UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERÍA MECÁNICA

Departamento Académico de Ciencias Básicas, Humanidades y cursos complementarios

METODOS NUMERICOS (MB –536)

Introducción a Matlab

Profesores:

Garrido Juárez, Rosa

Castro Salguero, Robert

Hermes Pantoja, Carhuavilca

Obregón Ramos, Máximo

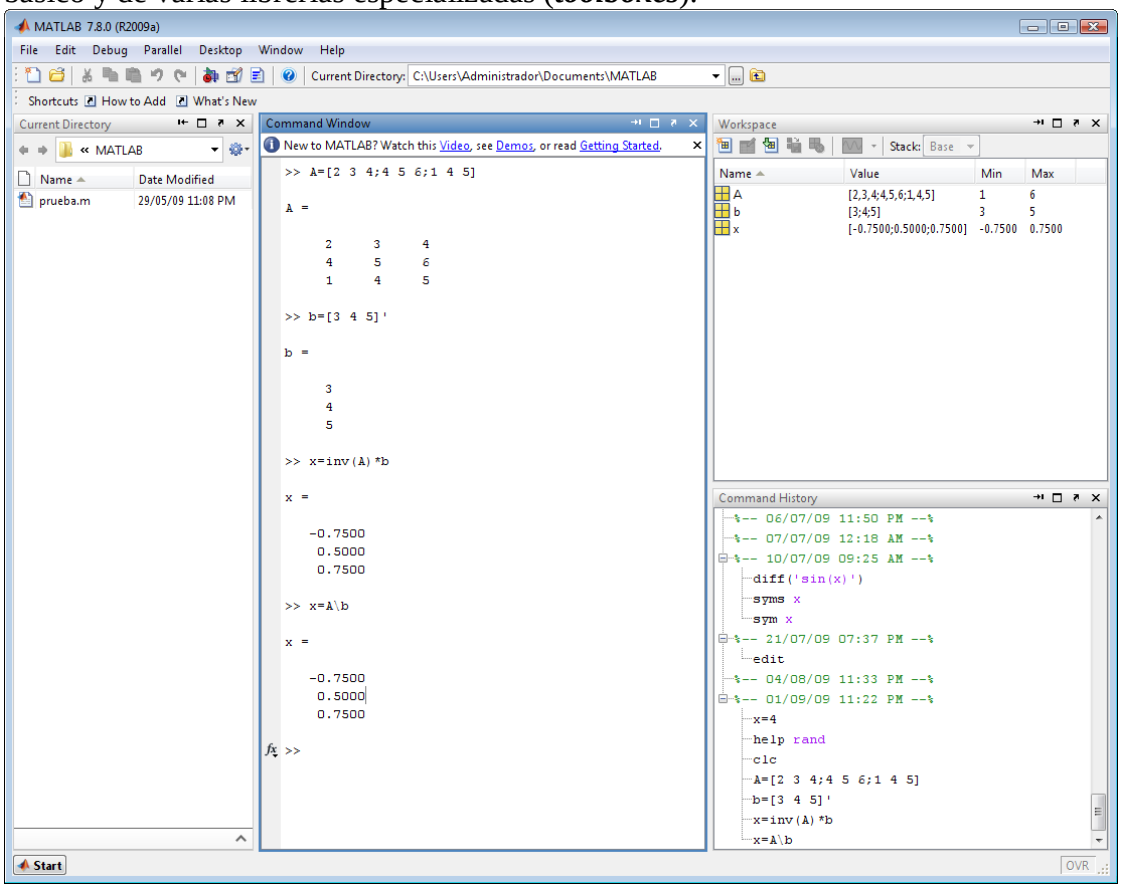
Ruiz Lizama, Edgar

2011-1

MATLAB

MATLAB es el nombre abreviado de "**MAT**rix **LAB**oratory". 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, con cadenas de caracteres y con otras estructuras de información más complejas. 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 es un gran programa de cálculo técnico y científico. Para ciertas operaciones es muy rápido, cuando puede ejecutar sus funciones en código nativo con los tamaños más adecuados para aprovechar sus capacidades de vectorización. En otras aplicaciones resulta bastante más lento que el código equivalente desarrollado en C/C++ o Fortran. A partir de la versión 6.5, MATLAB incorporó un ***acelerador JIT*** (Just In Time), que mejoraba significativamente la velocidad de ejecución de los archivos ****.m*** en ciertas circunstancias, por ejemplo cuando no se hacen llamadas a otros archivos ****.m***, no se utilizan estructuras y clases, etc. Aunque limitado en ese momento, cuando era aplicable mejoraba sensiblemente la velocidad, haciendo innecesarias ciertas técnicas utilizadas en versiones anteriores como la ***vectorización*** de los algoritmos. En cualquier caso, el lenguaje de programación de MATLAB siempre es una magnífica herramienta de alto nivel para desarrollar aplicaciones técnicas, fácil de utilizar y que, como ya se ha dicho, aumenta significativamente la productividad de los programadores respecto a otros entornos de desarrollo. MATLAB dispone de un código básico y de varias librerías especializadas (***toolboxes***).



Pantalla inicial del matlab 2009a

Otra característica 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)')

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 801 valores reales 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.

