PRÁCTICAS DE MATEMÁTICAS II
INGENIERÍA INFORMÁTICA
CURSO ACADÉMICO 2006-2007

Introducción a MATLAB.

1 Trabajando con vectores en MATLAB

Esta es la introducción básica a MATLAB: la definición de vectores y una serie de operaciones elementales.

El comienzo es sencillo: para iniciar MATLAB, en Unix/Linux abrimos una terminal en nuestro sistema y tecleamos: matlab. En Windows, cliqueamos en el icono correspondiente o seleccionamos en el Menú de inicio.

En el texto que sigue a continuación, cualquier línea que comienza con dos signos >> se utiliza para denotar una línea de comando MATLAB.

Casi todos los comandos básicos en MATLAB implican el uso de vectores. Para simplificar la creación de vectores, podemos definir un vector especificando: una primera entrada, un incremento y una última entrada. Por ejemplo, para crear un vector cuyas entradas son 0, 2, 4, 6 y 8, podemos teclear:

```
>> 0:2:8

ans =

     0     2     4     6     8
```

MATLAB también guarda el último resultado. En el ejemplo previo, se ha creado una variable "ans". Para obtener el vector traspuesto, tecleamos:

```
>> ans'

ans =

     0
```

```
     2
     4
     6
     8
```

Para ser capaz de guardar los vectores creados, podemos darles nombre. Por ejemplo, para crear el vector fila v, tecleamos:

```
>> v = [0:2:8]

v =

     0     2     4     6     8

>> v

v =

     0     2     4     6     8

>> v;
>> v'

ans =

     0
     2
     4
     6
     8
```

Podemos darnos cuenta del ejemplo anterior que si finalizamos una línea con un punto y coma, no se muestra el resultado. MATLAB permite también trabajar con elementos específicos del vector. Si, por ejemplo, queremos quedarnos sólo con las tres primeras entradas de un vector:

```
>> v(1:3)
```

```
ans =

     0     2     4

>> v(1:2:4)

ans =

     0     4

>> v(1:2:4)'

ans =

     0
     4
```

Una vez especificada la notación podemos realizar diversas operaciones:

```
>> v(1:3)-v(2:4)

ans =

    -2    -2    -2
```

2 Matrices en MATLAB

Damos a continuación una introducción básica a la definición y manipulación de matrices. La definición de una matriz es análoga a la definición de un vector. Podemos considerarla como una columna de vectores fila (los espacios son necesarios!):

```
>> A = [ 1 2 3; 3 4 5; 6 7 8]

A =
```

```
     1     2     3
     3     4     5
     6     7     8
```

o como una fila de vectores columna:

```
>> B = [ [1 2 3]' [2 4 7]' [3 5 8]']

B =

     1     2     3
     2     4     5
     3     7     8
```

(de nuevo, es importante incluir los espacios.)

Si hemos estado haciendo estas pruebas con vectores, tendremos muy probablemente una gran cantidad de variables definidas. Si queremos conocer esta información, el comando **whos** nos permitirá cuáles son las variables que tenemos en nuestro espacio de trabajo.

```
>> whos
              Name        Size       Elements     Bytes    Density   Complex

                A       3 by 3              9        72       Full      No
                B       3 by 3              9        72       Full      No
              ans       1 by 3              3        24       Full      No
                v       1 by 5              5        40       Full      No
```

La notación utilizada en MATLAB es la notación usual en álgebra lineal. De modo que, por ejemplo, la multiplicación de matrices en MATLAB se hace de forma sencilla. Debemos tener cuidado con las dimensiones de las matrices a la hora de multiplicarlas (deben tener el tamaño adecuado!.)

```
>> v = [0:2:8]
```

```
v =

     0     2     4     6     8

>> A*v(1:3)
??? Error using ==> *
Inner matrix dimensions must agree.

>> A*v(1:3)'

ans =

    16
    28
    46
```

Podemos trabajar con diferentes partes de una matriz, al igual que vimos que se podía hacer con vectores. De nuevo, debemos tener cuidado de hacer operaciones "legales":

```
>> A(1:2,3:4)
???  Index exceeds matrix dimensions.

>> A(1:2,2:3)

ans =

     2     3
     4     5

>> A(1:2,2:3)'

ans =

     2     4
     3     5
```

