bien cómo se dibujan el seno y el coseno en el siguiente ejemplo:

```
» x=0:pi/25:6*pi;
» y=sin(x); z=cos(x);
» plot(x,y,x,z)
```

Ahora se va a ver lo que pasa con los **vectores complejos**. Si se pasan a ***plot()*** varios vectores complejos como argumentos, MATLAB simplemente representa las partes reales y desprecia las partes imaginarias. Sin embargo, un único argumento complejo hace que se represente la parte real en abscisas, frente a la parte imaginaria en ordenadas. Véase el siguiente ejemplo. Para generar un vector complejo se utilizará el resultado del cálculo de valores propios de una matriz formada aleatoriamente:

```
» plot(eig(rand(20,20)),'+')
```

donde se ha hecho uso de elementos que se verán en la siguiente sección, respecto a dibujar con distintos tipos de "markers" (en este caso con signos +), en vez de con línea continua, que es la opción por defecto. En el comando anterior, el segundo argumento es un carácter que indica el tipo de marker elegido. El comando anterior es equivalente a:

```
» z=eig(rand(20,20));
» plot(real(z),imag(z),'+')
```

Como ya se ha dicho, si se incluye más de un vector complejo como argumento, se ignoran las partes imaginarias. Si se quiere dibujar varios vectores complejos, hay que separar explícitamente las partes reales e imaginarias de cada vector, como se acaba de hacer en el último ejemplo.

El comando ***plot*** puede utilizarse también con matrices como argumentos. Véanse algunos ejemplos sencillos:

plot(A)	dibuja una línea por cada columna de A en ordenadas, frente al índice de los elementos en abscisas
plot(x,A)	dibuja las columnas (o filas) de **A** en ordenadas frente al vector **x** en abscisas. Las dimensiones de **A** y **x** deben ser coherentes: si la matriz **A** es cuadrada se dibujan las columnas, pero si no lo es y la dimensión de las filas coincide con la de **x**, se dibujan las filas
plot(A,x)	análogo al anterior, pero dibujando las columnas (o filas) de **A** en abscisas, frente al valor de **x** en ordenadas
plot(A,B)	dibuja las columnas de **B** en ordenadas frente a las columnas de **A** en abscisas, dos a dos. Las dimensiones deben coincidir
plot(A,B,C,D)	análogo al anterior para cada par de matrices. Las dimensiones de cada par deben coincidir, aunque pueden ser diferentes de las dimensiones de los demás pares

Se puede obtener una excelente y breve descripción de la función ***plot()*** con el comando ***help plot***. La descripción que se acaba de presentar se completará en la siguiente sección, en donde se verá cómo elegir los colores y los tipos de línea.

5.1.2 ESTILOS DE LÍNEA Y MARCADORES EN LA FUNCIÓN *PLOT*

En la sección anterior se ha visto cómo la tarea fundamental de la función ***plot()*** era dibujar los valores de un vector en ordenadas, frente a los valores de otro vector en abscisas. En el caso general esto exige que se pasen como argumentos un par de vectores. En realidad, el conjunto básico de argumentos de esta función es una *tripleta* formada por dos vectores y una

cadena de 1, 2 ó 3 caracteres que indica el color y el tipo de línea o de marker. En la tabla siguiente se pueden observar las distintas posibilidades.

Símbolo	Color	Símbolo	Estilo de línea
y	yellow	.	puntos
m	magenta	o	círculos
c	cyan	x	marcas en x
r	red	+	marcas en +
g	green	*	marcas en *
b	blue	-	líneas continuas
w	white	:	líneas a puntos
k	black	-.	líneas a barra-punto
		--	líneas a trazos

Tabla 1. Colores, markers y estilos de línea.

Cuando hay que dibujar varias líneas, por defecto se van cogiendo sucesivamente los 8 colores de la tabla, y cuando se terminan se vuelve a empezar otra vez por el amarillo.

5.1.3 AÑADIR LÍNEAS A UN GRÁFICO YA EXISTENTE

Existe la posibilidad de añadir líneas a un gráfico ya existente, sin destruirlo o sin abrir una nueva ventana. Se utilizan para ello los comandos ***hold on*** y ***hold off***. El primero de ellos hace que los gráficos sucesivos respeten los que ya se han dibujado en la figura (es posible que haya que modificar la escala de los ejes); el comando ***hold off*** deshace el efecto de ***hold on***. El siguiente ejemplo muestra cómo se añaden las gráficas de **x2** y **x3** a la gráfica de **x** previamente creada (cada una con un tipo de línea diferente):

```
plot(x)
hold on
plot(x2,'--')
plot(x3,'-.')
hold off
```

5.1.4 COMANDO *SUBPLOT*

Una ventana gráfica se puede dividir en **m** particiones horizontales y **n** verticales, con objeto de representar múltiples gráficos en ella. Cada una de estas subventanas tiene sus propios ejes, aunque otras propiedades son comunes a toda la figura. La forma general de este comando es:

```
subplot(m,n,i)
```

donde **m** y **n** son el número de subdivisiones en filas y columnas, e **i** es la subdivisión que se convierte en activa. Las subdiviones se numeran consecutivamente empezando por las de la primera fila, siguiendo por las de la segunda, etc. Por ejemplo, la siguiente secuencia de comandos genera cuatro gráficos en la misma ventana:

```
» y=sin(x); z=cos(x); w=exp(-x*.1).*y; v=y.*z;
» subplot(2,2,1), plot(x,y)
» subplot(2,2,2), plot(x,z)
» subplot(2,2,3), plot(x,w)
» subplot(2,2,4), plot(x,v)
```

Se puede practicar con este ejemplo añadiendo títulos a cada *subplot*, así como rótulos para los ejes. Se puede intentar también cambiar los tipos de línea. Para volver a la opción por defecto basta teclear el comando:

```
» subplot(1,1,1)
```

5.1.5 CONTROL DE LOS EJES

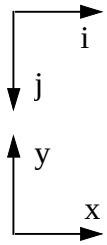
También en este punto MATLAB tiene sus opciones por defecto, que en algunas ocasiones puede interesar cambiar. El comando básico es el comando ***axis***. Por defecto, MATLAB ajusta la escala de cada uno de los ejes de modo que varíe entre el mínimo y el máximo valor de los vectores a representar. Este es el llamado modo "auto", o modo automático. Para definir de modo explícito los valores máximo y mínimo según cada eje, se utiliza el comando:

```
axis([xmin, xmax, ymin, ymax])
```

mientras que :

```
axis('auto')
```

devuelve el escalado de los ejes al valor por defecto o automático. Otros posibles usos de este comando son los siguientes:



v=axis	devuelve un vector **v** con los valores [xmin, xmax, ymin, ymax]
axis(axis)	mantiene los ejes en sus actuales valores, de cara a posibles nuevas gráficas añadidas con ***hold on***
axis('ij')	utiliza *ejes de pantalla*, con el origen en la esquina superior izda. y el eje **j** en dirección vertical descendente
axis('xy')	utiliza *ejes cartesianos* normales, con el origen en la esquina inferior izda. y el eje **y** vertical ascendente
axis('equal')	el escalado es igual en ambos ejes
axis('square')	la ventana será cuadrada
axis('image')	la ventana tendrá las proporciones de la imagen que se desea representar en ella (por ejemplo la de una imagen bitmap que se desee importar) y el escalado de los ejes será coherente con dicha imagen
axis('normal')	elimina las restricciones introducidas por 'equal' y 'square'
axis('off')	elimina las etiquetas, los números y los ejes
axis('on')	restituye las etiquetas, los números y los ejes

5.2 Control de ventanas gráficas: función *figure*

Si se llama a la función ***figure*** sin argumentos, se crea una nueva ventana gráfica con el número consecutivo que le corresponda. El valor de retorno es dicho número.

Por otra parte, el comando ***figure(n)*** hace que la ventana **n** pase a ser la ventana o figura activa. Si dicha ventana no existe, se crea una nueva ventana con el número consecutivo que le corresponda (que se puede obtener como valor de retorno del comando). La función ***close*** cierra la figura activa, mientras que ***close(n)*** cierra la ventana o figura número **n**.

El comando ***clf*** elimina el contenido de la figura activa, es decir, la deja abierta pero vacía. La función ***gcf*** devuelve el número de la figura activa en ese momento.

Para practicar un poco con todo lo que se acaba de explicar, ejecútense las siguientes instrucciones de MATLAB, observando con cuidado los efectos de cada una de ellas en la

ventana activa. El comando ***figure(gcf)*** (*get current figure*) permite hacer visible la ventana de gráficos desde la ventana de comandos.

```
» x=[-4*pi:pi/20:4*pi];
» plot(x,sin(x),'r',x,cos(x),'g')
» title('Función seno(x) -en rojo- y función coseno(x) -en verde-')
» xlabel('ángulo en radianes'), figure(gcf)
» ylabel('valor de la función trigonométrica'), figure(gcf)
» axis([-12,12,-1.5,1.5]), figure(gcf)
» axis('equal'), figure(gcf)
» axis('normal'), figure(gcf)
» axis('square'), figure(gcf)
» axis('off'), figure(gcf)
» axis('on'), figure(gcf)
» axis('on'), grid, figure(gcf)
```

5.3 Otras funciones gráficas 2-D

Existen otras funciones gráficas bidimensionales orientadas a generar otro tipo de gráficos distintos de los que produce la función ***plot()*** y sus análogas. Algunas de estas funciones son las siguientes (para más información sobre cada una de ellas en particular, utilizar ***help nombre_función***):

bar()	crea diagramas de barras
stairs()	función análoga a ***bar()*** sin líneas internas
errorbar()	representa sobre una gráfica –mediante barras– valores de errores
compass()	dibuja los elementos de un vector complejo como un conjunto de vectores partiendo de un origen común
feather()	dibuja los elementos de un vector complejo como un conjunto de vectores partiendo de orígenes uniformemente espaciados sobre el eje de abscisas
hist()	dibuja histogramas de un vector
rose()	histograma de ángulos (en radianes)
quiver()	dibujo de campos vectoriales como conjunto de vectores

A modo de ejemplo, genérese un vector de valores aleatorios entre 0 y 10, y ejecútense los siguientes comandos:

```
» x=[rand(1,100)*10];
» plot(x)
» bar(x)
» stairs(x)
» hist(x)
» hist(x,20)
» alfa=(rand(1,20)-0.5)*2*pi;
» rose(alfa)
```

5.3.1 FUNCIÓN *FPLOT*

La función ***plot*** vista anteriormente dibuja vectores. Si se quiere dibujar una función, antes de ser pasada a ***plot*** debe ser convertida en un vector de valores. Esto tiene algunos inconvenientes, por ejemplo, el que "a priori" es difícil predecir en que zonas la función varía más rápidamente y habría por ello que reducir el espaciado entre los valores en el eje de abscisas.

La función ***fplot*** admite como argumento un *nombre de función* o un *nombre de fichero* ****.m*** en el cual esté definida una función de usuario. La función puede ser escalar (un único resultado por cada valor de **x**) o vectorial. La forma general de esta función es la siguiente:

```
fplot('funcion',limites,'cadena',tol)
```

donde:

'funcion' representa el nombre de la función o del fichero ****.m*** entre apóstrofos (pasado como cadena de caracteres),

limites es un vector de 2 ó 4 elementos que puede tomar los valores [xmin,xmax] o [xmin,xmax,ymin,ymax],

'cadena' tiene el mismo significado que en ***plot*** y permite controlar el color, los markers y el tipo de línea. Además de las capacidades standard de ***plot***, ***fplot*** acepta '-+', '-x', '-o', '-*' (ó '+-', 'x-', 'o-', y '*-').

tol es la tolerancia de error relativo. El valor por defecto es 2e-03. El máximo número de valores en **x** es (1/tol)+1

Esta función puede utilizarse también en la forma:

```
[x,y]=fplot('funcion', limites,'cadena',tol)
```

y en este caso se devuelven los vectores **x** e **y**, pero no se dibuja nada. El gráfico puede obtenerse con un comando posterior por medio de la función ***plot***. Véase un ejemplo de utilización de esta función. Se comienza creando un fichero llamado ***mifunc.m*** en el directorio ***G:\matlab*** que contenga las líneas siguientes:

```
function y = mifunc(x)
y(:,1)=200*sin(x)./x;
y(:,2)=x.^2;
```

y a continuación se ejecuta el comando:

```
fplot('mifunc(x)',[-20 20],'g')
```

Obsérvese que la función ***mifunc*** devuelve una matriz con dos columnas, que constituyen las dos gráficas dibujadas. En este caso se ha utilizado para ellas el color verde.

5.3.2 FUNCIÓN *FILL* PARA POLÍGONOS

Ésta es una función especial para dibujar polígonos planos, rellenándolos de un determinado color. La forma general es la siguiente:

```
fill(x,y,c)
```

que dibuja un polígono definido por los vectores **x** e **y**, rellenándolo con el color especificado por **c**. Si es necesario, el polígono se cierra uniendo el último vértice con el primero.

Si **c** es un carácter de color ('r','g','b','c','m','y','w','k'), o un vector de valores [r g b], el polígono se rellena de modo uniforme con el color especificado.

Si **c** es un vector de la misma dimensión que **x** e **y**, sus elementos se trasforman de acuerdo con un mapa de colores determinado, y el llenado del polígono –no uniforme en este caso– se obtiene interpolando entre los colores de los vértices. Sobre este tema de los colores, se volverá más adelante con un cierto detenimiento.

