

Tutorial: Programación en MATLAB

1. Introducción

1.1 Generalidades

El nombre del software MATLAB® proviene de las palabras en Inglés “**MA**Trix **LAB**oratory”. Es un programa muy potente con el cual podremos realizar cálculos numéricos con **vectores** y **matrices**, trabajar con números escalares, tanto reales como complejos y utilizar una amplia variedad de **gráficos** en dos y tres dimensiones. MATLAB tiene un lenguaje propio de programación.

En la ventana inicial aparece el *prompt* característico de MATLAB (»), indicándonos que el programa está preparado para recibir instrucciones. Es posible recuperar comandos anteriores 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 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**.

Para borrar todas las salidas anteriores de MATLAB y dejar limpia la ventana principal se pueden utilizar las funciones **clc** y **home**. 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).

1.2 Uso del **Help** (ayuda)

MATLAB dispone de un excelente **Help** que contiene la información que uno necesita. En la barra de menús, encontramos el menú **Help**, que incorpora varios submenús con distintas opciones.

- **Help Window**, abre una ventana en la que se puede buscar ayuda sobre la función o el concepto que se desee
- **Help Tips**, ofrece ideas prácticas para utilizar la ayuda.
- **Help Desk**, abre un browser (ej.: Internet Explorer) que permite acceder a toda la información sobre MATLAB en formato HTML, equivalente a los manuales impresos del programa. En la parte inferior de la página existe un enlace a manuales en formato *.pdf, **Online Manuals (in PDF)**.
- **Examples and Demos**, abre una ventana que da acceso a un buen número de ejemplos resueltos con MATLAB, cuyos resultados se presentan gráficamente de diversas formas. Es bastante interesante recorrer estos ejemplos para hacerse idea de las posibilidades del programa. Una manera fácil y rápida de obtener información, es a través de la línea de comandos, y se efectúa introduciendo la palabra **help** seguida del nombre del comando del cual se desea obtener información. Veamos un ejemplo:

» help cos

COS Cosine.
COS(X) is the cosine of the elements of X.

Desde la línea de comandos, introducimos la palabra help luego del prompt (») y seguidamente el nombre del comando. Cuando apretamos la tecla Enter, Matlab nos entrega la información solicitada. De la misma manera, utilizando la palabra **doc** seguida de un comando, obtenemos información correspondiente en formato HTML.

1.3 El entorno de trabajo de MATLAB

Los componentes más importantes del entorno de trabajo de MATLAB son el *editor de caminos de búsqueda (Path Browser)*, el *editor y depurador de errores (Editor & Debugger)* y el *visualizador del espacio de trabajo (Workspace Browser)*. A continuación se describen brevemente estos componentes.

1.3.1 PATH BROWSER: Establecer el camino de búsqueda (SEARCH PATH)

MATLAB puede llamar a una gran variedad de funciones, tanto propias como programadas por los usuarios. A veces puede incluso haber funciones distintas que tienen el mismo nombre. Interesa saber cuáles son las reglas que determinan qué función o qué archivo ***.m**¹ es el que se va a ejecutar cuando su nombre aparezca en una línea de

¹ Los archivos ***.m** son archivos ASCII que definen funciones o contienen comandos de MATLAB.