Una vez que somos capaces de crear y manipular una matriz, podemos realizar muchas operaciones habituales con ella. Podemos, por ejemplo, obtener la inversa de una matriz. Sin embargo, debemos tener cuidado puesto que las operaciones que se realizan pueden presentar errores de redondeo. En el ejemplo, la matriz A no es una matriz invertible, pero MATLAB devuelve una matriz como resultado.

```
>> inv(A)

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 4.565062e-18


ans =

   1.0e+15 *

   -2.7022    4.5036   -1.8014
    5.4043   -9.0072    3.6029
   -2.7022    4.5036   -1.8014
```

Conviene hacer notar, en este punto, que MATLAB distingue entre mayúsculas y minúsculas. Esta es otra potencial fuente de problemas cuando trabajamos con algoritmos complicados:

```
>> inv(a)
??? Undefined function or variable a.
```

Otra posible operación es, por ejemplo, la obtención de los valores propios aproximados de una matriz. Hay dos versiones de esta rutina: una encuentra los valores propios y la otra encuentra los valores y vectores propios. Si no recordamos cuál es cuál, podemos obtener más información tecleando **eig** en la línea de comandos de matlab.

```
>> eig(A)

ans =
```

```
   14.0664
   -1.0664
    0.0000

>> [v,e] = eig(A)

v =

   -0.2656    0.7444   -0.4082
   -0.4912    0.1907    0.8165
   -0.8295   -0.6399   -0.4082


e =

   14.0664         0         0
         0   -1.0664         0
         0         0    0.0000

>> diag(e)

ans =

   14.0664
   -1.0664
    0.0000
```

Existen también rutinas que permiten encontrar soluciones de ecuaciones. Por ejemplo, si $Ax = b$ y queremos encontrar x, un modo "lento" de hacerlo es, simplemente, invertir A y realizar una multiplicación por la izquierda sobre ambos lados de la ecuación. Obviamente, hay métodos más eficientes y más estables para hacer esto (descomposiciones L/U con pivotes, por ejemplo). MATLAB tiene comandos especiales para hacer esto. MATLAB posee además dos tipos diferentes de operadores / y \. La acción del primer operador es la siguiente: $x = A\backslash v \equiv A^{-1}v$; la acción del segundo operador es: $x = v/B \equiv vB^{-1}$. Se dan ejemplos de su uso a continuación:

```
>> v = [1 3 5]'

v =

     1
     3
     5

>> x = A\v

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 4.565062e-18


x =

   1.0e+15 *

    1.8014
   -3.6029
    1.8014

>> x = B\v

x =

     2
     1
    -1

>> B*x

ans =

     1
     3
     5
```

```
>> x1 = v'/B

x1 =

    4.0000   -3.0000    1.0000


>> x1*B

ans =

    1.0000    3.0000    5.0000
```

Finalmente, si queremos borrar todos los datos del sistema y comenzar de nuevo utilizaremos el comando **clear**. Cuidado!: MATLAB no pide una segunda opinión ...

```
>> clear

>> whos
```

3 Funciones de vectores

Es indudable que la gran ventaja de trabajar con MATLAB es la facilidad de manipulación de vectores y matrices. En este apartado, comenzaremos con manipulaciones simples (suma, resta, multiplicación). A continuación mostramos cómo se pueden definir operaciones relativamente complejas con un pequeño esfuerzo.

Comenzamos con la suma y resta de vectores. Definiremos dos vectores y a continuación los sumaremos y restaremos:

```
>> v = [1 2 3]'

v =
```

```
     1
     2
     3

>> b = [2 4 6]'

b =

     2
     4
     6

>> v+b

ans =

     3
     6
     9

>> v-b

ans =

    -1
    -2
    -3
```

La multiplicación de vectores y matrices sigue, lógicamente, reglas estrictas, así como la suma. En el ejemplo anterior, los vectores son ambos vectores columna con tres entradas:

```
>> v*b
Error using ==> *
Inner matrix dimensions must agree.

>> v*b'
```

```
ans =

     2     4     6
     4     8    12
     6    12    18

>> v'*b

ans =

    28
```