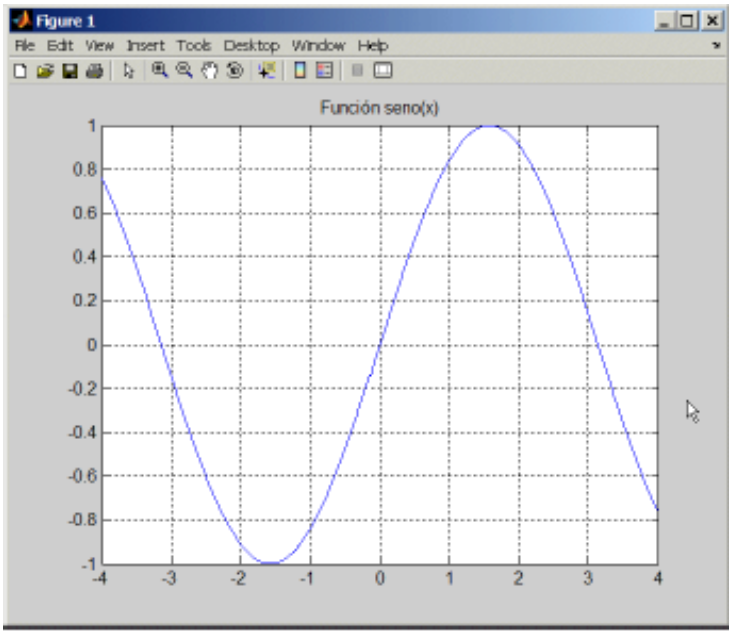


Figura 3: Grafico de la función seno

Un pequeño aviso antes de seguir adelante. Además de con la ***Command History***, es posible recuperar comandos anteriores de MATLAB y moverse por dichos comandos con el ratón y 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 comandos con las teclas ← y →, ir al principio de la línea con la tecla ***Inicio***, al final de la línea con ***Fin***, y borrar toda la línea con ***Esc***. Recuérdese que sólo hay una línea activa (la última).

Para borrar todas las salidas anteriores de MATLAB y dejar limpia la ***Command Window*** se pueden utilizar las funciones ***clc*** y ***home***.
La función ***clc*** (*clear console*) elimina todas las salidas anteriores, mientras que ***home*** las mantiene, pero lleva el ***prompt*** (>>) a la primera línea de la ventana.
Si se desea salir de MATLAB basta teclear los comandos ***quit*** o ***exit***, elegir ***Exit*** MATLAB en el menú ***File*** o utilizar cualquiera de los medios de terminar una aplicación en ***Windows***.

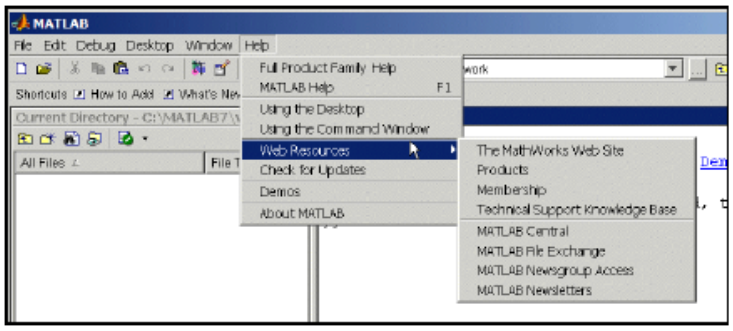


Figura 4: Menu Help de Matlab

GUARDAR VARIABLES Y ESTADOS DE UNA SESIÓN: Comandos *save* y *load*

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 existe el comando ***save***. Si se teclea:

>> save

antes de abandonar el programa, se crea en el ***directorio actual*** un archivo binario llamado ***matlab.mat*** (o ***matlab***) 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 archivos con nombre especificado por el usuario. Por ejemplo, el comando (sin comas entre los nombres de variables):

>> save filename A x y

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

>> load filename Si no se indica ninguna variable, se guardan todas las variables creadas en esa sesión.

OPERACIONES CON MATRICES Y VECTORES

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.

Definición de matrices desde teclado

Como en casi todos los lenguajes de programación, en MATLAB las matrices y vectores son ***variables*** que tienen ***nombres***.

Por el momento se sugiere que se utilicen letras ***mayúsculas para matrices*** y letras ***minúsculas para vectores y escalares*** (MATLAB no exige esto, pero puede resultar útil).

Para definir una matriz *no hace falta declararlas o 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 (o se utilizan). ***Las matrices se definen o introducen 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 (3×3):

>> 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 ***matriz traspuesta***.

En MATLAB el apóstrofe (') es el símbolo de ***transposición 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 se accede a los elementos de un vector poniendo el índice entre paréntesis (por ejemplo **x(3)** ó **x(i)**). Los elementos de las matrices se acceden poniendo los dos 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 antes), y teniendo en cuenta esto ***puede accederse a cualquier elemento de una matriz con un sólo subíndice***. Por ejemplo, si **A** es una matriz (3×3) 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 con que se muestra el resultado se puede cambiar con el menú ***File/Preferences/General***):

```
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
```

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 transpuesto de **y**:

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

MATLAB considera ***vectores fila por defecto***, como se ve en el ejemplo siguiente:

```
>> x(1)=1, x(2)=2
x =
        1
x =
        1 2
```

A continuación se van a estudiar estos temas con un poco más de detenimiento.

OPERADORES ARITMÉTICOS

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.
.* producto elemento a elemento.
./ y .\ división elemento a elemento.
.^ elevar a una potencia elemento a elemento.

Estos operadores se aplican también a las variables o valores escalares, aunque con algunas diferencias.

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 operando escalar y otro matricial. En este caso la operación con el escalar se aplica a cada uno de los elementos de la matriz.