comandos del programa. Esto queda determinado por el *camino de búsqueda* (*search path*) que el programa utiliza cuando encuentra el nombre de una función. El *search path* de MATLAB es una lista de directorios que se puede ver y modificar a partir de la línea de comandos, o utilizando el *Path Browser*. El comando *path* 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 las demás líneas de salida)
```

Para ver cómo se utiliza el *search path* supóngase que se utiliza la palabra *nombre* en un comando. El proceso que sigue el programa para tratar de conocer qué es *nombre*, es el siguiente:

1. Comprueba si *nombre* es una variable previamente definida por el usuario.
2. Comprueba si *nombre* es una *sub-función* o una función *privada* del usuario.
3. Comprueba si *nombre* es una función del usuario.
4. Comprueba si *nombre* es una función interna o intrínseca de MATLAB.
5. Comprueba si hay un archivo llamado *nombre.mex*, *nombre.dll* o *nombre.m* en el *directorio actual*, cuyo contenido se obtiene con el comando *dir*. El *directorio actual* se cambia con el comando *cd*.
6. Comprueba si hay archivos llamados *nombre.mex*, *nombre.dll* o *nombre.m* en los directorios incluidos en el *search path* de MATLAB.

Estos pasos se realizan por el orden indicado. En cuanto se encuentra lo que se está buscando se detiene la búsqueda y se utiliza el archivo que se ha encontrado. Conviene saber que, a igualdad de nombre, los archivos **.mex* tienen precedencia sobre los archivos **.m* que están en el mismo directorio. El concepto de *directorio actual* es importante en MATLAB. Para cambiar de *directorio actual* se utiliza el comando *cd* (de *change directory*). Para subir un nivel en la jerarquía de directorios se utiliza el comando *cd ..*, y *cd ../.* para subir dos niveles. MATLAB permite utilizar tanto la barra normal (/) como la barra invertida (\).

El *Path Browser* es el programa que ayuda a definir la lista de directorios donde MATLAB debe buscar los archivos de comandos y las funciones, tanto del sistema como de usuario. Con el comando *Set Path* del menú *File* aparece un cuadro de diálogo que presenta la lista de directorios en la que MATLAB buscará. Para añadir (o quitar) un directorio a esta lista se debe ejecutar el comando *Add to Path* (o *Remove Path*) en el menú *Path* de dicho cuadro de diálogo. El nuevo directorio se puede añadir al comienzo o final de la lista. El orden de la lista es muy importante, porque refleja el orden de la búsqueda: si una función está en dos directorios, se utilizará la que primero se encuentre.


1.3.2 EDITOR & DEBUGGER: Editor de archivos y depurador de errores

En MATLAB tienen particular importancia los *archivos-M* (o *M-files*). Son archivos de texto ASCII, con la extensión **.m*, que contienen *conjuntos de comandos* o *definición de funciones*. La importancia de estos *archivos-M* es que al teclear su nombre en la línea de comandos y pulsar ENTER, se ejecutan uno tras otro todos los comandos contenidos en dicho archivo. MATLAB dispone de un editor que permite tanto crear y modificar estos archivos, como ejecutarlos paso a paso para ver si contienen errores (proceso de *Debug* o depuración). El *Editor* muestra con diferentes colores los diferentes tipos o elementos constitutivos de los comandos (en *verde* los comentarios, en *rojo* las cadenas de caracteres, etc.). El *Editor* se preocupa también de que las comillas o paréntesis que se abren, no se queden sin el correspondiente elemento de cierre. Para realizar un ejecución del archivo o archivo de comandos controlada con el *Debugger*, se elige el comando *Run* en el menú *Tools* o tecleando el nombre del archivo en la línea de comandos.

Podemos introducir *breakpoints* (puntos en los que se detiene la ejecución de programa); la flecha amarilla indica la sentencia en que está detenida la ejecución. Posicionándonos con el cursor sobre una variable, aparece una pequeña ventana con los valores numéricos de esa variable. El *Debugger* es un programa enormemente útil para detectar y corregir errores. Estas son algunas de las características del depurador, el cual no analizaremos en detalle por el nivel introductorio de este tutorial.

1.3.3 WORKSPACE BROWSER: El espacio de trabajo de MATLAB

El espacio de trabajo de MATLAB (*Workspace*) es el conjunto de variables y de funciones de usuario que en un determinado momento están definidas en la memoria del programa. Para obtener información sobre el *Workspace* se pueden utilizar los comandos *who* y *whos*. El segundo proporciona una información más detallada que el primero.

Se puede obtener de modo gráfico. información análoga utilizando el **Workspace Browser**, que se activa con el comando **Show Workspace** del menú **File** de MATLAB, o clicando en el botón correspondiente de la barra de herramientas ().

1.4 Control de los formatos de salida

Los formatos de salida en la ventana principal de MATLAB se pueden controlar fácilmente a partir del cuadro de diálogo que se abre con el comando **Preferences** del menú **File**. Los formatos numéricos con que MATLAB muestra los resultados, se pueden activar también desde la línea de comandos tecleando algunas de las siguientes posibilidades:

format short: 4 decimales (por defecto, *default*)

format long : 15 decimales

format short e: notación científica con 4 decimales

format long e: notación científica con 15 decimales

...

2. Operaciones con matrices y vectores

MATLAB es fundamentalmente un programa para cálculo matricial. A continuación definiremos las matrices, los vectores y las expresiones que los combinan. 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 casi 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. Se sugiere utilizar letras mayúsculas para matrices y minúsculas para vectores y escalares. 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 (o se utilizan).

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 ENTER 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. Una sencilla operación con **A**, por ejemplo, es hallar su **matriz traspuesta**. En MATLAB el apóstrofo (') es el símbolo de *transposición matricial*. Para calcular **A'** (traspuesta de **A**) basta teclear lo siguiente (se añade a continuación, la respuesta):