Hay ocasiones en las que queremos realizar una operación sobre cada entrada de un vector o matriz. MATLAB permite hacer este tipo de operaciones. Por ejemplo, supongamos que queremos multiplicar cada entrada de un vector v con la entrada correspondiente al vector b. En otras palabras, Supongamos que queremos hallar v(1)*b(1), v(2)*b(2) y v(3)*b(3). Estaría bien utilizar el símbolo ***** puesto que estamos haciendo un tipo de multiplicación. Sin embargo, como este símbolo ha sido definido con otra función, debemos recurrir a otra cosa. Los programadores ocupados del desarrollo de MATLAB decidieron entonces utilizar los símbolos **.*** para hacer esta operación. De hecho, se puede emplear este símbolo antes de cualquier símbolo matemático para especificar a MATLAB que la operación en cuestión debe tener lugar sobre cada entrada del vector.

```
>> v.*b

ans =

     2
     8
    18

>> v./b

ans =

    0.5000
```

```
    0.5000
    0.5000
```

Puesto que hemos comenzado a hablar de operaciones no lineales, continuemos con ellas: si pasamos un vector a una operación matemática predefinida, obtendremos un vector del mismo tamaño con entradas obtenidas realizando la operación especificada sobre la correspondiente entrada del vector original:

```
>> sin(v)

ans =

    0.8415
    0.9093
    0.1411

>> log(v)

ans =

         0
    0.6931
    1.0986
```

La posibilidad de trabajar con estas funciones vectoriales es una de las ventajas de MATLAB. De este modo, podemos definir operaciones complejas rápida y fácilmente. En el siguiente ejemplo trabajamos con un vector con muchas componentes:

```
>> x = [0:0.1:100]

x =

  Columns 1 through 7

         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
```

```
    (stuff deleted)

  Columns 995 through 1001

   99.4000   99.5000   99.6000   99.7000   99.8000   99.9000  100.0000
```

```
>> y = sin(x).*x./(1+cos(x));
```

Además de esta simple manipulación de vectores, MATLAB nos permitirá también representar gráficamente los resultados que obtengamos. Tecleando,

```
>> plot(x,y)
```

tendremos una representación gráfica de la función antes considerada. Si tecleamos

```
>> plot(x,y,'rx')
```

obtenemos la misma gráfica pero las líneas son reempladas por puntos rojos en forma de x. Para ver más opciones del commando plot, podemos teclear

```
>> help plot
```

El comando help es, sin duda, el camino más corto para estar seguro de la sintaxis de un determinado comando de Matlab.

La notación compacta permitirá realizar un gran número de operaciones utilizando pocos comandos. Veamos el siguiente ejemplo:

```
>> coef = zeros(1,1001);
>> coef(1) = y(1);
>> y = (y(2:1001)-y(1:1000))./(x(2:1001)-x(1:1000));
>> whos
             Name        Size       Elements     Bytes    Density   Complex

              ans       3 by 1             3        24       Full      No
                b       3 by 1             3        24       Full      No
             coef       1 by 1001       1001      8008       Full      No
                v       3 by 1             3        24       Full      No
                x       1 by 1001       1001      8008       Full      No
```

```
                y       1 by 1000       1000      8000       Full      No

Grand total is 3011 elements using 24088 bytes

>> coef(2) = y(1);
>> y(1)

ans =

    0.0500

>> y = (y(2:1000)-y(1:999))./(x(3:1001)-x(1:999));
>> coef(3) = y(1);
>>
>>
```

A partir de este algoritmo podemos encontrar el polinomio de Lagrange que interpola los puntos definidos anteriormente (vector x). Por supuesto, con tantos puntos el proceso puede resultar algo tedioso. Afortunadamente, MATLAB dispone de un modo sencillo de hacer tareas monótonas, como veremos a continuación.

4 Bucles

En esta sección veremos cómo utilizar los bucles **for** y **while**. En primer lugar, discutiremos el uso del bucle **for** con ejemplos para operaciones fila sobre matrices. A continuación, mostraremos el uso del bucle **while**.