Nota: x = inv(A)*b es lo mismo que x = A\b

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
```

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*) y, desde la versión 5.0, también con otros tipos de datos: ***Matrices de más dos dimensiones***, ***matrices dispersas***, ***vectores y matrices de celdas***, ***estructuras*** y ***clases y objetos***. Algunos de estos tipos de datos más avanzados se verán en la última parte de este manual.

Los elementos constitutivos de vectores y matrices son números reales almacenados en 8 bytes (53 bits para la mantisa y 11 para el exponente de 2; entre 15 y 16 cifras decimales equivalentes). Es importante saber cómo trabaja MATLAB con estos números y los casos especiales que presentan.

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***.

Como ya se ha comentado, por defecto MATLAB trabaja con variables de punto flotante y doble precisión (double). Con estas variables pueden resolverse casi todos los problemas prácticos y con frecuencia no es necesario complicarse la vida declarando variables de tipos distintos, como se hace con cualquier otro lenguaje de programación. Sin embargo, en algunos casos es conveniente declarar variables de otros tipos porque puede ahorrarse mucha memoria y pueden hacerse los cálculos mucho más rápidamente.

```
>> i=int32(100); % se crea un entero de 4 bytes con valor 100
>> j=zeros(100); i=int32(j); % se crea un entero i a partir de j
>> i=zeros(1000,1000,'int32'); % se crea una mariz 1000x1000 de enteros
```

Las funciones **intmin('int64')** e **intmax('int64')** permiten por ejemplo saber el valor del entero más pequeño y más grande (en valor algebraico) que puede formarse con variables enteras de 64 bits:
>> disp([intmin('int64'), intmax('int64')])
-9223372036854775808 9223372036854775807

La función **isinteger(i)** devuelve 1 si la variable i es entera y 0 en otro caso. La función **class(i)** devuelve el tipo de variable que es **i** (***int8***, ***int16***, ...), mientras que la función **isa(i, 'int16')** permite saber exactamente si la variable **i** corresponde a un entero de 16 bits.
MATLAB dispone de dos tipos de variables reales o *float*: ***single*** y ***double***, que ocupan respectivamente 4 y 8 bytes. Por defecto se utilizan ***doubles***. Las funciones **single(x)** y **double(y)** permiten realizar conversiones entre ambos tipos de variables.
Las funciones **realmin** y **realmax** permiten saber los números ***double*** más pequeño y más grande (en valor absoluto) que admite el computador. Para los correspondientes números de simple precisión habría que utilizar **realmin('single')** y **realmax('single')**. La función **isfloat(x)** permite saber si **x** es una variable real, de simple o doble precisión. Para saber exactamente de qué tipo de variable se trata se pueden utilizar las funciones **isa(x, 'single')** ó **isa(x, 'double')**. Obsérvese el ejemplo siguiente, en el que se ve cómo con variables ***single*** se reduce el tiempo de CPU y la memoria:

>> n=1000; AA=rand(n); **A=single(AA)**;
>> tic, **Bs=inv(A)**; toc
Elapsed time is **1.985000** seconds.
>> tic, **Bd=inv(AA)**; toc
Elapsed time is **4.296000** seconds.

Quizás las variables más interesantes –aparte de las variables por defecto, las ***double***– sean las variables logical, que sólo pueden tomar los valores ***true*** (1) y ***false*** (0). Las variables lógicas surgen como resultado de los operadores relacionales (==, <, <=, >, >=, ~=, …) y de muchas funciones lógicas como **any** y **all** que se aplican a vectores y matrices
La función **logical(A)** produce una variable lógica, con el mismo número de elementos que **A**, con valores 1 ó 0 según el correspondiente elementos de **A** sea distinto de cero o igual a cero.
Una de las aplicaciones más importantes de las variables lógicas es para separar o extraer los elementos de una matriz o vector que cumplen cierta condición, y operar luego selectivamente sobre dichos elementos. Obsérvese, el siguiente ejemplo:

```
>> A=magic(4)
A =
        16 2 3 13
        5 11 10 8
        9 7 6 12
        4 14 15 1
>> j=A>10
j =
        1 0 0 1
        0 1 0 0
        0 0 0 1
        0 1 1 0
>> isa(j,'logical')
ans =
        1
>> A(j)=-10
A =
        -10 2 3 -10
        5 -10 10 8
        9 7 6 -10
```

4 -10 -10 1

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: primero, asignando su resultado a una variable,

variable = expresión

y segundo evaluando simplemente el resultado del siguiente modo,

expresión

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 63 letras y números. La función ***namelengthmax*** permite preguntar al programa por este número máximo de caracteres. El carácter guión bajo (_) 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. Esta misma información se puede obtener gráficamente con el ***Workspace Browser***, que aparece con el comando ***View/Workspace*** o activando la ventana correspondiente si estaba abierto.

El comando ***clear*** tiene varias formas posibles:

clear	sin argumentos, ***clear*** elimina todas las variables creadas previamente (exceptolas variables globales).
clear A, b	borra las variables indicadas.
clear global	borra las variables globales.
clear functions	borra las funciones.
clear all	borra todas las variables, incluyendo las globales, y las funciones.

NÚMEROS COMPLEJOS: FUNCIÓN *COMPLEX*

En muchos cálculos matriciales los datos y/o los resultados no son reales sino ***complejos***, con ***parte real*** y ***parte imaginaria***. 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):

```
>> 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 % En este caso el * es necesario
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.

MATLAB dispone asimismo de la función ***complex***, que crea un número complejo a partir de dos argumentos que representan la parte real e imaginaria, como en el ejemplo siguiente:

```
>> complex(1,2)
ans =
```

1.0000 + 2.0000i

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 la matriz conjugada (***conj( )***) y el operador punto y apóstrofo (**.'**) que calcula simplemente la matriz traspuesta.