```
» 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):

```
» A*B
```

```
ans =
    14    32    50
    32    77   122
    50   122   194
```

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 $A(i,j)$). Cabe destacar que 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 (3x3) se obtiene el mismo valor escribiendo $A(1,2)$ que escribiendo $A(4)$. Invertir una matriz es casi tan fácil como transponerla. A continuación se va a definir una nueva matriz **A** -no singular- en la forma:

```
» A = [2 4 -6; 1 2 5; -3 5 1]
```

```
A =
     2     4    -6
     1     2     5
    -3     5     1
```

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 =
    0.13068    0.19318   -0.18182
    0.090909    0.090909    0.090909
   -0.0625     0.125         0
```

Para comprobar que este resultado es correcto basta *pre-multiplicar* **A** por **B**; el resultado debería ser una matriz identidad (todos unos)

```
» B*A
```

```
ans =
    1.0000    0.0000   -0.0000
    0.0000    1.0000   -0.0000
     0         0         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=[1 4 15]    % vector fila
```

```
x =
     1     4    15
```

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

```
» y=[21; 12; 14] % vector columna
```

```
y =
    21
    12
    14
```

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

```
» x+y
```

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

Estas dificultades desaparecen si se suma \mathbf{x} con el vector traspuesto de \mathbf{y} :

```
» x + y'
```

```
ans =  
    22    16    29
```

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
- .* producto elemento a elemento
- .^ elevar a una potencia elemento a elemento
- ./ y \ división 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. Considérese el siguiente ejemplo:

<pre>» A=[1 2; 3 4]</pre>	<pre>» A*2</pre>	<pre>» A-4</pre>
<pre>A = 1 2 3 4</pre>	<pre>ans = 2 4 6 8</pre>	<pre>ans = -3 -2 -1 0</pre>

Para entender el funcionamiento de los **operadores de división**, consideraremos el sistema de ecuaciones lineales, $\mathbf{Ax}=\mathbf{b}$ (1), en donde \mathbf{x} y \mathbf{b} son vectores columna, y \mathbf{A} una matriz cuadrada invertible. La resolución de este sistema de ecuaciones se puede escribir en las 2 formas siguientes

```
x = inv(A)*b
```

```
x = A\b
```

Así pues, el operador *división-izquierda* por una matriz (barra invertida \) equivale a premultiplicar por la inversa de esa matriz. Aunque no es una forma demasiado habitual, también se puede escribir un sistema de ecuaciones lineales en la forma correspondiente a la traspuesta de la ecuación (1), $\mathbf{yB} = \mathbf{c}$ (2), donde \mathbf{y} y \mathbf{c} son vectores fila (\mathbf{c} conocido). Si la matriz \mathbf{B} es cuadrada e invertible, la solución de este sistema se puede escribir en las formas siguientes:

```
y = c*inv(B)
```

```
y = c/B (3)
```

En este caso, el operador *división-derecha* por una matriz (/) equivale a postmultiplicar por la inversa de la matriz. Si se traspone la ecuación (2) y se halla la solución aplicando el operador *división-izquierda* de obtiene:

```
y' = (B')\c' (4)
```

Comparando las expresiones (3) y (4) se obtiene la relación entre los operadores *división-izquierda* y *división-derecha* (MATLAB sólo tiene implementado el operador *división-izquierda*):

```
c/B = ((B')\c)'
```

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

```
» [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
```

3. Tipos de datos

MATLAB es un programa preparado para trabajar con vectores, matrices, y variables escalares (matrices de dimensión 1), trabaja siempre en doble precisión, es decir guardando cada dato en 8 bytes, con unas 15 cifras decimales exactas. 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 de dos dimensiones, vectores y matrices de celdas, estructuras y clases y objetos.

3.1 Números reales de doble precisión

Los elementos constitutivos de vectores y matrices son números reales almacenados en 8 bytes (53 bytes 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
```

MATLAB representa el infinito como **inf** ó **Inf**, y tiene una representación especial para los resultados que no están definidos como números. Ejecute siguientes comandos y observe 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.

MATLAB dispone de tres funciones útiles relacionadas con las operaciones de coma flotante. Estas funciones, que no tienen argumentos, son las siguientes:

- **eps** devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. Da una idea de la precisión o número de cifras almacenadas. En una PC, eps vale 2.2204e-016.
- **realmin** devuelve el número más pequeño con que se puede trabajar (2.2251e-308)
- **realmax** devuelve el número más grande con que se puede trabajar (1.7977e+308)

3.2 Números complejos

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

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

```
» 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
```

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.

3.3 Cadenas de caracteres

MATLAB puede definir variables que contengan cadenas de caracteres. En MATLAB las cadenas de texto van entre *apóstrofes o comillas simples*. Veamos un ejemplo:

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

La utilización de las cadenas se verá con mayor detalle en secciones posteriores.

4. Variables y expresiones matriciales

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 cualquiera de las formas siguientes:

- asignando su resultado a una variable, **variable = expresión**
- evaluando simplemente el resultado, **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 ENTER. Si se desea que una expresión continúe en la línea siguiente, hay que introducir **tres puntos** (...) antes de pulsar ENTER. 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.

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 31 letras y números. El carácter guión bajo (_) se considera como una letra. 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. Esta misma información se puede obtener gráficamente con el **Workspace Browser**, que aparece con el comando **Show Workspace** del menú **File** o clicando en el



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

- clear** sin argumentos, **clear** elimina todas las variables creadas previamente (excepto las 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.

5. 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. Recuerde 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.

5.1 Tipos de matrices predefinidos

Existen en MATLAB varias funciones orientadas a definir con gran facilidad matrices de tipos particulares. A continuación veremos algunas de estas funciones:

eye (4) matriz unidad de tamaño (4x4)

zeros (3,5) matriz de *ceros* de tamaño (3x5)

zeros (4) ídem de tamaño (4x4)

ones (3) matriz de *unos* de tamaño (3x3)

ones (2,4) ídem 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**², los puntos se generan entre 10^{d1} y **pi**

rand (3) matriz de números aleatorios entre 0 y 1, con distribución uniforme, de tamaño (3x3)

² **pi** es una variable predefinida en MATLAB, que representa el número π .

rand (2,5) idem de tamaño (2x5)

randn (4) 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, ... 4^2 , con la propiedad de que todas las filas, las columnas, y la diagonal principal, suman lo mismo (válido para matrices cuadradas NxN, donde N=1,3,4,5,...)