El bucle **for** permite repetir ciertos comandos. Todas las estructuras de bucles en matlab comienzan con una palabra clave (como **for** o **while**) y terminan con un **end** (parece sencillo, ¿no?). Veamos un ejemplo trivial:

```
>> for j=1:4,
   j
end


j =
```

```
     1


j =

     2


j =

     3


j =

     4

>>
```

Otro ejemplo:

```
>> v = [1:3:10]

v =

     1     4     7    10

>> for j=1:4,
     v(j) = j;
end
>> v

v =

     1     2     3     4
```

Éste es un ejemplo simple y una demostración bonita de cómo funcionan los bucles **for**. Sin embargo, no se debe utilizar en la práctica: la notación

utilizada en la primera línea es mucho más rápida que el bucle. Un mejor ejemplo se presenta a continuación, donde realizamos operaciones sobre las filas de una matriz. Si queremos comenzar en la segunda fila de una matriz y restar la fila previa y repetir esta operación sobre las siguientes filas, un bucle **for** puede ocuparse de esto:

```
>> A = [ [1 2 3]' [3 2 1]' [2 1 3]']

A =

     1     3     2
     2     2     1
     3     1     3

>> B = A;
>> for j=2:3,
    A(j,:) = A(j,:) - A(j-1,:)
end

A =

     1     3     2
     1    -1    -1
     3     1     3


A =

     1     3     2
     1    -1    -1
     2     2     4
```

Presentamos a continuación un ejemplo más realista (implementación de eliminación Gaussiana):

```
>> for j=2:3,
     for i=j:3,
        B(i,:) = B(i,:) - B(j-1,:)*B(i,j-1)/B(j-1,j-1)
```

```
     end
   end

B =

     1     3     2
     0    -4    -3
     3     1     3


B =

     1     3     2
     0    -4    -3
     0    -8    -3


B =

     1     3     2
     0    -4    -3
     0     0     3
```

La forma general de un bucle **while** es

```
>> while (condiciones)
    (operaciones)
   end
```

Las operaciones se repetirán mientras que las condiciones sean ciertas. Por ejemplo, dado un número a, el siguiente bloque permite calcular y mostrar el entero no negativo más pequeño tal que $2^n \geq a$:

```
>> n=0;
>> while 2^n < a
    n=n+1;
   end
>> n
```

4.1 Relaciones

Los operadores de relación en MATLAB son:

$<$	menor que
$>$	mayor que
$<=$	menor o igual que
$>=$	mayor o igual que
$==$	igual que
$\sim=$	distinto a

Démonos cuenta que el símbolo "==" se utiliza en lugar de "=" en una relación. Podemos conectar varias relaciones utilizando los operadores lógicos:

&	y
|	o
$\sim$	no

5 La instrucción if

La forma general de una instrucción **if** simple es:

```
>> if (condiciones)
    (operaciones)
   end
```

Las operaciones se realizarán únicamente si se cumplen las condiciones especificadas. Se admiten las ramificaciones múltiples como puede verse en el siguiente ejemplo:

```
>> if n <0
     a=1;
   elseif n<5
     a=2;
   else
     a=3;
   end
```

6 Ficheros ejecutables

En esta sección introducimos los conceptos básicos para crear ficheros ejecutables. Los ficheros ejecutables son ficheros de texto que incluyen una serie de comandos Matlab. Si una tarea Matlab la vamos a ejecutar muchas veces, es buena idea escribir un fichero con estos comandos para poder ejecutarlos tantas veces como queramos.

La edición del fichero ejecutable la realizamos con un editor cualquiera. Si nos resulta más cómodo, podemos utilizar el editor que incorpora MATLAB y al que invocaremos desde la línea de comandos como:

```
>>edit
```

Los ficheros ejecutables de MATLAB (llamados ficheros M) deben tener como extensión ".m". En el ejemplo que damos a continuación, crearemos un programa que calcula el factorial de 6:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Este es un programa no muy util,
%que calcula el factorial de 6
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
n=6;
fac=1;
for i=2:n
   fac=fac*i;
end
fac
```

Si guardamos esto en el fichero **fac.m** en el directorio de trabajo (o cualquier otro incluido en el "path") y tecleamos el comando **fac**, obtenemos

```
>> fac

fac =

   720
```

Las lineas tras el símbolo "%" son líneas de comentario, que se conviene utilizar como explicación del programa. El comando **help** sirve para mostrar esas líneas:

```
>> help fac

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 Este es un programa no muy util,
 que calcula el factorial de 6
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

En efecto, este no es un programa muy útil, en primer lugar porque el propio MATLAB tiene su comando para calcular el factorial de números enteros:

```
 factorial(6)

ans =

   720
```

y en segundo lugar porque sólo calcula el factorial de 6. Para poder calcular el factorial para distintos números deberemos crea una subrutina o función MATLAB.