CADENAS DE CARACTERES

MATLAB puede definir variables que contengan cadenas de caracteres. En MATLAB las cadenas de texto van entre apóstrofos o comillas simples (Nótese que en C van entre comillas dobles: "cadena").

Por ejemplo, en MATLAB:

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.

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.

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 (4×4)
zeros(3,5)	forma una matriz de *ceros* de tamaño (3×5)
zeros(4)	ídem de tamaño (4×4)
ones(3)	forma una matriz de *unos* de tamaño (3×3)
ones(2,4)	idem de tamaño (2×4)
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 **pi9**, 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 (3×3)
rand(2,5)	idem de tamaño (2×5)
randn(4)	forma una matriz de números aleatorios de tamaño (4×4), con distribución normal, de valor medio 0 y varianza 1.
magic(4)	crea una matriz (4×4) 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 (5×5). 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
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)

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))	ídem 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**
blkdiag(A,B)	crea una matriz diagonal de submatrices a partir de las matrices que se le pasan como argumentos
triu(A)	forma una matriz triangular superior a partir de una matriz **A** (no tiene por qué ser cuadrada). Con un segundo argumento puede controlarse que se mantengan o eliminen más diagonales por encima o debajo de la diagonal principal.
tril(A)	ídem 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 $m \times n$ 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 $m \times n$ 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 (6×6) se forma por composición de cuatro matrices de tamaño (3×3). 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.

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

OPERADORES LÓGICOS

Los operadores lógicos de MATLAB son los siguientes:

&	*and* (función equivalente: **and(A,B)**). Se evalúan siempre ambos operandos, y el resultado es ***true*** sólo si ambos son ***true***.
&&	*and* breve: si el primer operando es ***false*** ya no se evalúa el segundo, pues el resultado final ya no puede ser más que ***false***.
|	*or* (función equivalente: **or(A,B)**). Se evalúan siempre ambos operandos, y el resultado es ***false*** sólo si ambos son ***false***.
||	o*r* breve: si el primer operando es ***true*** ya no se evalúa el segundo, pues el resultado final no puede ser más que ***true***.
~	*negación lógica* (función equivalente: **not(A)**)
xor(A,B)	realiza un "or exclusivo", es decir, devuelve 0 en el caso en que ambos sean 1 ó ambos sean 0.

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 6×6 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

Recuérdese que 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

Para acceder a la última fila o columna puede utilizarse la palabra ***end***, en lugar del número correspondiente.
Por ejemplo, para extraer la sexta fila (la última) de la matriz:

>> A(end, :)
ans =
4 36 29 13 18 11

El siguiente comando 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 6×6 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, tales como la siguiente10:

>> 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:

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

MATRIZ VACÍA A[ ]. BORRADO DE FILAS O COLUMNAS

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.

FUNCIONES MATEMÁTICAS ELEMENTALES QUE OPERAN DE MODO ESCALAR

Estas funciones, que comprenden las funciones matemáticas trascendentales y otras funciones básicas, cuando se aplican a una matriz actúan sobre cada elemento de la matriz como si se tratase de un escalar. Por tanto, 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 -p/2 y +p/2)
atan2(x)	arco tangente (devuelve un ángulo entre -p y +p); 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,y)	resto de la división (2 argumentos que no tienen que ser enteros)
mod(x,y)	similar a ***rem*** (Ver diferencias con el ***Help***)
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 -8
ceil(x)	valor entero más próximo hacia +8
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

Las siguientes funciones ***sólo 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
cumsum(x)	devuelve el vector suma acumulativa de los elementos de un vector (cada elemento del resultado es una suma de elementos del original)
mean(x)	valor medio de los elementos de un vector
std(x)	desviación típica
prod(x)	producto de los elementos de un vector
cumprod(x)	devuelve el vector producto acumulativo 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.

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:

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**
nf = size(A,1)	devuelve el número de filas de **A**
nc = size(A,2)	devuelve el número de columnas de **A**

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 multiplicada 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***.^n***X'** siendo [**X**,**D**]=eig(**A**)

FUNCIÓN LINSOLVE()

La función ***linsolve*** es la forma más eficiente de que dispone MATLAB para resolver sistemas de ecuaciones lineales. A diferencia del operador barra invertida \, esta función no trata de averiguar las características de la matriz que permitan hacer una resolución más eficiente: se fía de lo que le dice el usuario. Si éste se equivoca, se obtendrá un resultado incorrecto sin ningún mensaje de error. Las formas generales de la función ***linsolve*** para resolver **Ax**=**b** son las siguientes:

x = linsolve(A,b)
x = linsolve(A,b,opts)

Obviamente, si **b** es una matriz de segundos miembros, **x** será una mariz de soluciones con el mismo nº de columnas. La primera forma de esta función utiliza la factorización LU con pivotamiento parcial si la matriz **A** es cuadrada, y la factorización QR también con pivotamiento por columnas si no lo es. La función ***linsolve*** da un ***warning*** si la matriz **A** es cuadrada y está mlas

condicionada, o si es rectangular y de rango deficiente. Estos warnings se suprimen si se recoge un segundo valor de retorno **r**, que representa el inverso de la condición numérica si **A** es cuadrada o el rango si no lo es:

```
[x,r] = linsolve(A,b)
```

El argumento opcional ***opts*** representa una estructura por medio de la cual el programador proporciona información sobre las características de la matriz.. Los campos de esta estructura se pueden poner a ***true*** o a ***false***, y son los siguientes: LT (triangular inferior), UT (triangular superior), UHESS (forma de Hessenberg superior), SYM (simétrica), POSDEF (definida positiva), RECT (rectangular general) y TRANSA (se desea resolver T = **A x b**, en lugar de = **Ax b**). Obviamente, no todas estas características son compatibles entre sí; las que lo son se indican en la Tabla siguiente:

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
true	false	false	false	false	true/false	true/false
false	true	false	false	false	true/false	true/false
false	false	true	false	false	false	true/false
false	false	false	true	true	false	true/false
false	false	false	false	false	true/false	true/false

En esta Tabla se observa que, en la actual versión de MATLAB, sólo se admiten matrices simétricas que son al mismo tiempo definidas positivas. Por ultimo, considérense los ejemplos siguientes:

```
>> opts.LT=true; x=linsolve(L,b,opts);
>> clear opts; opts.SYM=true; opts.POSDEF=true; x=linsolve(A,b,opts);
```

Obsérvese que, antes de realizar una nueva ejecución se han borrado las opciones utilizadas en la ejecución anterior.