5.2 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 caminos posibles:

- 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:

<code>[f,c]=size (A)</code>	devuelve el número de filas (f) y de columnas (c) de la matriz A . Si la matriz es cuadrada basta recoger el primer valor de retorno
<code>n=length(x)</code>	calcula el número de elementos de un vector x
<code>zeros(size(A))</code>	forma una matriz de <i>ceros</i> del mismo tamaño que una matriz A previamente creada
<code>ones(size(A))</code>	ídem con <i>unos</i>
<code>A=diag(x)</code>	forma una matriz diagonal A cuyos elementos diagonales son los elementos de un vector ya existente x (da el mismo resultado tanto si el vector es columna o fila)
<code>x=diag(A)</code>	forma un vector x a partir de los elementos de la diagonal de una matriz ya existente A
<code>diag(diag(A))</code>	crea una matriz diagonal a partir de la diagonal de la matriz A
<code>triu(A)</code>	forma una matriz triangular superior a partir de una matriz A (no tiene por qué ser cuadrada)
<code>tril(A)</code>	ídem con una matriz triangular inferior
<code>rot90(A,k)</code>	Gira $k \cdot 90$ grados la matriz rectangular A en sentido antihorario. k es un entero que puede ser negativo. Si se omite, se supone $k=1$
<code>flipud(A)</code>	halla la matriz simétrica de A respecto de un eje horizontal
<code>fliplr(A)</code>	halla la matriz simétrica de A respecto de un eje vertical

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

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 *ENTER* o *puntos y comas*. Los tamaños de las submatrices deben de ser coherentes.

5.3 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 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.

5.4 Operador dos puntos (:)

Este operador es muy importante en MATLAB y puede usarse de varias maneras. Para empezar, definamos 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 transponer 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
```

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 o columna (la última) de la matriz utilizamos:

```
» A(end, :)
```

```
ans =
```

```
4   36   29   13   18   11
```

```
» A(:,end)
```

```
ans =
```

```
24  
25  
20  
15  
16  
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. 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**. La siguiente secuencia de comandos sustituye las filas 2, 4 y 5 de **B** por las filas 1, 2 y 3 de **A**:

```
» A
```

```
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
```

```
» B=eye(size(A))
```

```
B =
```

```
1  0  0  0  0  0
0  1  0  0  0  0
0  0  1  0  0  0
0  0  0  1  0  0
0  0  0  0  1  0
0  0  0  0  0  1
```

```
» 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 complicadas³, tales como la siguientes:

```
» B=eye(size(A));
```

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

```
B =
```

```
0  1  0  0  0  0
1  0  0  0  0  0
0  0  1  0  0  0
0  0  0  1  0  0
0  0  0  0  1  0
0  0  0  0  0  1
```

A continuación veremos la forma de invertir el orden de los elementos de un vector:

```
» x=rand(1,5)
```

³ Se sustituyen las dos primeras filas de **B** por el producto de dichas filas por una matriz de permutación.

```
x =  
    0.9501    0.2311    0.6068    0.4860    0.8913  
  
» x=x(5:-1:1)
```

```
x =  
    0.8913    0.4860    0.6068    0.2311    0.9501
```

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.

5.5 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*. Analice 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.

5.6 Definición de vectores y matrices a partir de un archivo

MATLAB acepta como entrada un archivo **nombre.m** (siempre con extensión **.m**) que contiene instrucciones y/o funciones. Dicho archivo o archivo se invoca desde la línea de comandos tecleando simplemente su nombre, sin la extensión. A su vez, un archivo ***.m** puede llamar a otros archivos ***.m**, e incluso puede llamarse a sí mismo (funciones recursivas). Las variables definidas dentro de un archivo de comandos ***.m** que se ejecuta desde la línea de comandos son variables del *espacio de trabajo base*, esto es, pueden ser accedidas desde fuera de dicho archivo; no sucede lo mismo si el archivo ***.m** corresponde a una función. Si un archivo de comandos se llama desde una función, las variables que se crean pertenecen al espacio de trabajo de dicha función.

Como ejemplo se puede crear un archivo llamado **unidad.m** que construya una matriz unidad de tamaño 3x3 llamada **U33** en un directorio llamado d:\matlab. Este archivo 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 archivos de comandos para poder utilizar el **Debugger** y para evitar teclear muchas veces los mismos datos, sentencias o expresiones.

5.7 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).

6. 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
```