Este comando puede utilizarse también con matrices:

```
fill(A,B,C)
```

donde **A** y **B** son matrices del mismo tamaño. En este caso se dibuja un polígono por cada par de columnas de dichas matrices. **C** puede ser un vector fila de colores uniformes para cada

polígono, o una matriz del mismo tamaño que las anteriores para obtener colores de relleno por interpolación. Si una de las dos, o **A** o **B**, son un vector en vez de una matriz, se supone que ese vector se repite tantas veces como sea necesario para dibujar tantos polígonos como columnas tiene la matriz. Considérese un ejemplo sencillo de esta función:

```
» x=[1 5 4 2]; y=[1 0 4 3];
» fill(x,y,'r')
» colormap(gray), fill(x,y,[1 0.5 0.8 0.7])
```

5.4 Entrada de puntos con el ratón

Se realiza mediante la función ***ginput***, que permite introducir las coordenadas del punto sobre el que está el cursor, al clicar (o al pulsar una tecla). Algunas formas de utlizar esta función son las siguientes:

[x,y] = ginput	lee un número indefinido de puntos –cada vez que se clica o se pulsa una tecla cualquiera– hasta que se termina pulsando la tecla ***intro***
[x,y] = ginput(n)	lee las coordenadas de **n** puntos
[x,y,bot] = ginput	igual que el anterior, pero devuelve también un vector de enteros **bot** con el código ASCII de la tecla pulsada o el número del botón del ratón (1, 2, ...) con el que se ha clicado

Como ejemplo de utilización de este comando, ejecútense las instrucciones siguientes en la ventana de comandos de MATLAB para introducir un cuadrilátero arbitrario y dibujarlo de dos formas:

```
» clf, [x,y]=ginput(4);
» figure(gcf), plot(x,y,'w'), pause(5), fill(x,y,'r')
```

donde se ha introducido el comando ***pause(5)*** que espera 5 segundos antes de pasar a ejecutar el siguiente comando.

5.5 Preparación de películas o "movies"

Para preparar pequeñas películas o movies se pueden utilizar las funciones ***movie***, ***moviein*** y ***getframe***. Una película se compone de varias imágenes, denominadas *frames*. La función ***getframe*** devuelve un vector columna con la información necesaria para reproducir la imagen que se acaba de representar en la figura o ventana gráfica activa, por ejemplo con la función ***plot***. El tamaño de este vector columna depende del tamaño de la ventana, pero no de la complejidad del dibujo. La función ***moviein(n)*** reserva memoria para almacenar **n** frames. La siguiente lista de comandos crearía una película de 17 imágenes o frames, que se almacenarán como las columnas de la matriz **M**:

```
M = moviein(17);
x=[-2*pi:0.1:2*pi]';
for i=1:pi/8:2*pi
    y=sin(x+i);
    plot(x,y);
    M(:,i) = getframe;
end
```

Una vez creada la película se puede representar el número de veces que se desee con el comando ***movie***. Por ejemplo, para representar 10 veces la película anterior a 15 imágenes por segundo habría que ejecutar el comando siguiente (los dos últimos parámetros son opcionales):

```
movie(M,10,15)
```

Los comandos ***moviein***, ***getframe*** y ***movie*** tienen posibilidades adicionales para las que puede consultarse el ***Help*** correspondiente. Hay que señalar que en MATLAB no es lo mismo un *movie* que una *animación*. Una *animación* es simplemente una ventana gráfica que va cambiando como consecuencia de los comandos que se van ejecutando. Un *movie* es una animación grabada o almacenada en memoria previamente.

5.6 Impresión de las figuras en impresora láser

Es relativamente fácil enviar a la impresora o a un fichero una figura producida con MATLAB. Por defecto, MATLAB produce salidas tipo *postcript* (un formato de descripción de páginas propio de impresoras de la gama alta). Esta salida se dirige por defecto a una impresora *postcript* –si está disponible– o a un fichero de disco, desde donde podrá ser enviado más tarde a la impresora o trasmitido a otro ordenador. Si no hay ninguna impresora *postcript* disponible, MATLAB puede trasformar la salida y convertirla al formato de la impresora disponible en ese momento (está muy extendido el formato *HPGL*, propio de las impresoras láser de Hewlett-Packard de la gama media-baja).

Existen varios tipos de formato *postcript*. El nivel 2 es más reciente y capaz que el nivel 1, pero no todas las impresoras están preparadas para él. Existe otra variante llamada *encapsulated postcript*, destinada a producir figuras que van a ser insertadas en otros documentos. Finalmente, hay un *postcript* para impresoras en blanco y negro –con diversas tonalidades de gris– y otro para impresoras en color.

La impresión de una figura puede hacerse desde menú o mediante un comando. La forma general del comando de impresión es la siguiente (si se omite el nombre del fichero, la figura se envía a la impresora):

```
» print filename -tipo
```

donde *tipo* puede tomar los siguientes valores:

dps	postcript nivel 1 en blanco y negro
dpsc	postcript nivel 1 en color
dps2	postcript nivel 2 en blanco y negro
dpsc2	postcript nivel 1 en color
deps	encapsulated postcript nivel 1 en blanco y negro
depsc	encapsulated postcript nivel 1 en color
deps2	encapsulated postcript nivel 2 en blanco y negro
depsc2	encapsulated postcript nivel 1 en color

Si la figura es en color y se envía a una impresora en blanco y negro, se realiza una conversión de colores a tonalidades de gris. En este caso, puede sin embargo ser más adecuado el realizar la figura con un mapa de colores que se ajuste directamente a las distintas tonalidades de gris.

Cuando se imprime en blanco y negro una figura, se suele invertir por defecto el color del fondo (negro) y el de los ejes y rótulos. Si se desea evitarlo, y que se imprima tal como se ve en la pantalla, se tendría que ejecutar previamente el siguiente comando:

```
set(gcf,'InvertHardCopy','off')
```

donde el comando ***set*** permite cambiar las propiedades gráficas de los distintos objetos de MATLAB. Para más información utilizar el ***Help***.

6. Gráficos tridimensionales

Quizás sea ésta una de las características de MATLAB que más admiración despierta entre los usuarios no técnicos (cualquier alumno de ingeniería sabe que hay ciertas operaciones algebraicas –como la descomposición de valor singular, sin ir más lejos– que tienen dificultades muy superiores, aunque "luzcan" menos).

6.1 Tipos de funciones gráficas tridimensionales

MATLAB tiene posibilidades de realizar varios tipos de gráficos 3D. Para darse una idea de ello, lo mejor es verlo en la pantalla cuanto antes, aunque haya que dejar las explicaciones detalladas para un poco más adelante.

La primera forma de gráfico 3D es la función ***plot3***, que es el análogo tridimensional de la función ***plot***. Esta función dibuja puntos cuyas coordenadas están contenidas en 3 vectores, bien uniéndolos mediante una línea continua (defecto), bien mediante markers. Asegúrese de que no hay ninguna ventana gráfica abierta y ejecute el siguiente comando que dibuja una línea espiral:

```
» fi=[0:pi/20:6*pi]; plot3(cos(fi),sin(fi),fi,'g')
```

Ahora se verá cómo se representa una función de dos variables. Para ello se va a definir una función de este tipo en un fichero llamado ***test3d.m***. La fórmula será la siguiente:

$$z = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

El fichero ***test3d.m*** debe contener las líneas siguientes:

```
function z=test3d(x,y)
z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

Ahora, ejecútese la siguiente lista de comandos (directamente, o mejor creando un fichero ***test3dFC.m*** que los contenga):

```
» x=[-3:0.4:3]; y=x;
» close
» subplot(2,2,1)
» figure(gcf),fi=[0:pi/20:6*pi]; plot3(cos(fi),sin(fi),fi,'g')
» [X,Y]=meshgrid(x,y);
» Z=test3d(X,Y);
» subplot(2,2,2)
» figure(gcf), mesh(Z)
» subplot(2,2,3)
» figure(gcf), surf(Z)
» subplot(2,2,4)
» figure(gcf), contour3(Z,16)
```

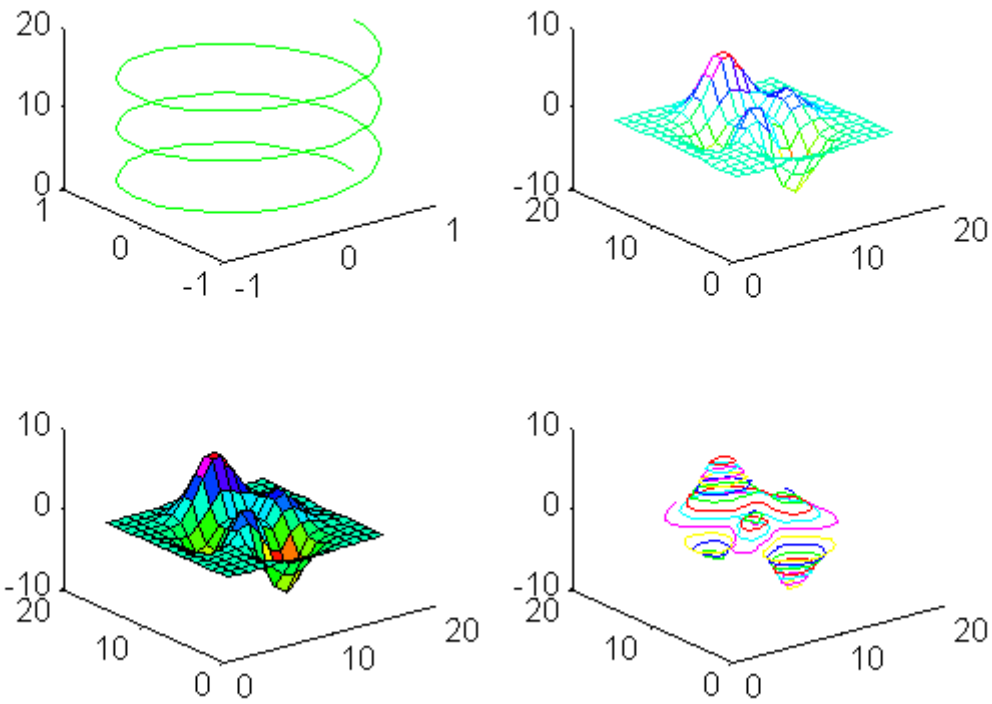


Figura 7. Gráficos 3D realizados con MATLAB

En la figura resultante (figura 7) aparece una buena muestra de algunas de las posibilidades gráficas tridimensionales de MATLAB. En las próximas secciones se encontrará la explicación de qué se ha hecho y cómo se ha hecho.

6.1.1 DIBUJO DE LÍNEAS: FUNCIÓN *PLOT3*

La función ***plot3*** es análoga a su homóloga bidimensional ***plot***. Su forma más sencilla es la siguiente:

```
plot3(x,y,z)
```

que dibuja una línea que une los puntos (x(1), y(1), z(1)), (x(2), y(2), z(2)), etc. y la proyecta sobre un plano para poderla representar en la pantalla. Al igual que en el caso plano, se puede incluir una cadena de 1, 2 ó 3 caracteres para determinar el color, los markers, y el tipo de línea:

```
plot3(x,y,z,s)
```

También se pueden utilizar tres matrices **X**, **Y** y **Z** del mismo tamaño:

```
plot3(X,Y,Z)
```

en cuyo caso se dibujan tantas líneas como columnas tienen estas 3 matrices, cada una de las cuales está definida por las 3 columnas homólogas de dichas matrices.

A continuación se va a realizar un ejemplo sencillo consistente en dibujar un *cubo*. Para ello se creará un fichero llamado ***cubo.m*** que contenga las aristas correspondientes, definidas mediante los vértices del cubo como una línea poligonal continua (obsérvese que algunas aristas se dibujan dos veces). El fichero ***cubo.m*** define una matriz **A** cuyas columnas son las coordenadas de los vértices, y cuyas filas son las coordenadas **x**, **y** y **z** de los mismos:

```
A=[0 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0
   0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1 1
   0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0]
```

Ahora basta ejecutar los comandos siguientes (el trasponer los vectores en este caso es opcional):

```
» cubo;
» plot3(A(1,:)',A(2,:)',A(3,:)')
```

6.1.2 DIBUJO DE MALLADOS: FUNCIONES MESHGRID, MESH Y SURF

Ahora se verá con detalle cómo se puede dibujar una función de dos variables (z=f(x,y)) sobre un dominio rectangular. Se verá que también se pueden dibujar los elementos de una matriz como función de los dos índices.

Sean **x** e **y** dos vectores que contienen las coordenadas en una y otra dirección de la retícula (*grid*) sobre la que se va a dibujar la función. Después hay que crear dos matrices **X** (cuyas filas son copias de **x**) e **Y** (cuyas columnas son copias de **y**). Estas matrices se crean con la función ***meshgrid***. Estas matrices representan respectivamente las coordenadas *x* e *y* de todos los puntos de la retícula. La matriz de valores **Z** se calcula a partir de las matrices de coordenadas **X** e **Y**. Finalmente hay que dibujar esta matriz **Z** con la función ***mesh***, cuyos elementos son función elemento a elemento de los elementos de **X** e **Y**. Véase como ejemplo el dibujo de la función ***sen(r)/r*** (siendo $r=sqrt(x^2+y^2)$; para evitar dividir por 0 se suma el número pequeño ***eps***). Para distinguirla de la función ***test3d*** anterior se utilizará **u** y **v** en lugar de **x** e **y**. Créese un fichero llamado ***sombrero.c*** que contenga las siguientes líneas:

```
» u=-8:0.5:8; v=u;
» [U,V]=meshgrid(u,v);
» R=sqrt(U.^2+V.^2)+eps;
» W=sin(R)./R;
» mesh(W)
```

Ejecutando este fichero se obtiene el gráfico mostrado en la figura 8.

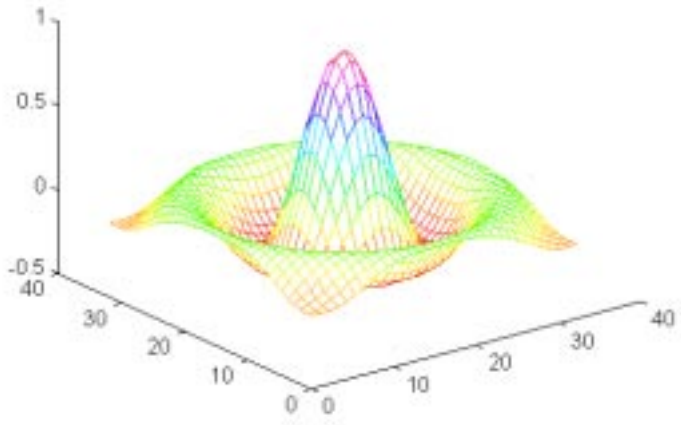


Figura 8. Representación 3D de la función “sombrero”.

Se habrá podido comprobar que la función ***mesh*** dibuja *en perspectiva* una función en base a una retícula de líneas de colores, rodeando cuadriláteros del color de fondo, con

eliminación de líneas ocultas. Más adelante se verá cómo controlar estos colores que aparecen. Baste decir por ahora que el color depende del valor **z** de la función. Ejecútese ahora el comando:

```
» surf(W)
```

y obsérvese la diferencia en la figura 9. En vez de líneas aparece ahora una superficie *faceteada*, también con eliminación de líneas ocultas. El color de las facetas depende también del valor de la función.

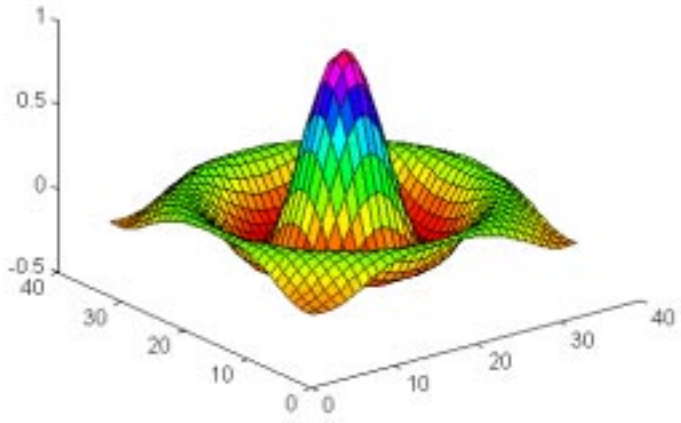


Figura 9. Función “sombrero” con facetas.

Como un segundo ejemplo, se va a volver a dibujar la función ***picos*** (la correspondiente al fichero ***test3d.m*** visto previamente). Créese ahora el fichero ***picos.m*** con las siguientes sentencias:

```
x=[-3:0.2:3];
y=x;
[X,Y]=meshgrid(x,y);
Z=test3d(X,Y);
figure(gcf), mesh(Z), pause(5), surf(Z)
```

Es necesario poner la instrucción ***pause*** –que espera 5 segundos– para que se puedan ver las dos formas de representar la función **Z** (si no, sólo se vería la segunda). Una vez creado este fichero, tecléese ***picos*** en la línea de comandos y obsérvese el resultado. Más adelante se verá también cómo controlar el punto de vista en estos gráficos en perspectiva.

6.1.3 DIBUJO DE LÍNEAS DE CONTORNO: FUNCIONES *CONTOUR* Y *CONTOUR3*

Una forma distinta de representar funciones tridimensionales es por medio de isolíneas o curvas de nivel. Ahora se verá cómo se puede utilizarlas con las matrices de datos **Z** y **W** que tenemos calculadas:

```
» contour(Z,20)
» contour3(Z,20)
» contour(W,20)
» contour3(W,20)
```

donde "20" representa el número de líneas de nivel. Si no se pone se utiliza un número por defecto. Otras posibles formas de estas funciones son las siguientes:

`» contour(Z, val)`	siendo **val** un vector de valores para las isolíneas a dibujar
`» contour(u,v,W,20)`	se utilizan **u** y **v** para dar valores a los ejes de coordenadas
`» contour(Z,20,'r--')`	se puede especificar el tipo de línea como en la función ***plot***

6.2 Utilización del color en gráficos 3-D

En los dibujos realizados hasta ahora, se ha visto que el resultado adoptaba determinados colores, pero todavía no se ha explicado de dónde han salido. Ahora se verá qué sistema utiliza MATLAB para determinar los colores.

6.2.1 MAPAS DE COLORES

Un ***mapa de colores*** se define como una matriz de tres columnas, cada una de las cuales contiene un valor entre 0 y 1 que representa la intensidad de uno de los colores fundamentales: R (red o rojo), G (green o verde) y B (blue o azul).

La longitud por defecto de los mapas de colores de MATLAB es 64, es decir, cada mapa de color contiene 64 colores.

Algunos mapas de colores están predefinidos en MATLAB, por ejemplo, con ***help color*** se obtiene –entre otra información– la lista de los siguientes mapas de colores:

```
hsv         - Hue-saturation-value color map.
gray        - Linear gray-scale color map.
hot         - Black-red-yellow-white color map.
cool        - Shades of cyan and magenta color map.
bone        - Gray-scale with a tinge of blue color map.
copper      - Linear copper-tone color map.
pink        - Pastel shades of pink color map.
prism       - Prism color map.
jet         - A variant of HSV.
flag        - Alternating red, white, blue, and black color map.
```

Para visualizar estos mapas de colores se pueden utilizar los comandos:

```
» colormap(hot)
» pcolor([1:65;1:65]')
```

donde la función ***pcolor*** permite visualizar por medio de colores la magnitud de los elementos de una matriz (en realidad representa colores de “celdas”, para lo que necesita que la matriz tenga una fila y columna más de las necesarias; esa es la razón de que en el ejemplo anterior se le pasen 65 filas y 2 columnas).

Si se desea imprimir una figura en una impresora láser en blanco y negro, puede utilizarse el mapa de color ***gray***. En el siguiente apartado se explica con más detalle el dibujo en "pseudocolor" (*pcolor*, abreviadamente).

El comando ***colormap*** actúa sobre la figura activa, cambiando sus colores. Si no hay ninguna figura activa, sustituye al mapa de color anterior para las siguientes figuras que se vayan a dibujar.

6.2.2 IMÁGENES Y GRÁFICOS EN PSEUDOCOLOR

Cuando se desea dibujar una figura con un determinado mapa de colores se establece una correspondencia (o un *mapping*) entre los valores de la función y los colores del mapa de colores. Esto hace que los valores pequeños se dibujen con los colores *bajos* del mapa, mientras que los valores grandes se dibujan con los colores *altos*.

La función ***pcolor*** es -en cierta forma- equivalente a la función ***surf*** con el punto de vista situado perpendicularmente al dibujo. Un ejemplo interesante de uso de la función ***pcolor*** es el siguiente: se genera una matriz **A** de tamaño 100x100 con valores aleatorios entre 0 y 1. La función ***pcolor(A)*** dibuja en color los elementos de la matriz **A**, mientras que la función ***pcolor(inv(A))*** dibuja los colores correspondientes a los elementos de la matriz inversa. Se puede observar que los colores de la matriz inversa son mucho más uniformes que los de la matriz original. Los comandos son los siguientes:

```
» A=rand(100,100); colormap(hot); pcolor(A); pause(5), pcolor(inv(A));
```

donde el comando ***pause(5)*** simplemente introduce un pausa de 5 seg en la ejecución. Al ejecutar todos los comandos en la misma línea es necesario poner ***pause*** pues si no dibuja directamente la inversa sin pasar por la matriz inicial.

Si todavía se conservan las matrices **Z** y **W** que se han definido previamente, se pueden hacer algunas pruebas cambiando el mapa de colores.

6.2.3 DIBUJO DE SUPERFICIES FACETEADAS

La función ***surf*** tiene diversas posibilidades referentes a la forma en que son representadas las facetas o polígonos coloreados. Las tres posibilidades son las siguientes:

shading flat	determina sombreado con color constante para cada polígono. Este sombreado se llama plano o *flat*.
shading interp	establece que el sombreado se calculará por interpolación de colores entre los vértices de cada faceta. Se llama también sombreado de Gouraud
shading faceted	consiste en sombreado constante con líneas negras superpuestas. Esta es la opción por defecto

Edite el fichero ***picos.m*** de forma que aparezcan menos facetas y más grandes. Se puede probar con ese fichero, eliminando la función ***mesh***, los distintos tipos de sombreado o *shading* que se acaban de citar. Para obtener el efecto deseado, basta poner la sentencia ***shading*** a continuación de la sentencia ***surf***.

6.2.4 OTRAS FORMAS DE LAS FUNCIONES *MESH* Y *SURF*

Por defecto, las funciones ***mesh*** y ***surf*** atribuyen color a los bordes y facetas en función de los valores de la función, es decir en función de los valores de la matriz **Z**. Ésta no es sin embargo la única posibilidad. En las siguientes funciones, las dos matrices argumento **Z** y **C** tienen el mismo tamaño:

```
mesh(Z,C)
surf(Z,C)
```

pero mientras se dibujan los valores de **Z**, los colores se obtienen de **C**. Un caso típico es aquél en el que se quiere que los colores dependan de la curvatura de la superficie (y no de su valor). MATLAB dispone de la función ***del2***, que sustituye el valor de un elemento de la matriz por el promedio de los 4 elementos contiguos, produciendo una matriz proporcional a la curvatura. Obsérvese el efecto de esta forma de la función ***surf*** en el siguiente ejemplo (si todavía se tiene la matriz **Z** formada a partir de ***test3d***, utilícese. Si no se conserva, vuélvase a calcular):

```
» C=del2(Z);
» close, surf(Z,C)
```

6.2.5 FORMAS PARAMÉTRICAS DE LAS FUNCIONES *MESH*, *SURF* Y *PCOLOR*

Existen unas formas más generales de las funciones ***mesh***, ***surf*** y ***pcolor***. Son las siguientes (se presentan principalmente con la funciones ***mesh*** y ***surf***). La función:

```
mesh(x,y,Z,C)
```

dibuja una superficie cuyos puntos tienen como coordenadas (x(j), y(i), Z(i,j)) y como color C(i,j). Obsérvese que **x** varía con el índice de columnas e **y** con el de filas. Análogamente, la función:

```
mesh(X,Y,Z,C)
```

dibuja una superficie cuyos puntos tienen como coordenadas (X(i,j), Y(i,j), Z(i,j)) y como color C(i,j). Las cuatro matrices deben ser del mismo tamaño. Si todavía están disponibles las matrices calculadas con el fichero ***picos.m*** ejecútese el siguiente comando y obsérvese que se obtiene el mismo resultado que anteriormente:

```
» close,surf(X,Y,Z),pause(5),mesh(X,Y,Z)
```

¿Cuál es la ventaja de estas nuevas formas de las funciones ya conocidas? La principal es que admiten más variedad en la forma de representar la cuadrícula en el plano (x-y). La primera forma admite vectores **x** e **y** con puntos desigualmente espaciados, y la segunda admite conjuntos de puntos muy generales, incluso los provenientes de coordenadas *cilíndricas* y *esféricas*.

6.2.6 OTRAS FUNCIONES GRÁFICAS 3D

Las siguientes funciones se derivan directamente de las anteriores, pero añaden algún pequeño detalle y/o funcionalidad:

surfc	combinacion de ***surf***, y ***contour*** en z=0
meshz	***mesh*** con plano de referencia en el valor mínimo y una especie de "cortina" en los bordes del dominio de la función
surfl	para controlar la iluminación determinando la posición e intensidad de un foco de luz.

Se pueden probar estas funciones con los datos de que se dispone. Utilícese el ***help*** para ello.

6.2.7 ELEMENTOS GENERALES: EJES, PUNTOS DE VISTA, LÍNEAS OCULTAS, ...

Las funciones ***surf*** y ***mesh*** dibujan funciones tridimensionales en perspectiva. La localización del punto de vista o dirección de observación se puede hacer mediante la función ***view***, que tiene la siguiente forma:

```
view(azimut, elev)
```

donde ***azimut*** es el ángulo de rotación horizontal, medido sobre el eje **z** a partir del eje **x** en sentido antihorario, y ***elev*** es el ángulo de elevación respecto al plano **x-y**. Ambos ángulos se miden ***en grados***, y pueden tomar valores positivos y negativos (sus valores por defecto son -37.5 y 30). También se puede definir la dirección del punto de vista mediante las tres coordenadas cartesianas de un vector (sólo se tiene en cuenta la dirección):

```
view([xd,yd,zd])
```

En los gráficos tridimensionales existen funciones para controlar los ejes, por ejemplo:

```
axis([xmin,xmax,ymin,ymax,zmin,zmax])
```

También se pueden utilizar las funciones siguientes: ***xlabel***, ***ylabel***, ***zlabel***, ***axis('auto')***, ***axis(axis)***, etc.

Las funciones ***mesh*** y ***surf*** disponen de un algoritmo de *eliminación de líneas ocultas* (los polígonos o facetas, no dejan ver las líneas que están detrás). El comando ***hidden*** activa y desactiva la eliminación de líneas ocultas.

En el dibujo de funciones tridimensionales, a veces también son útiles los *NaNs*. Cuando una parte de los elementos de la matriz de valores **Z** son *NaNs*, esa parte de la superficie no se dibuja, permitiendo ver el resto de la superficie.

7. Programación de MATLAB

Como ya se ha dicho varias veces –incluso con algún ejemplo– MATLAB es una aplicación que se puede programar muy fácilmente. De todas formas, como lenguaje de programación pronto verá que no tiene tantas posibilidades como C (ni tan complicadas...). Se comenzará viendo las bifurcaciones y bucles, y la lectura y escritura interactiva de variables, que son los elementos básicos de cualquier programa de una cierta complejidad.

7.1 Bifurcaciones y bucles

Las bifurcaciones y los bucles se encuentran explicados en el apartado 2.9 de este manual.

7.2 Lectura y escritura interactiva de variables

Se verá a continuación una forma sencilla de leer variables desde teclado y escribir mensajes en la pantalla del PC. Más adelante se considerarán otros modos más generales –y complejos– de hacerlo.

7.2.1 FUNCIÓN *INPUT*

La función ***input*** permite imprimir un mensaje en la pantalla y recuperar como valor de retorno un valor numérico o el resultado de una expresión tecleada por el usuario. Después de imprimir el mensaje, el programa espera que el usuario teclee el valor numérico o la expresión. Cualquier expresión válida de MATLAB es aceptada por este comando. El usuario puede teclear simplemente un vector o una matriz. En cualquier caso, la expresión introducida es evaluada con los valores actuales de MATLAB y el resultado se devuelve como valor de retorno. Véase un ejemplo de uso de esta función:

```
n = input('Teclee el número de ecuaciones')
```

Otra posible forma de esta función es la siguiente (obsérvese el parámetro `'s'`):

```
nombre = input('¿Cómo te llamas?','s')
```

En este caso el texto tecleado como respuesta se lee y se devuelve sin evaluar, con lo que se almacena en la cadena ***nombre***. Así pues, en este caso, si se teclea una fórmula, se almacena como texto sin evaluarse.

7.2.2 FUNCIÓN *DISP*

La función ***disp*** permite imprimir en pantalla un mensaje de texto o el valor de una matriz, pero sin imprimir su nombre. En realidad, ***disp*** siempre imprime vectores y/o matrices: las

cadenas de caracteres son un caso particular de vectores. Considérense los siguientes ejemplos de cómo se utiliza:

```
disp('El programa ha terminado')
A=rand(4,4)
disp(A)
```

Obsérvese la diferencia entre las dos formas de imprimir la matriz **A**.

7.3 Ficheros *.m

Los ficheros con extensión (***.m***) son ficheros de texto (ASCII) que constituyen el centro de la programación en MATLAB. Ya se han utilizado en varias ocasiones. Estos ficheros se crean y modifican con un editor de textos cualquiera. En el caso de MATLAB ejecutado en un PC bajo ***Windows***, lo más sencillo es utilizar ***Notepad*** como editor de textos. Con ***Options/Editor Preference*** en la ventana principal de MATLAB se puede elegir el editor que se desee utilizar.

Existen dos tipos de ficheros ****.m***, los *ficheros de comandos* (llamados *scripts* en inglés) y las *funciones*. Los primeros contienen simplemente una sucesión de comandos que se ejecutan sucesivamente cuando se teclea el nombre del fichero en la línea de comandos de MATLAB. Un fichero de comandos puede llamar a otros ficheros de comandos.

Las *funciones* permiten definir funciones enteramente análogas a las de MATLAB, con su nombre, sus argumentos y sus valores de retorno. Los ficheros ****.m*** que definen funciones permiten extender las posibilidades de MATLAB; de hecho existen bibliotecas de ficheros ****.m*** que se venden (***toolkits***) o se distribuyen gratuitamente (a través de la *Internet*).

Recuérdese que un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso puede llamarse a sí mismo de forma recursiva.

A continuación se verá con un poco más de detalle ambos tipos de ficheros.

7.3.1 FICHEROS DE COMANDOS

Como ya se ha dicho, los ficheros de comandos o *scripts* son ficheros con un nombre tal como ***file1.m*** que contienen una sucesión de comandos análoga a la que se tecletaría en el uso interactivo del programa. Dichos comandos se ejecutan cuando se teclea el nombre del fichero que los contiene (sin la extensión), es decir cuando se teclea ***file1*** con el ejemplo considerado.

En los ficheros de comandos conviene poner los (;) al final de cada sentencia, para evitar una salida de resultados demasiado cuantiosa. Un fichero ****.m*** puede llamar a otros ficheros ****.m***, e incluso se puede llamar a sí mismo de modo recursivo.

Las variables definidas por los ficheros de comandos son *variables globales*, esto es variables con el mismo carácter que las que se crean interactivamente en MATLAB. Al terminar la ejecución del *script*, dichas variables permanecen en memoria.

El comando ***echo*** hace que se impriman los comandos que están en un script a medida que van siendo ejecutados. Este comando tiene varias formas:

echo on	activa el ***echo*** en todos los ficheros script
echo off	desactiva el ***echo***
echo file on	donde 'file' es el nombre de un fichero de función, activa el ***echo*** en esa función
echo file off	desactiva el ***echo*** en la función
echo file	pasa de **on** a **off** y viceversa

echo on all	activa el ***echo*** en todas las funciones
echo off all	desactiva el ***echo*** de todas las funciones

Mención especial merece el fichero de comandos ***startup.m***. Este fichero se ejecuta cada vez que se entra en MATLAB. En él puede introducir todos aquellos comandos que le interesa se ejecuten siempre al iniciar la sesión, por ejemplo ***format compact*** y los comandos necesarios para modificar el ***path***.

7.3.2 DEFINICIÓN DE FUNCIONES

La ***primera línea*** de un fichero llamado ***name.m*** que define una función tiene la forma:

```
function [lista de valores de retorno] = name(lista de argumentos)
```

donde ***name*** es el nombre de la función. Entre corchetes y separados por comas van los ***valores de retorno***, y entre paréntesis también separados por comas los ***argumentos***. Recuérdese que los argumentos son los datos de la función y los valores de retorno sus resultados.

Una diferencia importante con C es que en MATLAB los argumentos de una función no se modifican nunca, y los resultados se obtienen siempre a través de los valores de retorno, que pueden ser múltiples y matriciales. Tanto el número de argumentos como el de valores de retorno no son fijos, dependiendo de cómo el usuario llama a la función.

Las variables definidas dentro de una función son *variables locales*, en el sentido de que son inaccesibles desde otras partes del programa y en el de que no interfieren con variables del mismo nombre definidas en otras funciones o partes del programa. Para que la función tenga acceso a variables que no han sido pasadas como argumentos es necesario declarar dichas variables como ***variables globales***, tanto en el programa principal como en las distintas funciones que deben acceder a su valor. Es frecuente utilizar el convenio de usar para las variables globales nombres largos (más de 5 letras) y con mayúsculas.

Por razones de eficiencia, los argumentos de una función no se copian si no son modificados por la función (en términos de C diríamos que se pasan *por referencia*). Esto tiene importantes consecuencias en términos de eficiencia y ahorro de tiempo de cálculo. Sin embargo, si dentro de la función se realizan modificaciones sobre ellos, se sacan copias y se modifican las copias (diríase que en este caso se pasan *por valor*).

Dentro de la función, los valores de retorno deben ser calculados en algún momento (no hay sentencia *return*, como en C). De todas formas, no hace falta calcular siempre todos los posibles valores de retorno de la función, sino sólo los que el usuario espera obtener. Existen dos variables definidas de modo automático, llamadas ***nargin*** y ***nargout***, que representan respectivamente el número de argumentos y el número de valores de retorno actuales, con los que la función ha sido llamada. Dentro de la función, estas variables puden ser utilizadas como el usuario desee.

7.3.3 *HELP* PARA LAS FUNCIONES DE USUARIO

También las funciones creadas por el usuario pueden tener su ***help***, análogo al que tienen las propias funciones de MATLAB. Esto se consigue de la siguiente forma: las primeras líneas de comentarios de cada fichero de función son muy importantes, pues permiten construir ***help*** sobre esa función. En otras palabras, cuando se teclea:

```
» help mi_func
```

en la ventana de comandos de MATLAB, el programa responde escribiendo las primeras líneas del fichero ***mi_func.m*** que comienzan por el carácter (%), es decir, que son comentarios.

De estas líneas, tiene una importancia particular la primera línea de comentarios. En ella hay que intentar poner toda la información relevante sobre esa función. La razón es que existe una función, llamada ***lookfor*** que busca una determinada palabra en cada primera línea de comentario de todas las funciones ****.m***.

7.3.4 VARIABLES GLOBALES

Las variables globales son visibles en todas las funciones (y en el espacio de trabajo base o general) que las declaran como tales. Dichas variables de declaran precedidas por la palabra ***global*** y separadas por blancos, en la forma:

```
global VARIABLE1 VARIABLE2
```

Como ya se ha apuntado, estas variables sólo son visibles en las funciones (y en el propio espacio de trabajo) que las declaran como tales. Ya se ha dicho también que se suele recurrir al criterio de utilizar nombres largos y con mayúsculas, para distinguirlas fácilmente de las demás variables.

7.4 Cadenas de caracteres

MATLAB trabaja también con cadenas de caracteres, con ciertas semejanzas y también diferencias respecto a C. A continuación se explica lo más importante del manejo de cadenas de caracteres en MATLAB.

7.4.1 FUNCIONES PARA MANEJO DE CADENAS DE CARACTERES

Los caracteres se almacenan en un vector, un carácter por elemento. Cada carácter ocupa un byte. Las cadenas de caracteres van entre apóstrofos o comillas simples. A continuación puede ver –y practicar con ellos– algunos ejemplos:

```
» c='cadena'
c =
cadena
» size(c)
ans =
     1     6
» abs(c)           % devuelve los números ASCII de cada carácter
ans =
    99    97   100   101   110    97
» setstr(abs(c))   % convierte números ASCII en caracteres
ans =
cadena
```

Otras funciones para manejo de caracteres son las siguientes:

disp(c)	imprime el texto contenido en la variable c
isstr	detecta si una variable es una cadena de caracteres
strcmp	comparación de cadenas. Funciona de modo diferente que la correspondiente función de C. Si las cadenas son iguales devuelve un uno, y si no lo son, devuelve un cero
s=[s,' y más']	concatena cadenas, añadiendo la segunda a continuación de la primera

int2str	convierte un número entero en cadena de caracteres
num2str	convierte un número real en cadena de caracteres, con cuatro cifras decimales por defecto (pueden especificarse más cifras, con un argumento opcional)
sprintf	convierte valores numéricos en cadenas de caracteres, de acuerdo con las reglas y formatos de conversión del lenguaje C. Esta es la función más general para este tipo de conversión

A continuación se pueden ver algunos ejemplos:

```
» num2str(pi)
ans =
3.142
» num2str(pi,8)
ans =
3.1415927
```

Es habitual convertir los valores numéricos en cadenas de caracteres para poder imprimirlos como títulos en los dibujos o gráficos. Véase el siguiente ejemplo:

```
fahr=70; grad=(fahr-32)/1.8;
title(['Temperatura ambiente: ',num2str(grad),' grados centígrados'])
```

7.4.2 LAS FUNCIONES *EVAL* Y *FEVAL*

La función ***eval***('cadena de caracteres') hace que se evalúe como expresión de MATLAB el texto contenido entre las comillas como argumento de la función. Este texto puede ser un comando, una fórmula matemática o -en general- una expresión válida de MATLAB. La función ***eval*** puede tener los valores de retorno necesarios para recoger los resultados de la expresión evaluada.

Esta forma de definir *macros* es particularmente útil para pasar nombres de función a otras funciones definidas en ficheros ****.m***.

El siguiente ejemplo va creando variables llamadas **A1**, **A2**, ..., **A10** utilizando la posibilidad de concatenar cadenas antes de pasárselas como argumento a la función ***eval***:

```
for n = 1:10
    eval(['A'num2str(n)' = magic(n)'])
end
```

Por su parte la función ***feval*** sirve para evaluar, dentro de una función, otra función cuyo nombre se ha recibido como argumento. Por ejemplo, si dentro de una función se quiere evaluar la función ***calcular(A, b, c)***, donde el nombre ***calcular*** se envía como argumento en la cadena ***nombre***, entonces ***feval(nombre, A, b, c)*** equivale a ***calcular(A, b, c)***.

7.5 Entrada y salida de datos

Ya se ha visto una forma de realizar la entrada interactiva de datos por medio de la función ***input*** y de imprimir resultados por medio de la función ***disp***. Ahora se van a ver otras formas de intercambiar datos con otras aplicaciones.

7.5.1 IMPORTAR DATOS DE OTRAS APLICACIONES

Hay varias formas de pasar datos de otras aplicaciones –por ejemplo de ***Excel***– a MATLAB. Se pueden enumerar las siguientes:

– se puede utilizar el ***Copy*** y ***Paste*** para copiar datos y depositarlos entre los corchetes de una matriz o vector, en una línea de comandos. Tiene el inconveniente de que estos datos no se pueden editar.

– se puede crear un fichero ****.m*** con un editor de textos como ***Notepad***, con lo cual no existen problemas de edición.

– es posible leer un ***flat file*** en ASCII. Un *flat file* es un fichero con filas de longitud constante separadas con ***Intro***, y varios datos por fila separados por *blancos*. Estos ficheros pueden ser leídos desde MATLAB con el comando ***load***. Si se ejecuta ***load datos.txt*** el contenido del ***flat file*** se deposita en una matriz con el nombre ***datos***. Para más información utilizar ***help load***.

– se pueden leer datos de un fichero con las funciones ***fopen*** y ***fread***.

– existen también otros métodos posibles: escribir funciones en C para traducir a formato ****.mat*** (y cargar después con ***load***), crear un fichero ejecutable ****.mex*** que lea los datos, etc.

7.5.2 EXPORTAR DATOS A OTRAS APLICACIONES

De forma análoga, también los resultados de MATLAB se pueden exportar a otras aplicaciones como ***Word*** o ***Excel***.

– utilizar el comando ***diary*** (para datos pequeño tamaño)

– utilizar el comando ***save*** con la opción ***-ascii***

– utilizar las funciones de bajo nivel ***fopen***, ***fwrite*** y otras

– otros métodos: escribir subrutinas en C para traducir de formato ****.mat*** (guardando previamente con ***save***), crear un fichero ejecutable ****.mex*** que escriba los datos, etc.

Hay que señalar que los ficheros binarios ****.mat*** son trasportables entre versiones de MATLAB en distintos tipos de computadores porque contienen información sobre el tipo de máquina en el *header* del fichero, y el programa realiza la trasformación de modo automático. Los ficheros ****.m*** son de tipo ASCII, y por tanto pueden ser leídos por distintos computadores sin problemas de ningún tipo.

7.6 Lectura y escritura de ficheros

MATLAB dispone de funciones análogas a las del lenguaje C (en las que están inspiradas) aunque con algunas diferencias. En general son versiones simplificadas –con menos opciones y posibilidades– que las correspondientes funciones de C.

7.6.1 FUNCIONES *FOPEN* Y *FCLOSE*

Estas funciones sirven para abrir y cerrar ficheros, respectivamente. La función ***fopen*** tiene la forma siguiente:

```
[fi,texto] = fopen('filename','c')
```

donde **fi** es un valor de retorno que sirve como identificador del fichero, **texto** es un mensaje para caso de que se produzca un error, y **c** es un carácter (o dos) que indica el tipo de operación que se desea realizar. Las opciones son las siguientes:

'r'	lectura (de read)
'w'	escritura reemplazando (de write)
'a'	escritura a continuación (de append)
'r+'	lectura y escritura

Cuando por alguna razón el fichero no puede ser abierto, se devuelve un (-1). En este caso el valor de retorno **texto** puede proporcionar información sobre el tipo de error que se ha producido (también existe una función llamada ***ferror*** que permite obtener información sobre los errores).

Después de realizar las operaciones de lectura y escritura deseadas, el fichero se puede cerrar con la función ***close*** en la forma siguiente:

```
st = fclose(fi)
```

donde **st** es un valor de retorno para posibles condiciones de error. Si se quieren cerrar a la vez todos los ficheros abiertos puede utilizarse el comando:

```
st = close('all')
```

7.6.2 FUNCIONES *FSCANF*, *SSCANF*, *FPRINTF* Y *SPRINTF*

Estas funciones permiten leer y escribir en ficheros ASCII, es decir, en ficheros formateados. La forma general de la función ***fscanf*** es la siguiente:

```
[var1,var2,...] = fscanf(fi,'cadena de control',size)
```

donde **fi** es el identificador del fichero (devuelto por la función ***fopen***), y **size** es un argumento opcional que puede indicar el tamaño del vector o matriz a leer. Obsérvese otra diferencia con C: las variables leídas se devuelven como valor de retorno y no como argumentos pasados por referencia (precedidos por el carácter &). La cadena de control va encerrada entre apóstrofos simples, y contiene los especificadores de formato para las variables:

%s	para cadenas de caracteres
%d	para variables enteras
%f	para variables de punto flotante
%lf	para variables de doble precisión

La función ***sscanf*** es similar a ***fscanf*** pero la entrada no proviene de un fichero sino de una cadena de caracteres.

Finalmente, la función ***fprintf*** dirige su salida formateada hacia el fichero indicado por el identificador. Su forma general es:

```
fprintf(fi,'cadena de control',var1,var2,...)
```

Esta es la función más parecida a su homóloga de C. La cadena de control contiene los formatos de escritura, que son similares a los de C, como muestran los ejemplos siguientes:

```
fprintf(fi,'El número de ecuaciones es: %d\n',n)
fprintf(fi,'El determinante es: %lf10.4\n',n)
```

De forma análoga, la función ***sprintf*** convierte su resultado en una cadena de caracteres, en vez de enviarlo a un fichero. Véase un ejemplo:

```
resultado = sprintf('El cuadrado de %f es %12.4f\n',n,n*n)
```

donde **resultado** es una cadena de caracteres. Esta función constituye el método más general de convertir números en cadenas de caracteres, por ejemplo para ponerlos como títulos de figuras.

7.6.3 FUNCIONES *FREAD* Y *FWRITE*

Estas funciones son análogas a ***fscanf*** y ***fprintf***, pero en vez de leer o escribir en un fichero de texto (ASCII), lo hacen en un fichero *binario*. Aunque dichos ficheros no se pueden leer y/o modificar con un editor de textos, tienen la ventaja de que las operaciones de lectura y escritura son mucho más rápidas y eficientes. Esto es particularmente significativo para grandes ficheros de datos. Para más información sobre estas funciones se puede utilizar el ***help***.

7.6.4 FICHEROS DE ACCESO DIRECTO

De ordinario los ficheros de disco se leen y escriben secuencialmente, es decir, de principio a final, sin volver nunca hacia atrás ni realizar saltos. A veces interesa acceder a un fichero de un modo arbitrario, sin ningún orden preestablecido. Esto se puede conseguir con las funciones ***ftell*** y ***fseek***.

En cada momento, hay una especie de *cursor* que indica en qué parte del fichero se está posicionado. La función ***fseek*** permite mover este cursor hacia delante o hacia atrás, respecto a la posición actual ('cof'), respecto al principio ('bof') o respecto al final del fichero ('eof'). La función ***ftell*** indica en qué posición está el cursor. Si alguna vez se necesita utilizar este tipo de acceso a disco, se puede buscar más información por medio del ***help***.

7.7 Recomendaciones generales de programación

Las funciones vectoriales de MATLAB son mucho más rápidas que sus contrapartidas escalares. En la medida de lo posible es muy interesante vectorizar los algoritmos de cálculo, es decir, realizarlos con vectores y matrices y no con variables escalares.

Aunque los vectores y matrices pueden ir creciendo a medida que se necesita, es mucho más rápido reservarles toda la memoria necesaria al comienzo del programa. Se puede utilizar para ello la función ***zeros***. Además de este modo la memoria reservada es contigua.

8. Construcción de Interfaces Gráficas con MATLAB

MATLAB permite desarrollar de manera simple un conjunto de pantallas con botones, menús, ventanas, etc., que permiten utilizar de manera muy simple programas realizados en este entorno. Este conjunto de herramientas se denomina ***interface de usuario***. Las posibilidades que ofrece MATLAB no son muy amplias, en comparación a otras aplicaciones de ***Windows*** como ***Visual Basic***. Para poder hacer programas que utilicen las capacidades gráficas avanzadas de MATLAB hay que conocer algunos conceptos que se explican en los apartados siguientes.

8.1 Estructura de los gráficos de MATLAB

Los gráficos de MATLAB tienen una estructura jerárquica formada por ***objetos*** de distintos ***tipos***. Esta jerarquía tiene forma de *árbol*, con el aspecto mostrado en la figura 10.

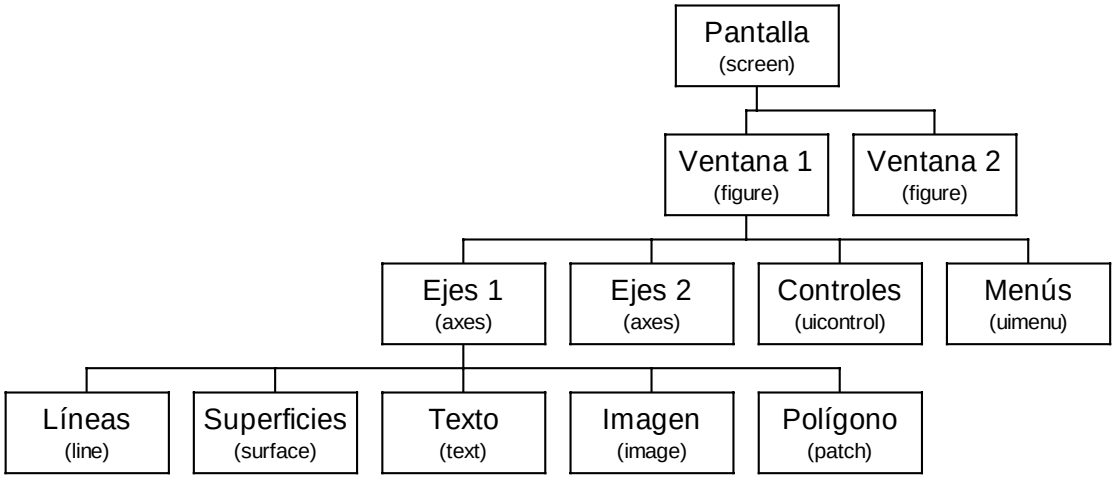


Figura 10. Jerarquía gráfica de MATLAB.

8.1.1 OBJETOS GRÁFICOS DE MATLAB

Según se muestra en la figura 10, el objeto más general es la ***pantalla*** (***screen***). Este objeto es la raíz de todos los demás y sólo puede haber un objeto pantalla. Una pantalla puede contener una o más ***ventanas*** (***figures***). A su vez cada una de las ventanas puede tener uno o más ***ejes*** de coordenadas (***axes***) en los que representar otros objetos de más bajo nivel. Una ventana puede tener también ***controles*** (***uicontrol***) tales como botones, barras de desplazamiento, botones de selección o de opción, etc.) y ***menús*** (***uimenu***). Finalmente, los ejes pueden contener los cinco tipos de elementos gráficos que permite MATLAB: ***líneas*** (***line***), ***polígonos*** (***patches***), ***superficies*** (***surface***), ***imágenes bitmap*** (***image***) y ***texto*** (***text***).

La jerarquía de objetos mostrada en la figura 10 indica que en MATLAB hay ***objetos padres*** e ***hijos***. Por ejemplo, todos los objetos ***ventana*** son hijos de ***pantalla***, y cada ***ventana*** es padre de los objetos ***ejes***, ***controles*** o ***menús*** que están por debajo. A su vez los elementos gráficos (líneas, polígonos, etc.) son hijos de un objeto ***ejes***, y no tienen otros objetos que sean sus hijos.

Cuando se borra un objeto de MATLAB automáticamente se borran todos los objetos que son sus descendientes. Por ejemplo, al borrar unos ejes, se borran todas las líneas y polígonos que son hijos suyos.

8.1.2 IDENTIFICADORES (*HANDLES*)

Cada uno de los objetos de MATLAB tiene un ***identificador único*** (***handle***). En este escrito a los identificadores se les llamará ***handle*** o ***id***, indistintamente. Algunos gráficos tienen muchos objetos, en cuyo caso tienen múltiples *handles*. El ***objeto raíz*** (pantalla) es siempre único y su identificador es el ***cero***. El identificador de las ventanas es un entero, que aparece en la barra de nombre de dicha ventana. Los identificadores de otros elementos gráficos son números *float*, que pueden ser obtenidos como valor de retorno y almacenados en variables de MATLAB.

MATLAB puede tener varias ventanas abiertas, pero siempre hay una y sólo una que es la ventana activa. A su vez una ventana puede tener varios ejes, pero sólo unos son los ejes activos. MATLAB dibuja en los ejes activos de la ventana activa. Los identificadores de la ventana activa, de los ejes activos y del objeto activo se pueden obtener respectivamente con los comandos ***gcf*** (*get current figure*), ***gca*** (*get current axes*) y ***gco*** (*get current object*):

gcf devuelve un entero, que es el handle de la ventana activa

gca devuelve el handle de los ejes activos

gco devuelve el handle del objeto activo

Los objetos se pueden borrar con el comando ***delete***:

delete(handle) borra el objeto correspondiente y todos sus hijos

MATLAB dispone de funciones gráficos de alto y bajo nivel. Son funciones de alto nivel las funciones ***plot***, ***plot3***, ***mesh***, ***surf***, ***fill***, ***fill3***, etc. Cada una de estas funciones llama a una o más funciones de bajo nivel. Las funciones de bajo nivel crean cada uno de los 9 tipos de objetos disponibles y de ordinario tienen el nombre inglés del objeto correspondiente: ***figure***, ***axes***, ***uicontrol***, ***uimenu***, ***line***, ***patch***, ***surface***, ***image*** y ***text***.

Como ejemplo, ejecútense los siguientes comandos, observando la evolución de lo dibujado en la ventana y como MATLAB devuelve el ***id*** de cada objeto como valor de retorno:

```
» fig = figure
» li1 = line([0,5],[0,5])
» li2 = line([0,5],[5,0])
» po1 = patch([1,4,3],[1,1,4],'g')
» delete(po1)
» delete(li1)
```

Estos valores de retorno pueden ser almacenados en variables para un uso posterior. En ocasiones el ***id*** es un vector de valores, como por ejemplo al crear una superficie que consta de líneas y polígonos. Los comandos anteriores han abierto una ventana y dibujado sobre ella dos líneas cruzadas de color amarillo y un triángulo de color verde.

8.2 Propiedades de los objetos

Todos los objetos de MATLAB tienen distintas ***propiedades***. Algunas de éstas son el *tipo*, el *estilo*, el *padre*, los *hijos*, si es *visible* o no, y otras propiedades particulares del objeto concreto de que se trate. Las propiedades comunes a todos los objetos son: ***children***, ***clipping***, ***parent***, ***type***, ***UserData***, ***Visible***. Otras propiedades son propias de un tipo determinado de objeto.

Las propiedades tienen ***valores por defecto***, que se utilizan siempre que el usuario no indique otra cosa. Es posible cambiar las propiedades por defecto, y también devolverles su valor original (llamado ***factory***, por ser el valor por defecto con que salen de fábrica). El usuario puede consultar (***query***) los valores de las propiedades de cualquier objeto. Algunas propiedades pueden ser modificadas y otras no (son ***read only***). Hay propiedades que pueden tener cualquier valor y otras que sólo pueden tener un conjunto limitado de valores (por ejemplo, ***on*** y ***off***).

8.2.1 FUNCIONES *SET()* Y *GET()*

MATLAB dispone de las funciones ***set*** y ***get*** para consultar y cambiar el valor de las propiedades de un objeto. Las funciones ***set(id)*** lista en pantalla todas las propiedades del objeto al que corresponde el *handle* (sólo los *nombres*, sin los valores de las propiedades). La función ***get(id)*** produce un listado de las propiedades y de sus *valores*. Como ejemplo, siendo ***li1*** el ***id*** de la primera línea dibujada anteriormente, la respuesta a ***set(li1)*** es:

```
» set(li1)
     Color
     EraseMode: [ {normal} | background | xor | none ]
     LineStyle: [ {-} | -- | : | -. | + | o | * | . | x ]
     LineWidth
     MarkerSize
     Xdata
     Ydata
     Zdata

     ButtonDownFcn
     Clipping: [ {on} | off ]
     Interruptible: [ {no} | yes ]
     Parent
     UserData
     Visible: [ {on} | off ]
```

que muestra las propiedades del objeto *línea*. Las propiedades que tienen un conjunto finito de valores presentan estos valores entre *corchetes* [ ] separados por barras verticales. La opción por defecto se muestra entre *llaves* { }. En general el significado de cada propiedad es bastante evidente a partir de su nombre.

El comando ***get(id)*** devuelve las propiedades del objeto junto con sus valores:

```
» get(li1)
     Color = [1 1 0]
     EraseMode = normal
     LineStyle = -
     LineWidth = [0.5]
     MarkerSize = [6]
     Xdata = [0 5]
     Ydata = [0 5]
     Zdata = []

     ButtonDownFcn =
     Children = []
     Clipping = on
     Interruptible = no
     Parent = [34.0007]
     Type = line
     UserData = []
     Visible = on
```

Para conocer el valor de una propiedad particular de un objeto se utiliza la función ***get(id,'propiedad')***.

```
» get(li1,'color')
ans =
     1     1     0
```

Las propiedades de un objeto pueden ser cambiadas o modificadas (salvo que sean *read-only*) con el comando ***set(id,'propiedad','valor')***. Por ejemplo, para cambiar el color de la segunda línea en el ejemplo anterior:

```
» set(li2,'color','r')
```

Es interesante hacer pruebas con los distintos tipos de objetos gráficos que se pueden crear y manipular con MATLAB. Por ejemplo, ejecútense los siguientes comandos observando cómo el grosor de las líneas y los colores van cambiando:

```
» close(gcf)
» fig=figure
» li1=line([0,5],[0,5])
» li2=line([0,5],[5,0],'color','w')
» po1=patch([1,4,3],[1,1,4],'g')
```

```
» pause(3)
» set(li1,'LineWidth',2), pause(1);
» set(li2,'LineWidth',2,'color','r'), pause(1);
» set(po1,'LineWidth',2,'EdgeColor','w','FaceColor','b')
```

El comando ***set*** permite cambiar varias propiedades a la vez, poniendo sus nombres entre apóstrofos seguidos de sus valores. Los ejemplos anteriores demuestran que es esencial disponer de los ***id*** si se desea modificar un gráfico o utilizar propiedades distintas de las de defecto.

Es posible también establecer las propiedades en el momento de la creación del objeto, como en el ejemplo siguiente que crea una figura con fondo blanco:

```
» fig = figure('color','w')
```

Se puede utilizar la propiedad ***type*** para saber qué tipo de objeto (*línea*, *polígono*, *texto*, ...) corresponde a un determinado ***id***. Esto es especialmente útil cuando el ***id*** es un vector de valores correspondientes a objetos de distinto tipo.

```
» get(li2,'type')
line
```

8.2.2 PROPIEDADES POR DEFECTO

Anteponiendo la palabra ***Default*** al nombre de un objeto y de una propiedad se puede acceder al valor por defecto de una propiedad, bien para consultar su valor, bien para modificarlo. Por ejemplo, ***DefaultLineColor*** representa el color por defecto de una *línea*, y ***DefaultFigureColor*** representa el *color de fondo* por defecto de las ventanas. Cambiando un valor por defecto a un determinado nivel de la jerarquía de objetos se cambia ese valor para todos los objetos que están por debajo y que se creen a partir de ese momento. Por ejemplo, el siguiente comando cambia el color de fondo de todas las ventanas (hijas de pantalla) que sean creadas a partir de ese momento:

```
» set(0,'DefaultFigureColor','w')
```

Cuando se crea un objeto se busca el valor por defecto de sus propiedades a su nivel, y si no se encuentra se sube en al jerarquía hasta que se encuentra un valor por defecto, y ese es el que se utiliza. Para devolver una propiedad a su valor original se utiliza el valor 'factory', como por ejemplo:

```
» set(id, 'FaceColor', 'factory')
```

De forma análoga, el valor 'remove' elimina un valor introducido previamente. Por ejemplo, para que el fondo de las ventanas deje de ser blanco se debe ejecutar el comando:

```
» set(0,'DefaultFigureColor','remove')
```

8.2.3 FUNCIONES DE UTILIDAD

MATLAB dispone de algunas funciones que permiten modificar las propiedades de algunos objetos de una forma más directa y sencilla que con las funciones ***get*** y ***set***. Algunas de estas funciones son ***axis***, ***caxis***, ***cla***, ***colormap*** y ***grid***.

Para obtener más información sobre estas funciones puede utilizarse el ***Help*** de MATLAB.

9. Creación de controles gráficos

MATLAB permite desarrollar programas con el aspecto típico de las aplicaciones de ***Windows***. En este apartado se estudiará cómo crear algunos de los controles más utilizados. Para todos los controles, se utilizará la función ***uicontrol***, que permite desarrollar dichos controles. La forma general del comando ***uicontrol*** es la siguiente:

```
id_control = uicontrol(id_parent,...
                       'Propiedad1',valor1,...
                       'Propiedad2',valor2,...
                        otras propiedades
                       'callback','sentencias')
```

Las propiedades son las opciones del comando, que se explican en el apartado siguiente. Éstas tendrán comillas (‘) a su izquierda y derecha, e irán seguidas de los parámetros necesarios. En caso de que el conjunto de propiedades de un control exceda una línea de código, es posible continuar en la línea siguiente, poniendo tres puntos seguidos (…).

9.1 Opciones del comando *uicontrol*

El comando ***uicontrol*** permite definir los controles gráficos de MATLAB descritos en el siguiente apartado. En este apartado se explican las distintas propiedades de ***uicontrol***.

9.1.1 COLOR DEL OBJETO (*BACKGROUNDCOLOR)*

Controla el color del objeto. Por defecto éste suele ser gris claro, aunque puede tomar distintos valores: ***green***, ***red, white***, etc. Éstos irán definidos entre comillas (por ejemplo '***green***').

9.1.2 ACCIÓN A EFECTUAR POR EL COMANDO (*CALLBACK*)

Este comando especifica la ***acción*** a efectuar por MATLAB al actuar sobre el control. Se trata de una expresión u orden, almacenada en una cadena de caracteres, que se ejecutará al activar el control. Esta instrucción es equivalente a realizar ***eval****(‘expresión’*).

9.1.3 CONTROL ACTIVADO/DESACTIVADO (*ENABLE)*

Este comando permite desactivar un control, de tal forma que una acción sobre el mismo (por ejemplo, apretar sobre el mismo con el ratón) no produce ningún efecto. Los valores que puede tomar esta variable son ***on*** u ***off***.

9.1.4 ALINEAMIENTO HORIZONTAL DEL TITULO (*HORIZONTALALIGNMENT*)

Esta opción determina la posición del título del control en el mismo. Los valores que puede tomar son: ***left***, ***center*** o ***right***.

9.1.5 VALOR MÁXIMO (*MAX*)

Esta opción determina el máximo valor que puede tomar un valor cuando se utilizan cajas de textos o botones de opción. En caso de botones tipo ***on/off***, que solamente admiten las dos posiciones de abierto y cerrado, esta variable corresponde al valor del mismo cuando está activado.

9.1.6 VALOR MÍNIMO (*MIN*)

Análogo a la opción anterior para el valor mínimo.

9.1.7 CONTROL DEL OBJETO PADRE (*PARENT*)

Esta opción especifica el ***id*** del objeto *padre*. Debe ir siempre en primer lugar.

9.1.8 POSICIÓN DEL OBJETO (*POSITION*)

En esta opción se determina la posición y el tamaño del control dentro del objeto *padre*. Para ello se define un vector de cuatro elementos, cuyos valores siguen el siguiente orden: [*izquierda*, *abajo*, *longitud*, *altura*]. Aquí *izquierda* y *abajo* son la distancia a la esquina inferior izquierda de la figura y *longitud* y *altura* las dimensiones del control.

9.1.9 NOMBRE DEL OBJETO (*STRING*)

Esta opción define el nombre que aparecerá en el control. Cuando el control sea una opción de menú desplegable (***popup menu***), los nombres se sitúan en orden dentro del *string*, separados por un carácter barra vertical (|).

9.1.10 TIPO DE CONTROL (*STYLE*)

Esta opción puede tomar los siguientes valores: ***pushbutton, radiobutton, checkbox, slider, edit, popupmenu*** y ***text***, según el tipo de control que se desee (Como se verá en los ejemplos siguientes, pueden usarse nombres abreviados: así, ***pushbutton*** puede abreviarse en ***push***).

9.1.11 UNIDADES (*UNITS*)

Unidades de medida en las que se interpretará el comando ***Position***. Los valores que puede tomar ***Units*** son: ***pixels*** (puntos de pantalla), ***normalized*** (coordenadas de 0 a 1), ***inches*** (pulgadas), ***cent*** (centímetros), ***points*** (equivalente a 1/72 parte de una pulgada). La opción por defecto es ***pixels***.

9.1.12 VALOR (*VALUE*)

Permite utilizar el valor que tiene del control en un momento dado. Por ejemplo, un botón de chequeo (***check button***) puede tener dos tipos de valores, definidos por las variables ***Max*** y ***Min***. Según sea uno u otro el programa ejecutará una acción. La variable ***value*** permite controlar dicho valor.

9.1.13 VISIBLE (*VISIBLE*)

Puede tomar dos valores: ***on/off***. Indica si el control es visible en la ventana o no. Este comando es similar al ***Enable***, si bien éste además de quedar inhabilitado el control éste desaparece de la pantalla.

9.2 Tipos de *uicontrol*

Existen ocho tipos de controles diferentes. La utilización de cada uno vendrá dada en función de sus características y aplicación.

9.2.1 BOTONES (*PUSH BUTTONS*)

Los botones son pequeños objetos de la pantalla normalmente acompañados con texto. Al pulsar sobre ellos con el ratón se producirá una acción que deberá ser ejecutada por MATLAB.

Figura 11. Botón (***push button***).

Ejemplo: La siguiente instrucción dibujará en la pantalla del gráfico un botón como el de la figura, que tiene como nombre *Start Plot*. Al pulsarlo se ejecutarán las instrucciones contenidas en el fichero ***mifunc.m***:

```
pbstart = uicontrol(gcf,...
   'Style','push',...
   'Position',[10 10 100 25],...
   'String','Start Plot',...
   'CallBack','mifunc');
```

donde es necesario crear un fichero llamado ***mifunc.m,*** que será ejecutado al pulsar sobre el botón. En lugar de este fichero también puede ponerse una cadena de caracteres conteniendo distintos comandos, que será evaluada como si se tratase de un fichero ****.m***. Como ejemplo se puede crear un fichero ***mifunc.m*** que contenga sólo la *sentencia disp('Ha pulsado en Start Plot')*.

9.2.2 BOTONES DE SELECCIÓN (*CHECK BOXES*)

Los botones de selección permiten al usuario seleccionar entre varias una o más opciones. Los botones de selección actúan como interruptores indicando un estado ***on*** (si el botón está activado) u ***off*** (si el botón está desactivado). El valor de ambos estados viene definido por las opciones ***Max*** y ***Min***, respectivamente. Por convención los botones de selección deben ser independientes unos de otros.

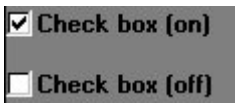


Figura 12. Botones de selección (***Check Boxes***).

Ejemplo: El siguiente conjunto de instrucciones crea una caja con dos opciones, ambas permiten el control de los ejes, de manera independiente, dentro de la función ***plot***:

```
%Definir un texto fijo como título para los botones de selección

txt_axes = uicontrol(gcf,...
   'Style','text',...
   'Position',[0.1 0.4 0.25 0.1],'Units','normalized',...
   'String','Set Axes Properties');
```

```
%Definir la check box para la propiedad Box de los ejes

cb_box = uicontrol(gcf,...
   'Style','checkbox',...
   'Position',[0.1 0.3 0.25 0.1],'Units','normalized',...
   'String','Box=on',...
   'CallBack',['set(gca,''Box'',''off''),',...
      'if get(cb_box,''value'')==1,',...
         'set(gca,''Box'',''on''),',...
      'end']);


% Definir la check box para la propiedad TickDir de los ejes

cb_tdir = uicontrol(gcf,...
   'Style','checkbox',...
   'Position',[0.1 0.2 0.25 0.1],'Units','normalized',...
   'String','TickDir=out',...
   'CallBack',['set(gca,''TickDir'',''in''),',...
      'if get(cb_tdir,''value'')==1,',...
         'set(gca,''TickDir'',''out''),',...
      'end']);
```

9.2.3 BOTONES DE OPCIÓN (*RADIO BUTTONS*)

Al igual que los botones de selección, los botones de opción permiten al usuario seleccionar entre varias posibilidades. La diferencia fundamental reside en que en los botones de opción, las opciones son excluyentes, es decir, no puede haber más de uno activado, mientras que el control anterior permite tener una o más cajas activadas.

Figura 13. Botones de opción (***Radio Buttons***).

Ejemplo: El siguiente ejemplo corresponde a los controles de la figura 13. Estos botones permiten cambiar de dirección los indicadores de los ejes: ***In*** para orientarlos dentro de la figura, o ***Out*** para que se sitúen en el exterior de la gráfica. Se define además otro tipo ***uimenu***, que no es un control propiamente dicho, el texto, que se comenta más adelante.

```
%Definir el texto de título para este grupo de controles

txt_tdir = uicontrol(gcf,...
   'Style','text',...
   'String','Select Tick Direction',...
   'Position',[200 100 150 20]);

%Definir la propiedad TickDir In con radio button (the default)

td_in = uicontrol(gcf,...
   'Style','radio',...
   'String','In',...
   'Position',[200 75 150 25],...
   'Value',1,...
   'CallBack',[...
      'set(td_in,''Value'',1),'...
      'set(td_out,''Value'',0),'...
      'set(gca,''TickDir'',''in''),']);
```

```
%Definir la propiedad TickDir Out con radio button

td_out = uicontrol(gcf,...
   'Style','radio',...
   'String','Out',...
   'Position',[200 50 150 25],...
   'CallBack',[...
      'set(td_out,''Value'',1),'...
      'set(td_in,''Value'',0),'...
      'set(gca,''TickDir'',''out''),']);
```

9.2.4 BARRAS DE DESPLAZAMIENTO (*SCROLLING BARS O SLIDERS*)

Las barras de desplazamiento permiten al usuario elegir un valor entre un rango de valores. El usuario puede cambiar el valor moviendo un indicador, bien mediante la flechas laterales o bien arrastrando el elemento central con el ratón.

Figura 14. Barra de desplazamiento (***slider***).

Ejemplo: El siguiente ejemplo muestra como se utilizan las barras de desplazamiento para mover un sistema de referencia espacial:

```
%Obtener un id de la ventana activa y borrar su contenido

fig = gcf;
clf

%Los callbacks definen la propiedad View de los ejes
%a partir de os valores de las barras de desplaz. (Value property)
%y escriben sus valores en los controles text
%
%Definir la barra para el ángulo de azimut
sli_azm = uicontrol(fig,'Style','slider','Position',[50 50 120 20],...
    'Min',-90,'Max',90,'Value',30,...
    'CallBack',[...
    'set(azm_cur,''String'','...
        'num2str(get(sli_azm,''Val''))),',...
    'set(gca,''View'',',...
    '[get(sli_azm,''Val''),get(sli_elv,''Val'')])']);

%Definir la barra para el ángulo de elevación

sli_elv = uicontrol(fig,...
   'Style','slider',...
   'Position',[240 50 120 20],...
   'Min',-90,'Max',90,'Value',30,...
   'CallBack',[...
   'set(elv_cur,''String'','',...
        'num2str(get(sli_elv,''Val''))),',...
   'set(gca,''View'','',...
   '[get(sli_azm,''Val''),get(sli_elv,''Val'')])']);

%Definir los controles de texto para los valores mínimos

azm_min = uicontrol(fig,...
   'Style','text',...
   'Pos',[20 50 30 20],...
   'String',num2str(get(sli_azm,'Min')));
```

```
elv_min = uicontrol(fig,...
   'Style','text',...
   'Pos',[210 50 30 20],...
   'String',num2str(get(sli_elv,'Min')));

%Definir los controles de texto para los valores máximos

azm_max = uicontrol(fig,...
   'Style','text',...
   'Pos',[170 50 30 20],...
   'String',num2str(get(sli_azm,'Max')),...
   'Horiz','right');

elv_max = uicontrol(fig,...
   'Style','text',...
   'Pos',[360 50 30 20],...
   'String',num2str(get(sli_elv,'Max')),...
   'Horiz','right');

%Definir las etiquetas de las barras

azm_label = uicontrol(fig,...
   'Style','text',...
   'Pos',[50 80 65 20],...
   'String','Azimuth');

elv_label = uicontrol(fig,...
   'Style','text',...
   'Pos',[240 80 65 20],...
   'String','Elevation');

%Definir los controles de texto para los valores actuales
%Los valores son inicializados aquí y
%son modificados por los callbacks de las barras
%cuando el usuario cambia sus valores

azm_cur = uicontrol(fig,...
   'Style','text',...
   'Pos',[120 80 50 20],...
   'String',num2str(get(sli_azm,'Value')));

elv_cur = uicontrol(fig,...
   'Style','text',...
   'Pos',[310 80 50 20],...
   'String',num2str(get(sli_elv,'Value')));
```

9.2.5 CAJAS DE SELECCIÓN DESPLEGABLES (*POP-UP MENUS*)

Las cajas de selección desplegables permiten elegir una opción entre varias mostradas en una lista. Eligiendo una de las opciones, MATLAB realizará la opción elegida. Según la figura 15, el menú se despliega pulsando sobre la flecha de la derecha. La opción sobre la que pase el ratón (en este caso *red*) aparecerá de otro color.

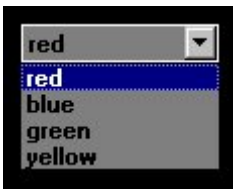


Figura 15. Menú desplegable (***Pop-up menu***).

Ejemplo: el siguiente ejemplo permite cambiar el color de fondo de una ventana:

```
%Definir el menú pop-up

popcol = uicontrol(gcf,...
   'Style','popup',...
   'String','red|blue|green|yellow',...
   'Position',[400 50 120 20],...
   'CallBack',['cb_col = [''R'',''B'',''G'',''Y''];',...
   'set(gcf,''Color'',cb_col(get(popcol,''Value'')))']);
```

9.2.6 CAJAS DE TEXTO (*STATIC TEXT BOXES*)

No son controles, propiamente dichos, ya que no permite realizar ninguna operación con el ratón. Permiten escribir un texto en la pantalla. Una aplicación de este ***uicontrol*** aparece en la variable ***txt_tdir*** del ejemplo del apartado 9.2.3. En este caso particular se el texto indica la función de los botones de opción.

9.2.7 CAJAS DE TEXTO EDITABLES (*EDITABLE TEXT BOXES*)

Se utilizan para introducir y modificar cadenas de caracteres. Puede tener una o más líneas, y la llamada a la opción de ejecución ***Callback*** será efectiva una vez que se apriete las teclas ***Control-Return*** o bien se mueva el ratón fuera del control.

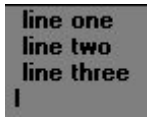


Figura 16. Texto editable (***text***).

Ejemplo: El siguiente ejemplo escribe el texto modificado por el usuario en la ventana de MATLAB.

```
fig=gcf;

edmulti = uicontrol(fig,...
   'Style','edit',...
   'String','Change|these|four|lines',...
   'Position',[10 200 75 100],...
   'Max',2,...
   'CallBack','get(edmulti,''String'')'...
   );
```

9.2.8 MARCOS (*FRAMES*)

Un marco tampoco es un control propiamente dicho. Su función es la de englobar una serie de opciones (botones, cajas de texto, etc.…) con el fin de mantener una estructura ordenada de controles, separando unos de otros en función de las características del programa y del gusto del programador. En la figura 17 pueden observarse dos marcos, en los que se sitúan dos botones.

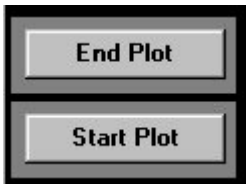


Figura 17. Marcos (***Frames***).

Ejemplo: El ejemplo dibuja el segundo botón que aparece en la figura 17. El otro se realiza de manera análoga, variando el segundo valor de la opción ***position*** en cada ***uimenu***. Al pulsar el botón se realiza una llamada a la función *myplot.m*.

```
pbstart = uicontrol(gcf,...
   'Style','push',...
   'Position',[10 10 100 25],...
   'String','Start Plot',...
   'CallBack','myplot');

ft_dir = uicontrol(gcf,...
   'Style','frame',...
   'Position',[2 2 114 40]);
```

10. Creación de menús

En MATLAB se pueden construir aplicaciones basadas en ventanas, con menús definidos por el usuario. En este apartado se describe cómo crear diferentes menús y submenús. Para crear los controles se utiliza la función ***uimenu***. El aspecto general del comando ***uimenu*** es el siguiente:

```
id_menu = uimenu(id_parent,...
                 'Propiedad1', 'valor1',...
                 'Propiedad2', 'valor2',...
                 'Propiedad3', 'valor3',...
                 'Otras Propiedadades', 'Otros valores');
```

Normalmente una de las propiedades suele ser la propiedad ***CallBack***, aunque no siempre es necesario, ya que algunos menús no ejecutan ningún comando de MATLAB, sino que se encargan de desplegar nuevos submenús. Por lo tanto no es imprescindible definirla en todos los menús.

Para generar submenús basta con crear un menú nuevo, donde el ***id_parent*** sea el identificador del menú padre.

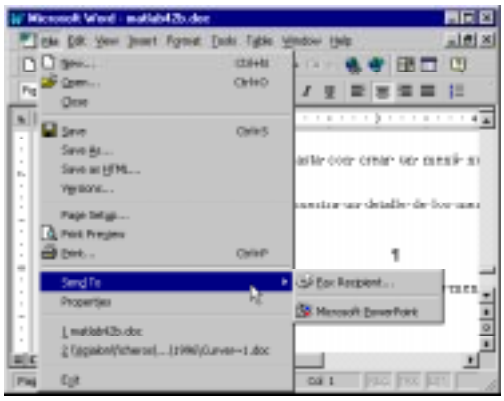


Figura 18. Detalle de los menús de ***Word 97***.

En la figura 18 se muestra un detalle de los menús que componen una aplicación en particular.

10.1 Descripción de las propiedades

A continuación se hace una descripción de las propiedades más importantes, con las que cuenta el comando ***uimenu***.

10.1.1 ACELERADOR (*ACCELERATOR*)

Esta propiedad es opcional y permite definir un carácter con el que activar el menú desde el teclado, sin necesidad de utilizar el ratón. Para activar el menú hay que teclear simultáneamente ALT + el carácter elegido.

Formato:

```
uimenu(id_parent,...
           'Accelerator','carácter',...
       );
```

10.1.2 ACCIÓN A EFECTUAR POR EL MENÚ (*CALLBACK*)

Esta propiedad es muy importante, ya que determina la acción a realizar por MATLAB al actuar sobre el menú correspondiente.

Formato:

```
uimenu(id_parent,...
           'CallBack','string',...
       );
```

10.1.3 CREACIÓN DE SUBMENUS (*CHILDREN*)

Permite crear submenús a partir de menús creados previamente.

Formato:

```
uimenu(id_parent,...
           'Children',vector de handles,...
       );
```

10.1.4 MENÚ ACTIVADO/DESACTIVADO (*ENABLE*)

Esta propiedad permite modificar la posibilidad de acceso al menú correspondiente. A veces resulta interesante que un determinado menú o submenú esté inactivo, porque su acción no tenga sentido en algún momento de la ejecución del programa, (por ejemplo, WORD no permite guardar un fichero, si no hay ninguno abierto).

Formato:

```
uimenu(id_parent,...
           'Enable','on'/'off',...
       );
```

10.1.5 NOMBRE DEL MENÚ (*LABEL*)

Esta propiedad establece el texto correspondiente del menú. Esta opción se puede combinar con la del acelerador, con el fin de señalar al usuario del programa, qué tecla debe pulsar para

que se ejecute el comando. Así, anteponiendo el carácter ***&*** a la letra que se desee, ésta aparece subrayada. Esto no significa que este método sustituya al acelerador, solamente es una indicación para el usuario, que le indica la tecla con la que se activa el menú.

Formato:

```
uimenu(id_parent,...
           'Label','string',...
       );
```

10.1.6 CONTROL DEL OBJETO PADRE (*PARENT*)

Esta opción especifica el ***id*** del objeto padre. Debe ir siempre en primer lugar. El padre de un ***uimenu*** es siempre una **figura** u otro ***uimenu***.

Formato:

```
uimenu(id_parent,...
           'Parent',handle,...
       );
```

10.1.7 POSICIÓN DEL MENÚ (*POSITION*)

En esta opción se suministra un escalar, que determina la posición relativa del menú. Un orden creciente en los menús, equivale a situarlos de izquierda a derecha en la barra de menús. En los submenús ese orden creciente significa que se colocan de arriba hacia abajo.

Formato:

```
uimenu(id_parent,...
           'Position',scalar,...
       );
```

10.1.8 SEPARADOR (*SEPARATOR*)

Esta propiedad permite colocar una línea de separación encima del menú al que se aplica. Cuando se pone a ***on*** se traza la línea y cuando está en ***off*** no se coloca. Por defecto está en ***off***.

Formato:

```
uimenu(id_parent,...
           'Separator', 'on'/'off',...
       );
```

10.1.9 VISIBLE (*VISIBLE*)

Puede tomar dos valores: ***on/off***. Indica si el menú es visible en la ventana o no.

Formato:

```
uimenu(id_parent,...
           'Visible', 'on'/'off',...
       );
```

10.2 Ejemplo de utilización del comando *uimenu*

A continuación se muestra un ejemplo de utilización de los menús de usuario. El ejemplo consta de tres menús (***File***, ***Time*** y ***Curve***) en la barra de menús.

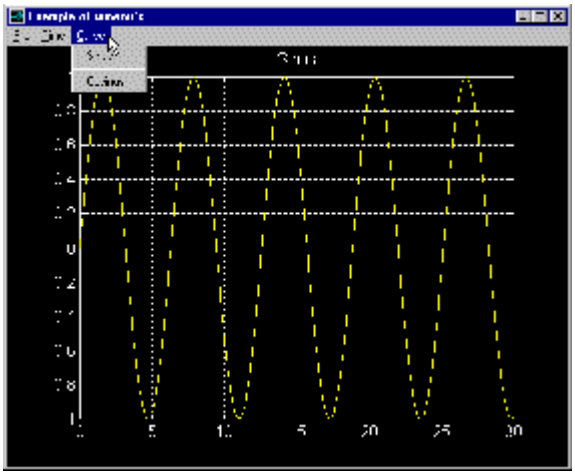


Figura 19. Ejemplo de menús programados por el usuario.

El menú ***File*** incluye tres submenús, ***Load Save*** y ***Exit***. El submenú ***Load*** permite cargar el fichero de variables ***data.mat***. El submenú ***Save*** salva las variables actuales en el fichero ***data.mat***, y el submenú ***Exit***, acaba la ejecución del programa. El menú ***Time*** permite seleccionar entre tres límites para la variable ***t***, que es un vector. El menú ***Curve*** despliega a su vez dos submenús (***Sinus*** y ***Cosinus***), que trazan las gráficas de seno y coseno respectivamente, en función del tiempo seleccionado en el menú ***Time***. En la figura 19 se muestra un aspecto general del programa.

Para realizar esta aplicación, se debe crear en primer lugar la ventana del programa. A continuación se muestra cómo se debe crear una ventana sin ningún elemento en la barra de menús. Los menús se irán incluyendo posteriormente.

```
% Crear una figura sin barra de menús
id_Fig = figure('Units','normalized',...
                'Numbertitle','off',...
                'Name','Example of uimenu´s',...
                'menubar','none');
```

A continuación se colocan todos los menús de la barra de menús.

```
% Creación de los diferentes menús
% Menú File
id_File = uimenu(id_Fig,'Label','&File',...
                 'Accelerator','f');

% Menú Time
id_Time = uimenu(id_Fig,'Label','&Time',...
                 'Accelerator','t');

% Menú Curve
id_Curve = uimenu(id_Fig,'Label','&Curve',...
                 'Accelerator','c');
```

Por último se incluyen todos los submenús de cada uno de los menús.

```
% Creación de los diferentes submenús
% File
id_Load = uimenu(id_File,'Label','&Load',...
                  'Accelerator','L',...
                  'CallBack','load data.mat');
id_Save = uimenu(id_File,'Label','&Save',...
                  'Accelerator','s',...
                  'CallBack','save data.mat');
id_Exit = uimenu(id_File,'Label','E&xit',...
                  'Accelerator','x',...
                  'CallBack','close');
% Time
id_10 = uimenu(id_Time,'Label','10',...
                  'CallBack','t=0:.1:10;');
id_20 = uimenu(id_Time,'Label','20',...
                  'CallBack','t=0:.1:20;');
id_30 = uimenu(id_Time,'Label','30',...
                  'CallBack','t=0:.1:30;');

% Curve
id_Sinus = uimenu(id_Curve,'Label','Sinus',...
                  'CallBack','plot(t,sin(t));grid;title(''Sinus'')');
id_Cosinus = uimenu(id_Curve,'Label','Cosinus',...

'CallBack','plot(t,cos(t));grid;title(''Cosinus'')',...
                  'Separator','on');
```

11. Debugger

11.1 ¿Qué es un *debugger*?

Al programar es inevitable introducir errores. La mayoría de los errores de sintaxis se detectan durante la compilación y son fácilmente resolubles. Pero los errores que se producen durante la ejecución de un programa suelen ser mucho más difíciles de encontrar. Puede ser que el programa compilado ofrezca unos resultados que no son los esperados, que no ofrezca siquiera resultados, o incluso que bloquee el ordenador. En la mayoría de los programas de MATLAB, se utiliza el punto y coma (;) para evitar la impresión por pantalla de los resultados del programa, para así acelerar su ejecución, pero esto hace que el usuario no tenga información sobre las variables que no son resultados finales, sino sólo intermedios.

Para obtener más información sobre esas variables intermedias existen las siguientes posibilidades:

- Borrar todos los puntos y coma de las sentencias, de manera que aparezcan los resultados de todos los cálculos.
- Comentar la línea de declaración de función y convertir el fichero ***.m***, de un fichero de función en un fichero de comandos o *script*, pudiendo visualizar así los resultados en el *espacio de trabajo*[7] del programa y no en el local de la función.

[7] Se entiende por ***espacio de trabajo*** el conjunto de variables que están definidas y son accesibles en un determinado punto del programa. En el caso de una función, son sus variables locales y las variables globales del programa.

– Utilizar el comando ***keyboard***, que permite dentro de un fichero *script*, pasar el control al usuario, de forma que éste pueda –interactivamente, por medio del teclado– examinar las variables que desee. El comando ***keyboard*** debe introducirse en un fichero *.m. Al llegar a ese punto la ejecución se detiene y se devuelve el control al usuario (con un *prompt* doble). El usuario puede examinar o modificar variables. Con el comando ***return*** (tal como está escrito) la ejecución prosigue en la línea siguiente a ***keyboard***.

– Utilizar el ***debugger*** de MATLAB (¡Ni qué decir tiene que ésta es la mejor solución!).

En la mayoría de los casos, los tres primeros métodos son inviables en la práctica, y tienen muchas desventajas respecto al cuarto (es necesario editar el fichero ****.m***, corregirlo y salir cada vez que se detecta un fallo; la información no puede ser solicitada en tiempo de ejecución, ...). Por el contrario, el ***debugger*** es una herramienta especialmente diseñada para detectar y corregir errores.

El ***debugger*** es, en definitiva, una herramienta que permite mantener el control del programa en tiempo de ejecución, siendo unas de sus posibilidades más interesantes la de poder ejecutar el programa paso a paso, deteniéndose en las sentencias más críticas, y la de permitir conocer el valor de todas las variables en cualquier instante de la ejecución. El ***debugger*** sirve principalmente para localizar y corregir errores del programa que no han sido detectados por el compilador.

11.2 Comandos del *debugger*.

El ***debugger*** consiste en un conjunto de comandos adicionales, dirigidos a realizar su tarea de detección y corrección de errores. Estos comandos son fácilmente distinguibles, pues todos comienzan por ***db***. Los principales comandos del ***debugger*** de MATLAB son los siguientes:

dbstop	inserta un *breakpoint* o punto de interrupción en la línea indicada
dbclear	elimina un *breakpoint*
dbcont	continúa con la ejecución detenida
dbstack	lista las llamadas a funciones, es decir qué función llama a qué función
dbstatus	lista los *breakpoints* vigentes
dbstep	ejecuta una o más sentencias
dbtype	lista los ficheros ****.m***, asignando números de línea
dbup	vuelve al espacio de trabajo de la función que ha llamado a la función actual
dbdown	cambia al espacio de trabajo de la función que ha sido llamada
dbquit	sale del modo *debug*

11.3 Utilización del *debugger*.

Cuando se detecta un error en una función en un fichero ****.m*** es conveniente introducir puntos de parada de la ejecución o *breakpoints*. Cuando la ejecución se detiene en un *breakpoint*, aparece el ***prompt*** y se puede introducir cualquier instrucción válida de MATLAB.

Conviene tener en cuenta estas dos características del ***debugger*** de MATLAB:

– El ***debugger*** sólo funciona en ficheros ****.m*** en los que estén definidas funciones, y no en los ficheros de comandos (*scripts*)

– La información de los *breakpoints* del fichero de función ****.m*** está firmemente asociada al fichero ****.m*** compilado. Si se borra el fichero ****.m*** editándolo o mediante el comando ***clear***, toda la información referente a los *breakpoints* de ese fichero también se borra automáticamente.

11.4 Ejemplo de utilización del *debugger*.

Ahora se mostrará cómo se utiliza el ***debugger***, por medio de un ejemplo desarrollado paso a paso. Síganse con atención las siguientes instrucciones:

11.4.1 INICIO DE UNA SESIÓN DE *DEBUGGER*

Para ver cómo se utiliza el ***debugger*** en un caso concreto, considérense los siguientes pasos:

1. Inicie la sesión de ***debugger*** creando los siguientes ficheros, llamados ***test.m*** y ***test1.m***.

```
test.m
     function a=test(b)
     c = sqrt(b)*cos(b);
     a = test1(b,c);

test1.m
     function a=test1(b)
     q = cond(b);
     [w,e] = eig(c);
     a = w*q;
```

2. Utilizando el comando ***dbtype*** puede obtener el listado de estas funciones con los números de cada línea, y puede aprovechar también para ver cómo es una función propia de MATLAB (por ejemplo, la función condición numérica ***cond***).

3. Inserción de *breakpoints*. Se puede insertar un *breakpoint* antes de la primera línea ejecutable de ***test.m*** mediante el comando:

```
dbstop in test
```

   Para insertar un *breakpoint* en la línea 19 de la función ***cond*** hay que teclear:

```
dbstop at 19 in cond
```

11.4.2 EJECUCIÓN EL FICHERO *.M

Para ejecutar el programa con ***debugger***, síganse los siguientes pasos:

1. Después de incluir los *breakpoints*, ejecútese ***test*** y compruébese que se detiene en la línea 2.

```
test(magic(3))
2 c=sqrt(b)*cos(b);
```

2. Puede obtenerse una especie de mapa de las llamadas a todas las funciones que ha habido hasta el momento, con el comando ***dbstack***.

3. Para continuar con la ejecución del programa, tecléese ***dbcont***. La ejecución prosigue hasta alcanzar el siguiente *breakpoint*.

4. Se pueden ver cuántas nuevas llamadas a función ha habido hasta ahora con el comando ***dbstack***.

11.4.3 COMPROBACIÓN DEL VALOR DE LAS VARIABLES.

Para conocer el valor de las variables del programa que desee, procédase del siguiente modo:

1. Se pueden conocer las variables que se están utilizando en ese espacio de trabajo tecleando los comandos ***who*** o ***whos***.
2. Para conocer el valor de cada variable en particular es suficiente con teclear su nombre.
3. Con el comando ***dbstep*** el programa avanza únicamente una nueva línea o sentencia.
4. Tecleando nuevamente ***who*** se ve que en la línea 20 ha aparecido una nueva variable en este espacio de trabajo.

11.4.4 CAMBIO DE ESPACIO DE TRABAJO.

Pruebe a realizar las siguientes tareas:

1. Se puede ir al espacio de trabajo de la función que llamó a ***cond*** tecleando ***dbup***.
2. Se pueden conocer las variables que son accesibles desde este espacio de trabajo tecleando ***who***, y el valor de cada una de ellas tecleando su nombre.

11.4.5 CREACION DE UNA NUEVA VARIABLE.

Una de las cosas que se puede hacer con el ***debugger*** es crear una nueva variable. Esto se hace del siguiente modo:

1. Tecleando new_var=123 se introduce una nueva variable.
2. Se puede ver que esta variable también aparece cuando se teclea ***who*** o ***whos***.

A partir de aquí está invitado a cambiar de espacio de trabajo, a conocer el valor de variables o a avanzar cuanto quiera en la ejecución del programa.

11.4.6 FIN DEL PROCESO DE *DEBUG*.

Para terminar la ejecución del ***debugger*** y volver al modo normal, hay que hacer lo siguiente:

1. Cuando crea que ya ha encontrado el error, puede finalizar la sesión tecleando ***dbquit***.
2. Puede darse cuenta de que los *breakpoint* siguen vigentes si teclea ***dbstatus test***.
3. Para borrar los *breakpoints* utilice ***dbclear***.