7 Subrutinas

Si ahora escribimos en un fichero con nombre **facf.m** los siguientes comandos

```
%Esta es una funcion para calcular
%el factorial de n.
%El valor de entrada es n
function fac=facf(n)
fac=1;
for i=2:n
   fac=fac*i;
end
```

habremos definido una función que podemos utilizar tal como lo hacemos con los comandos intrínsecos de MATLAB. Por ejemplo, tecleando en la linea de comandos **facf(6)** tenemos:

```
>> facf(6)

ans =

   720
```

Las funciones pueden tener varios parámetros de entrada y/o salida. Por ejemplo, la siguiente es una función que, dados dos vectores con la misma longitud, devuelve dos valores (es decir, la subrutina implementa una función $f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^2$).

```
%Esta es una funcion que, dados
%dos vectores de la misma longitud
%calcula dos valores reales a y b
%a=sin(x*y'), b=a*x*x
function [a,b]=funci(x,y)
a=sin(x*y');
b=a*x*x';
```

Guardamos este fichero como **funci.m** y, como prueba, ejecutamos en la línea de comandos:

```
>> x=1:1:5

x =

     1     2     3     4     5

>> y=0:0.1:0.4

y =

         0    0.1000    0.2000    0.3000    0.4000

>> [f1,f2]=funci(x,y);
```

```
>> f1

f1 =

   -0.7568

>> f2

f2 =

  -41.6241
```

Por supuesto, si las matrices de entrada x e y no son vectores de la misma longitud, el programa puede fallar. En el siguiente ejemplo, la primera llamada a **funci** es correcta, pero no así la segunda:

```
>> [a,b]=funci(1:1:5,0:1:4);
>> [a,b]=funci(1:1:5,0:1:5);
??? Error using ==> *
Inner matrix dimensions must agree.

Error in ==> c:\matlab\temporal\funci.m
On line 6  ==> a=sin(x*y');
```

Importante:

– Fijémonos que la sintaxis de las rutinas es:

```
function [output1,output2,...]=nombre(input1,input2,...)
```

– Si queremos crear un fichero independiente que contenga a la rutina, el nombre que asignemos al fichero debe ser el mismo que el de la rutina con extensión ".m".

Por supuesto, las funciones así definidas pueden ser utilizadas tantas veces como sea necesario dentro de otras funciones y programas.

Nuestra norma será escribir TODOS los programas en ficheros de texto (nunca con Word, por favor) y procurar que estos sean lo más estructurados posibles, utilizando subrutinas (funciones) para implementar tareas independientes. Se trata de programar de la forma más modular y estructurada posible.

8 Cadenas de texto, mensajes de error, entradas

Se pueden introducir cadenas de texto en MATLAB si van entre comillas simples. Por ejemplo, la instrucción

```
>> s='Mi asignatura preferida es Calculo Numerico'
```

asigna a la variable *s* la anterior cadena de texto.

Se pueden mostrar cadenas de texto utilizando la función **disp**. Por ejemplo:

```
>> disp('En particular, me encantan las practicas de la asignatura')
```

Los mensajes de error pueden (deben) mostrarse utilizando la función **error**

```
>> error('Fue un error no acabar las memorias a tiempo')
```

puesto que cuando se utiliza en un fichero ".m", interrumpe la ejecución del mismo.

Podemos, asimismo, en un fichero ".m" pedir que un usuario introduzca datos de forma interactiva utilizando la función **input**. Cuando, por ejemplo, durante la ejecución de un programa aparece la línea

```
>>ifaltas =input('Cuantas veces has faltado a practicas?')
```

el citado mensaje de texto aparece en pantalla y la ejecución del programa se interrumpe hasta que el usuario teclea el dato de entrada. Después de presionar la tecla de "return",el dato es asignado a la variable "ifaltas" y se reanuda la ejecución del programa.

9 Comparando la eficiencia de algoritmos: flops y cputime

Dos medidas de la eficiencia de un algoritmos son el número de operaciones de punto flotante realizadas y el tiempo de CPU transcurrido.

La función **flops** de MATLAB proporciona la información sobre el número de operaciones de punto flotante realizadas. El comando **flops(0)** reinicializa este número a 0. Por tanto, tecleando **flops(0)** inmediatamente antes de la ejecución de un algoritmo y **flops** inmediatamente después, obtenemos la citada información.