En MATLAB los operadores relacionales pueden aplicarse a vectores y matrices, y eso hace que tengan un significado especial. 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**. Cuando los operadores relacionales 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
```

7. Operadores lógicos

Los operadores lógicos de MATLAB son los siguientes:

```
& and
| or
~ negación lógica
```

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.

8. 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 archivos **.m* y **.mex*⁴ 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. MATLAB 5.2, dispone también de archivos **.p*, que son los archivos **.m pre-compilados* con la función *pcode*. Recuerde que para que MATLAB encuentre una determinada función de usuario el correspondiente *archivo-M* debe estar en el *directorio actual* o en uno de los directorios del *search path*.

8.1 Características generales de las funciones de MATLAB

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 archivos 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 no llevan paréntesis, por lo que a simple vista no siempre son fáciles de distinguir de las simples variables. En la segunda línea de los ejemplos anteriores, *eps* es una función sin argumentos, que devuelve la diferencia entre 1.0 y el número de coma flotante inmediatamente superior. 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. Los nombres de las funciones de MATLAB no son *palabras reservadas* del lenguaje. Es posible crear una variable llamada *sin* o *cos*, que oculten las funciones correspondientes. Para poder acceder a las funciones hay que eliminar (*clear*) las variables del mismo nombre que las ocultan. MATLAB permite que una función tenga un número de argumentos y valores de retorno variable, determinado sólo en tiempo de ejecución. 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.

Algunas características generales de las funciones de MATLAB se enumeran a continuación:

- Los *argumentos actuales*⁵ de estas funciones pueden ser expresiones y también llamadas a otra función.

⁴ Los archivos **.mex* son archivos de código ejecutable.

⁵ Los argumentos actuales son los que se utilizan en la llamada de la 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, 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).

» `[xmax, imax] = max(x)`

» `xmax = max(x)`

- 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.
- Recuerdese que tecleando ***help nombre_funcion*** se obtiene de inmediato información sobre la función de ese nombre.

8.2 Funciones matemáticas elementales que operan de modo escalar

Estas funciones 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:

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

<code>lcm(x)</code>	mínimo común múltiplo
<code>real(x)</code>	partes reales
<code>imag(x)</code>	partes imaginarias
<code>abs(x)</code>	valores absolutos
<code>angle(x)</code>	ángulos de fase

8.3 Funciones que actúan sobre vectores

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

<code>[xm,im]=max(x)</code>	máximo elemento de un vector. Devuelve el valor máximo xm y la posición que ocupa im
<code>min(x)</code>	mínimo elemento de un vector. Devuelve el valor mínimo y la posición que ocupa
<code>sum(x)</code>	suma de los elementos de un vector
<code>cumsum(x)</code>	devuelve el vector suma acumulativa de los elementos de un vector
<code>mean(x)</code>	valor medio de los elementos de un vector
<code>Std(x)</code>	desviación estándar
<code>prod(x)</code>	producto de los elementos de un vector
<code>cumprod(x)</code>	devuelve el vector producto acumulativo de los elementos de un vector
<code>[y,i]=sort(x)</code>	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.

8.4 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:

8.4.1 Funciones matriciales elementales

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

8.4.2 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 $A=XX'$, $\text{expm}(A) = X \cdot \text{diag}(\exp(\text{diag}(D))) \cdot X'$
sqrtm(A)	devuelve una matriz que multiplicada por sí misma da la matriz A
logm(A)	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: $A^n = X \cdot D.^n \cdot X'$ siendo $[X,D]=\text{eig}(A)$

8.4.3 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.
B = inv(A)	calcula la inversa de A . Equivale a $B = \text{inv}(U) \cdot \text{inv}(L)$
d = det(A)	devuelve el determinante d de la matriz cuadrada A . Equivale a $d = \text{det}(L) \cdot \text{det}(U)$
E = rref(A)	reducción a forma de escalón de una matriz rectangular A (mediante eliminación de Gauss con pivotamiento por columnas)

Existen otras funciones pertenecientes a esta categoría, pero dada la brevedad del presente tutorial, se deja a criterio del estudiante el estudio de las mismas utilizando la ayuda de MATLAB.

- 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 ($Ax = \lambda Bx$).

- 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 ($A=U \cdot D \cdot V'$). 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

- Normas de matrices:

<code>norm(A)</code>	norma sub-2, es decir, máximo valor singular de A , $\max(\text{svd}(\mathbf{A}))$
<code>norm(A,1)</code>	norma sub-1 de A , máxima suma de valores absolutos por columnas, es decir: $\max(\text{sum}(\text{abs}(\mathbf{A})))$
<code>norm(A,inf)</code>	norma sub- ∞ de A , máxima suma de valores absolutos por filas, es decir: $\max(\text{sum}(\text{abs}(\mathbf{A}')))$

- Normas de vectores:

<code>norm(x,p)</code>	norma sub-p, es decir $\text{sum}(\text{abs}(\mathbf{x})^p)^{(1/p)}$
<code>norm(x)</code>	norma euclídea; equivale al módulo o $\text{norm}(\mathbf{x},2)$
<code>norm(x,inf)</code>	norma sub- ∞ , es decir $\max(\text{abs}(\mathbf{x}))$
<code>norm(x,1)</code>	norma sub-1, es decir $\text{sum}(\text{abs}(\mathbf{x}))$

8.5 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
```