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***. La función ***isequal(A, B)*** devuelve *uno* si las matrices son idénticas y *cero* si no lo son.

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 el nombre **var** existe como variable, función, directorio, archivo, etc.
isnan(A)	chequea si hay valores ***NaN*** en **A**, devolviendo una matriz de unos y ceros del mismo tamaño que **A**.
isinf(A)	chequea si hay valores ***Inf*** en **A**, devolviendo una matriz de unos y ceros del mismo tamaño que **A**.
isfinite(A)	chequea si los valores de **A** son finitos.
isempty(A)	chequea si un vector o matriz está vacío o tiene tamaño nulo.
ischar()	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)'), :)=[]
```

DETERMINACIÓN DE LA FECHA Y LA HORA

MATLAB dispone de funciones que dan información sobre la ***fecha*** y la ***hora*** actual (la del reloj del ordenador). Las funciones más importantes relacionadas con la fecha y la hora son las siguientes:

clock	devuelve un vector fila de seis elementos que representan el ***año***, el ***mes***, el ***día***,la ***hora***, los ***minutos*** y los ***segundos***, según el reloj interno del computador. Los cinco primeros son valores enteros, pero la cifra correspondiente a los segundos contiene información hasta las milésimas de segundo.
now	devuelve un número (***serial date number***) que contiene toda la información de la fecha y hora actual. Se utiliza como argumento de otras funciones.
date	devuelve la fecha actual como cadena de caracteres (por ejemplo: *24-Aug-2004*).
datestr(t)	convierte el *serial date number* ***t*** en cadena de caracteres con el *día*, *mes*, *año*, *hora*, *minutos* y *segundos*. Ver en los manuales on-line los formatos de cadena admitidos.
datenum()	convierte una cadena ('mes-día-año') o un conjunto de seis números (año, mes, día, horas, minutos, segundos) en *serial date number*.
datevec()	convierte *serial date numbers* o cadenas de caracteres en el vector de seis elementos que representa la fecha y la hora.

calendar()	devuelve una matriz 6×7 con el calendario del mes actual, o del mes y año que se especifique como argumento.
weekday(t)	devuelve el día de la semana para un *serial date number* ***t***.

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.
interp1(xp,yp,x)	calcula el valor interpolado para la abscisa **x** a partir de un conjunto de puntos dado por los vectores **xp** e **yp**.
interp1(xp,yp,x,'m')	como la anterior, pero permitiendo especificar también el método de interpolación. La cadena de caracteres **m** admite los valores 'nearest', 'linear', 'spline', 'pchip', 'cubic' y 'v5cubic'.

FUNCIONES INLINE

MATLAB permite definir funciones a partir de expresiones matemáticas por medio de la función ***inline***. Esta función trata de averiguar inteligentemente cuáles son los argumentos de la función ***inline***, a partir del contenido de la expresión matemática. Por defecto se supone que 'x' es el argummento, aunque es también posible determinarlos explícitamente al llamar a ***inline***. Considérense los siguientes ejemplos:

```
>> f=inline('expresión entre apóstrofos');
>> f=inline('expresión', a1, a2, a3); % los argumentos son 'a1', 'a2', 'a3'
>> f=inline('expresión', N); % los argumentos son 'x', 'P1', ..., 'PN'
```

Las funciones ***inline*** se llaman con el handle (**f** en las sentencias anteriores) seguido de los argumentos
entre paréntesis.

ARCHIVOS DE COMANDOS (*SCRIPTS*)

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

Cuando se ejecuta desde la línea de comandos, las variables creadas por ***file1*** pertenecen al espacio de trabajo base de MATLAB. Por el contrario, si se ejecuta desde una función, las variables que crea pertenecen al espacio de trabajo de la función.

En los archivos de comandos conviene poner los puntos y coma (;) al final de cada sentencia, para evitar una salida de resultados demasiado cuantiosa. Un archivo ****.m*** puede llamar a otros archivos ****.m***, e incluso se puede llamar a sí mismo de modo recursivo. Sin embargo, no se puede hacer ***profile*** de un archivo de comandos: sólo se puede hacer de las funciones.

DEFINICIÓN DE FUNCIONES

FUNCIONES CON NÚMERO VARIABLE DE ARGUMENTOS

Desde la versión 5.0, MATLAB dispone de una nueva forma de pasar a una función un número variable de argumentos por medio de la variable ***varargin***, que es un ***vector de celdas*** que contiene tantos elementos como sean necesarios para poder recoger en dichos elementos todos los argumentos que se hayan pasado en la llamada. No es necesario que ***varargin*** sea el único argumento, pero sí debe ser el último de los que haya, pues recoge todos los argumentos a partir de una determinada posición. Recuérdese que a los elementos de un ***cell array*** se accede utilizando llaves {}, en lugar de paréntesis ().

De forma análoga, una función puede tener un número indeterminado de valores de retorno utilizando ***varargout***, que es también un ***cell array*** que agrupa los últimos valores de retorno de la función.

Puede haber otros valores de retorno, pero ***varargout*** debe ser el último. El cell array ***varargout*** se debe crear dentro de la función y hay que dar valor a sus elementos antes de salir de la función.

Recuérdese también que las variables ***nargin*** y ***nargout*** indican el número de argumentos y de valores de retorno con que ha sido llamada la función. A continuación se presenta un ejemplo sencillo: obsérvese el código de la siguiente función ***atan3***:

```
function varargout=atan3(varargin)
if nargin==1
        rad = atan(varargin{1});
elseif nargin==2
        rad = atan2(varargin{1},varargin{2});
else
        disp('Error: más de dos argumentos')
        return
end
varargout{1}=rad;
if nargout>1
varargout{2}=rad*180/pi;
end
```

MATLAB (y muchos otros lenguajes de programación) dispone de dos funciones, llamadas ***atan*** y ***atan2***, para calcular el arco cuya tangente tiene un determinado valor. El resultado de dichas funciones está expresado en radianes. La función ***atan*** recibe un único argumento, con lo cual el arco que devuelve está comprendido entre –p/2 y +p/2 (entre –90º y 90º), porque por ejemplo un arco de 45º es indistinguible de otro de –135, si sólo se conoce la tangente. La función ***atan2*** recibe dos argumentos, uno proporcional al seno del ángulo y otro al coseno. En este caso ya se pueden distinguir los ángulos en los cuatro cuadrantes, entre –p y p (entre –180º y 180º). La función ***atan3*** definida anteriormente puede recibir uno o dos argumentos: si recibe uno llama a ***atan*** y si recibe dos llama a ***atan2*** (si recibe más da un mensaje de error). Además, ***atan3*** puede devolver uno o dos valores de retorno. Por ejemplo, si el usuario la llama en la forma: **>> a = atan3(1);** devuelve un valor de retorno que es el ángulo en radianes, pero si se llama en la forma: **>> [a, b] = atan3(1,-1);** devuelve dos valores de retorno, uno con el ángulo en radianes y otro en grados. Obsérvese cómo la función ***atan3*** utiliza los vectores de celdas ***varargin*** y ***varargout***, así como el número actual de argumentos ***nargin*** con los que ha sido llamada.

SUB-FUNCIONES

Tradicionalmente MATLAB obligaba a crear un archivo ****.m*** diferente por cada función. El nombre de la función debía coincidir con el nombre del archivo. A partir de la versión 5.0 se introdujeron las ***sub-funciones***, que son funciones adicionales definidas en un mismo archivo ****.m***, con nombres diferentes del nombre del archivo (y del nombre de la función principal) y que ***las sub-funciones sólo pueden ser llamadas por las funciones contenidas en ese archivo***, resultando "invisibles" para otras funciones externas. A continuación se muestra un ejemplo contenido en un archivo llamado ***mi_fun.m***:

```
function y=mi_fun(a,b)
        y=subfun1(a,b);
function x=subfun1(y,z)
        x=subfun2(y,z);
function x=subfun2(y,z)
        x=y+z+2;
```

CREACIÓN DE REFERENCIAS DE FUNCIÓN

Ya se ha comentado que las ***referencias de función*** son un nuevo tipo de datos de MATLAB 6. Una referencia de función se puede crear de dos formas diferentes:

1) Mediante el ***operador @*** ("at" o "arroba")

La referencia a la función se crea precediendo el nombre de la función por el operador @. El resultado puede asignarse a una variable o pasarse como argumento a una función. Ejemplos:

```
fh = @sin;
```

area = quad(@sin, 0, pi);

2) Mediante la función ***str2func***

La función ***str2func*** recibe como argumento una cadena de caracteres conteniendo el nombre de una función y devuelve como valor de retorno la referencia de función. Una de las ventajas de esta función es que puede realizar la conversión de un vector de celdas con los nombres en un vector de referencias de función. Ejemplos:

```
>> fh = str2func('sin');
>> str2func({'sin','cos','tan'})
    ans = @sin @cos @tan
```

Una característica común e importante de ambos métodos es que se aplican solamente al ***nombre de la función***, y no al nombre de la función precedido o cualificado por su ***path***. Además los nombres de función deben tener menos de 31 caracteres.

TOOLBOX SYMBOLICO

Uno de los paquetes más útiles para el cálculo con MATLAB lo constituye el paquete simbólico o Symbolic Math Toolbox, que permite realizar cálculo simbólico avanzado, es decir, se puede prescindir de asignar un numero a una variable y tratarla como una constante genérica.

Por ejemplo, si tratamos de calcular el seno de una variable no numérica:

```
>> y=sin(x)
    ??? Undefined function or variable 'x'.
```

Obtenemos un error, pues MATLAB no conoce el valor de x. Para declarar una variable como simbólica usamos la expresión

```
>> syms x
```

Ahora podemos evaluar expresiones que contengan a esta variable a través del comando simplify.