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. Recuerdese que cualquier valor distinto de cero equivale a **true**, mientras que un valor cero equivale a **false**. Algunas de estas funciones son:

<code>any(x)</code>	chequea si <i>alguno</i> de los elementos del vector x cumple una determinada condición (en este caso ser distinto de cero). Devuelve un uno ó un cero
<code>any(A)</code>	se aplica por separado a cada columna de la matriz A . El resultado es un vector de unos y ceros
<code>all(x)</code>	chequea si <i>todos</i> los elementos del vector x cumplen una condición. Devuelve un uno ó un cero
<code>all(A)</code>	se aplica por separado a cada columna de la matriz A . El resultado es un vector de unos y ceros
<code>find(x)</code>	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
<code>find(A)</code>	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.

9. Gráficos bidimensionales

Después de ver cómo funciona MATLAB, no debe resultar extraño que los gráficos 2-D 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.

9.1 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()	ídem con escala logarítmica en ambos ejes
semilogx()	ídem con escala lineal en el eje de ordenadas y logarítmica en el eje de abscisas
semilogy()	ídem 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:

<code>title('título')</code>	añade un título al dibujo
<code>xlabel('tal')</code>	añade una etiqueta al eje de abscisas. Con <i>xlabel off</i> desaparece
<code>ylabel('cual')</code>	añade una etiqueta al eje de ordenadas. Con <i>ylabel off</i> desaparece
<code>text(x,y,'texto')</code>	introduce 'texto' en el lugar especificado por las coordenadas x e y .
<code>gtext('texto')</code>	introduce texto con ayuda del ratón: el cursor cambia de forma y se espera un clic para introducir el texto en esa posición
<code>Grid</code>	activa la inclusión de una cuadrícula en el dibujo. Con <i>grid off</i> desaparece la cuadrícula
<code>legend()</code>	define r ótulos para las distintas líneas o ejes utilizados en la figura

Observación: cuando utilizamos el comando `text()`, 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

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 todas 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):

```
» x = [-10:0.2:10]; y = sin(x);
» 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.

9.1.1 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)
```

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 (eje x) el valor de los **n** elementos del vector frente a los índices 1, 2, ... **n** del mismo en abscisas (eje y). Más adelante se verá que si el vector es complejo, el funcionamiento es bastante diferente. En la pantalla de su PC 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. Si se quiere dibujar varios vectores complejos, hay que separar explícitamente las partes reales e imaginarias de cada vector. 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

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

Cuando hay que dibujar varias líneas, por defecto se van utilizando 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.

9.1.3 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
```

9.1.4 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 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 añadir 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)
```

9.1.5 Control de los ejes

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:

<code>v=axis</code>	devuelve un vector v con los valores [xmin, xmax, ymin, ymax]
<code>axis(axis)</code>	mantiene los ejes en sus actuales valores, de cara a posibles nuevas gráficas añadidas con hold on
<code>axis('ij')</code>	utiliza <i>ejes de pantalla</i> , con el origen en la esquina superior izda. y el eje j en dirección vertical descendente
<code>axis('xy')</code>	utiliza <i>ejes cartesianos</i> normales, con el origen en la esquina inferior izda. y el eje y vertical ascendente

<code>axis('equal')</code>	el escalado es igual en ambos ejes
<code>axis('square')</code>	la ventana será cuadrada
<code>axis('image')</code>	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
<code>axis('off')</code>	elimina las etiquetas, los números y los ejes
<code>axis('on')</code>	restituye las etiquetas, los números y los ejes
<code>axis('normal')</code>	elimina las restricciones introducidas por 'equal' y 'square'

9.2 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. 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, 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)

```

9.3 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:

<code>bar()</code>	crea diagramas de barras
<code>barh()</code>	diagramas de barras horizontales
<code>bar3()</code>	diagramas de barras con aspecto 3-D
<code>bar3h()</code>	diagramas de barras horizontales con aspecto 3-D
<code>pie()</code>	gráficos con forma de “tarta”
<code>pie3()</code>	gráficos con forma de “tarta” y aspecto 3-D
<code>area()</code>	similar plot() , pero rellenando en ordenadas de 0 a y
<code>stairs()</code>	función análoga a bar() sin líneas internas
<code>errorbar()</code>	representa sobre una gráfica –mediante barras– valores de errores
<code>hist()</code>	dibuja histogramas de un vector
<code>rose()</code>	histograma de ángulos (en radianes)
<code>quiver()</code>	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)

```


9.4 Tipos de funciones gráficas tridimensionales

MATLAB tiene posibilidades de realizar varios tipos de gráficos 3D. 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 archivo **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,'r')
» [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)
```

9.4.1 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, 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.

```
» plot3(X,Y,Z)
```

9.4.2 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 $\sin(r)/r$ (siendo $r=\sqrt{x^2+y^2}$); para evitar dividir por

0 se suma al denominador el número pequeño *eps*). Cree un archivo llamado *sombrero* 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)
```

Se habrá podido comprobar de la ejecución del archivo, 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. Ejecute el comando y observe la diferencia con el anterior. 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.

```
» surf(W)
```

Cree ahora el archivo *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 archivo, teclée *picos* en la línea de comandos y obsérvese el resultado.

9.4.3 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. A continuación se verá cómo se puede utilizar estas representaciones con las matrices de datos *Z* y *W* que se han calculado previamente:

```
» 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:

<code>contour(Z, val)</code>	siendo val un vector de valores para las isolíneas a dibujar
<code>contour(u,v,W,20)</code>	se utilizan u y v para dar valores a los ejes de coordenadas
<code>contour(Z,20,'r-')</code>	se puede especificar el tipo de línea como en la función <i>plot</i>
<code>contourf(Z, val)</code>	análoga a <i>contour()</i> , pero rellenando el espacio entre líneas

9.5 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.

9.5.1 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. El **colormap** por defecto es *jet*. 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; ésa es la razón de que en el ejemplo anterior a la función **pcolor** se le pasen 65 filas y 2 columnas). Si se desea imprimir una figura en blanco y negro, puede utilizarse el mapa de color **gray**. 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.

9.5.2 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)
```

En las figuras resultantes, 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).

9.5.3 Formas paramétricas de las funciones MESH, SURF Y PCOLOR

Existen unas formas más generales de las funciones **mesh**, **surf** y **pcolor**. 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**, ejecútase el siguiente comando y obsérvese que se obtiene el mismo resultado que anteriormente:

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

La principal ventaja de estas funciones 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*.