```
>> y=sin(x); z=cos(x);
>> simplify(y^2+z^2)
    ans =
        1
```

También es posible definir matrices simbólicas del siguiente modo

```
>> M=sym('[a b; c d]')
    M =
        [ a, b]
        [ c, d]
```

y ahora calcular, por ejemplo su determinante, y polinomio característico:

```
>> det(M), poly(M)
    ans =
        a*d-b*c
    ans =
        x^2-x*d-a*x+a*d-b*c
```

Con el paquete simbólico podemos calcular límites, derivadas, integrales impropias, etc. A continuación exponemos diversos ejemplos de estos cálculos.

$$\lim_{x\to 0} \frac{\operatorname{sen}(x)}{x},$$

Para calcular el límite

```
>> limit(sin(x)/x,0)
    ans =
        1
```

$$\lim_{n\to\infty} \frac{n^3+3n^2-2n}{3n^3-1},$$

o un limite en infinito:

```
>> syms n
>> limit((n^3+3*n^2-2*n)/(3*n^3-1),inf)
        ans =
                1/3
```

También es posible definir expresiones simbólicas sin necesidad de declarar la variable previamente,

```
>> f=’exp(z^3)+sin(z)^2’
        f =
        exp(z^3)+sin(z)^2
```

y después es posible, por ejemplo, derivar:

```
>> diff(f)
        ans =
        3*z^2*exp(z^3)+2*sin(z)*cos(z)
```

En algunos casos la lectura de la salida que proporciona MATLAB no es muy legible. El comando pretty genera en ocasiones una visualización mas usual. Si nuestra expresión depende de constantes o más variables, es posible especificar la variable de derivación del siguiente modo:

```
>> f=’a*exp(a+x)/sin(a*x)’
        f =
        a*exp(a+x)/sin(a*x)
>> diff(f,’a’)
        ans =
        exp(a+x)/sin(a*x)+a*exp(a+x)/sin(a*x)-a*exp(a+x)/sin(a*x)^2*cos(a*x)*x
>> pretty(ans)
        exp(a + x)   a exp(a + x)       a exp(a + x) cos(a x) x
        ---------- + --------------   -  -------------------------
         sin(a x)         sin(a x)               2 sin(a x)
```

La integración se lleva a cabo con el comando int. Es posible hacer integrales definidas o indefinidas:

```
>> f=’a*exp(a*x)’;
>> int(f,’x’)
        ans =
        exp(a*x)
>> int(f,’x’,0,1)
        ans =
        exp(a)-1
```

Es importante observar que el cálculo de una integral definida mediante la última sentencia es un cálculo simbólico, de manera que integrales como la siguiente

```
>> int(’exp(x^2)’,’x’,0,1)
        ans =
        -1/2*i*erf(i)*pi^(1/2)
```

no tienen mucho sentido, pues el resultado viene expresado mediante una función extraña, ya que no se conoce una primitiva de la función ex2 . Sin embargo, si acudimos al análisis numérico, MATLAB proporciona varios comandos que permiten obtener aproximaciones numéricas de cualquier integral definida. Por ejemplo, para usar la regla de Simpson tenemos el comando quad que puede actuar sobre funciones o expresiones simbólicas

```
>> f=inline(’exp(x.^2)’);
>> quad(f,0,1)
```

```
        ans =
        1.4627
>> quad('x.*exp(x)',0,1)
        ans =
        1.0000
```

Nótese en ambos casos la necesidad de usar los operadores de multiplicación o exponenciación de la forma .* o .^

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 (y matrices). En el caso más sencillo 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.

FUNCIONES GRÁFICAS 2D ELEMENTALES

MATLAB dispone de cinco 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:

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
legend()	define rótulos para las distintas líneas o ejes utilizados en la figura. Para más detalle, consultar el ***Help***
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:

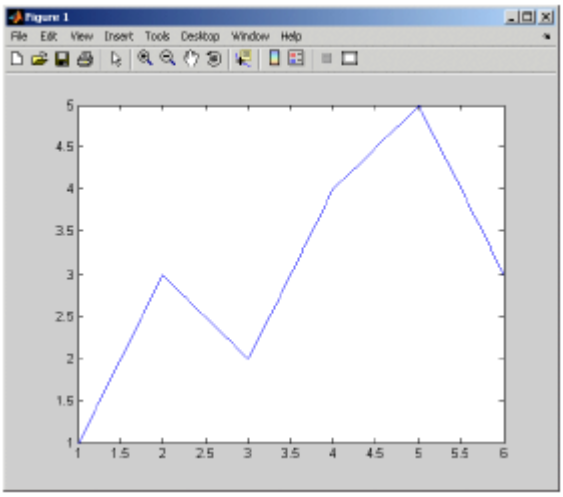
```
>> v = text(1,.0,'seno')
v =
        76.0001
>> delete(v)
```

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 % se cierra la ventana gráfica activa anterior
>> grid % se crea una ventana con una cuadrícula
>> plot(x,y) % se dibuja la función seno borrando la cuadrícula
Se puede observar la diferencia con la secuencia que sigue:
>> close
>> plot(x,y) % se crea una ventana y se dibuja la función seno
>> grid % se añade la cuadrícula sin borrar la función seno
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.

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 el gráfico de la Figura 38. 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 azul. Cuando a la función ***plot()*** se le pasa un único vector –real– como argumento, dicha función 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 que MATLAB utiliza por defecto color blanco para el fondo de la pantalla y otros colores más oscuros para los ejes y las gráficas.
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. Si el usuario no decide otra cosa, para las sucesivas líneas se utilizan colores que son permutaciones cíclicas del ***azul***, ***verde***, ***rojo***, ***cyan***, ***magenta***, ***amarillo*** y ***negro***. 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*** o ***helpwin 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.

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	Marcadores (markers)
y	yellow	.	puntos
m	magenta	o	círculos
c	cyan	x	marcas en x
r	red	+	marcas en +
g	green	*	marcas en *
b	blue	s	marcas cuadradas (square)
w	white	d	marcas en diamante (diamond)
k	black	^	triángulo apuntando arriba
		v	triángulo apuntando abajo
Símbolo	**Estilo de línea**	>	triángulo apuntando a la dcha
-	líneas continuas	<	triángulo apuntando a la izda
:	líneas a puntos	p	estrella de 5 puntas
-.	líneas a barra-punto	h	estrella se seis puntas
--	líneas a trazos		

Cuando hay que dibujar varias líneas, por defecto se van cogiendo sucesivamente los colores de la tabla comenzando por el azul, hacia arriba, y cuando se terminan se vuelve a empezar otra vez por el azul. Si el fondo es blanco, este color no se utiliza para las líneas.
También es posible añadir en la función ***plot*** algunos especificadores de línea que controlan el espesor de la línea, el tamaño de los marcadores, etc. Considérese el siguiente ejemplo:

```
plot(x,y,'-.rs', 'LineWidth',4, 'MarkerEdgeColor','k', 'MarkerFaceColor',
'g',...'MarkerSize',10)
```

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
```

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 subdivisiones 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)

FUNCIÓN LINE()
La función ***line()*** permite dibujar una o más líneas que unen los puntos cuyas coordenadas se pasan como argumentos. Permite además especificar el color, grosor, tipo de trazo, marcador, etc. Es una función de más bajo nivel que la función ***plot()***, pero ofrece una mayor flexibilidad. En su versión más básica, para dibujar un segmento de color verde entre dos puntos, esta función se llamaría de la siguiente manera:
>> line([xini, xend]', [yini, yend]', 'color', 'g')

Se puede también dibujar dos líneas a la vez utilizando la forma:
>> line([xini1 xini2; xend1 xend2], ([yini1 yini2; yend1 yend2]);

Finalmente, si cada columna de la matriz **X** contiene la coordenada *x* inicial y final de un punto, y lo mismo las columnas de la matriz **Y** con las coordenadas *y*, la siguiente sentencia dibuja tantas líneas como columnas tengan las matrices **X** e **Y**:
>> line([X], [Y]);
Se pueden controlar las características de la línea por medio de pares parámetro/valor, como por ejemplo:
>> line(x,y,'Color','r','LineWidth',4,'MarkerSize',12,'LineStyle','—','Marker','*')
CONTROL DE VENTANAS GRÁFICAS: FUNCIÓN FIGURE
Así 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)

La función ***figure*** también admite que se fijen algunas de sus propiedades, como por ejemplo la posición y el tamaño con que aparecerá en la pantalla. Por ejemplo, el comando:

>> figure('position',[left,botton, width,height])

abre una ventana cuya esquina inferior izquierda está en el punto (**left,botton**) respecto a la esquina inferior izquierda de la pantalla (en pixels), que tiene una anchura de **width** pixels y una altura de **height** pixels.

Otra característica muy importante de una ventana gráfica es la de representar animaciones utilizando la técnica del ***doble buffer***. De modo sencillo, esta técnica se puede explicar diciendo que es como si el ordenador tuviera dos paneles de dibujo: mientras uno está a la vista, se está dibujando en el otro, y cuando el dibujo está terminado este segundo panel se hace visible. El resultado del doble buffer es que las animaciones y el movimiento se ven de modo perfecto, sin el ***parpadeo*** (*flicker*) tan característico cuando no se utiliza esta técnica.

Para dibujar con doble buffer en la ventana activa basta ejecutar los comandos siguientes (sin demasiadas explicaciones, que se pueden buscar en el ***Help*** de MATLAB):

>> set(gcf,'DoubleBuffer','on', 'Renderer','painters')

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
barh()	diagramas de barras horizontales
bar3()	diagramas de barras con aspecto 3-D
bar3h()	diagramas de barras horizontales con aspecto 3-D
pie()	gráficos con forma de "tarta"
pie3()	gráficos con forma de "tarta" y aspecto 3-D
area()	similar ***plot()***, pero rellenando en ordenadas de 0 a y
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

Por ejemplo, genérese un vector de valores aleatorios entre 0 y 10, y ejecútense los 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)
```

DIBUJO SIMPLIFICADO DE FUNCIONES: FUNCIONES EZPLOT() Y EZPOLAR()

La función ***ezplot*** es una función de dibujo simplificada, útil cuando se quiere obtener de forma muy rápida la gráfica de una función. En su forma más simple, se puede llamar en la forma:

>> ezplot(f);

donde **f** es el nombre o mejor el handle de una función. También puede ser una ***función inline***. Por defecto la función se dibuja en el intervalo $[-2\pi \leq x \leq 2\pi]$. Si se desea dibijar f en un intervalo diferente, se puede escribir:

```
>> ezplot(f,[a,b]);
```

La función **f** puede ser una ***función implícita de dos variables*** $f(x,y)=0$. El intervalo por defecto para cada variable es $[-2\pi \leq x \leq 2\pi]$. También se puede definir un intervalo común o específico para cada variable.

```
>> ezplot(f); % dibuja f(x,y)=0 en -2*pi<x<2*pi y -2*pi<y<2*pi
>> ezplot(f, [a,b]); % dibuja f(x,y)=0 en a<x<b y a<y<b
>> ezplot(f, [xmin,xmax,ymin,ymax]);
```

La función ***ezplot*** puede dibujar también ***funciones paramétricas*** $x(t)$, $y(t)$, como por ejemplo:

```
>> ezplot('sin(t)','cos(t)'); % dibuja para 0<t<2*pi
>> ezplot('sin(t)','cos(t)', [t1,t2]); % dibuja para t1<t<t2
>> f = inline('cos(x)+2*sin(2*x)'); ezplot(f);
```

La función ***ezpolar*** es similar a ***ezplot*** y se utiliza para dibujar en ***coordenadas polares***.