El tiempo de cpu transcurrido tras la ejecución de un algoritmo puede obtenerse utilizando el comando **cputime**. De hecho, la secuencia

```
>> cpu1=cputime, (cualquier conjunto de operaciones), cpu2=cputime-cpu1
```

nos proporcionará el tiempo de cpu invertido en la ejecución del mencionado conjunto de operaciones.

Nota: en las recientes versiones de MATLAB se ha eliminado el comando flops.

10 Formatos de salida

Mientras que todos los cálculos en MATLAB se realizan en doble precisión, el formato de los datos de salida puede ser controlado con los siguientes comandos:

format short	Punto fijo y 4 decimales (es el que hay por defecto)
format long	Punto fijo y 14 decimales
format short e	notación científica con 4 decimales
format long e	notación científica con 14 decimales

11 Gráficos

Aunque ya hemos mencionado anteriormente la utilización del comando **plot**, vamos a dar en esta sección algún detalle adicional sobre las posibilidades gráficas de MATLAB.

MATLAB permite generar representaciones gráficas de curvas en 2D y 3D. Los comandos básicos con los que nos manejaremos serán **plot**, **plot3**, **mesh** y **surf**.

11.1 Representaciones en 2D

El comando **plot** crea gráficos de curvas en el plano x-y; si x e y son vectores de la misma longitud, el comando **plot(x,y)** abre una ventana gráfica y dibuja en el plano x-y los elementos de x versus los elementos de y. Podemos, por ejemplo, dibujar el gráfico de la función seno en el intervalo -4 a 4 con los siguientes comandos:

```
x=-4:.01:4; y=sin(x); plot(x,y)
```

El vector x es una partición del dominio en intervalos de longitud 0.01; el vector y es un vector que proporciona los valores del seno en los nodos de la partición.

Como segundo ejemplo vamos a dibujar el gráfico de $y = e^{-x^2}$ en el intervalo -1.5 a 1.5:

```
x=-1.5:.01:1.5; y=exp(-x.^2); plot(x,y)
```

Démonos cuenta de que la operación de elevar al cuadrado está precedida por un punto para que opere sobre cada una de las componentes del vector x.

MATLAB posee además la utilidad **fplot** para representar de forma eficiente y simple el gráfico de una función: para representar el ejemplo anterior podemos, de forma alternativa, definir la función en un fichero M (que llamaremos, por ejemplo, expcu.m):

```
function y=expcu(x)
y=exp(-x.^2)
```

Entonces el comando

```
fplot('expcu',[-1.5,1.5])
```

nos proporcionará el gráfico en cuestión.

Podemos generar también gráficos de curvas definidas paramétricamente. Por ejemplo,

```
t=0:.001:2*pi; x=cos(3*t); y=sin(2*t); plot(x,y)
```

De forma complementaria, podemos asignar a los gráficos: títulos, etiquetas en los ejes y texto (en la zona del gráfico). Para ello utilizaremos los comandos

title	título del gráfico
xlabel	etiqueta del eje x
ylabel	etiqueta del eje y
gtext	sitúa texto sobre el gráfico utilizando el ratón
text	sitúa texto en las coordenadas especificadas

Ejemplo, el comando:

```
title('Hola caracola')
```

proporciona al gráfico el título en cuestión.

Los ejes del gráfico son, por defecto, autoescalados. Para modificar esto podemos utilizar el comando **axis**:

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

El comando **axis** debe utilizarse después de **plot**.

Es posible realizar distintas representaciones en un mismo gráfico. Por ejemplo,

```
x=0:.01:2*pi;y1=sin(x); y2=sin(2*x); y3=sin(4*x); plot(x,y1,x,y2,x,y3)
```

Otra posibilidad es utilizar **hold**. El comando **hold on** "congela" la terminal gráfica en la que estamos trabajando, de modo que se pueden superponer diversos gráficos en ella. Los ejes pueden, sin embargo, ser reescalados. El comando **hold off** "descongela" la terminal gráfica.

Dentro de las opciones gráficas podemos elegir el tipo de línea, el tipo de punto y el color. Por ejemplo,

```
x=0:.01:2*pi;y1=sin(x); y2=sin(2*x); y3=sin(4*x);
plot(x,y1,'--',x,y2,':',x,y3,'+')
```

dibuja una línea discontinua y punteada, respectivamente, para los dos primeros gráficos mientras que el tercer gráfico se muestra con el símbolo "+". Los tipos de línea y marca son:

Tipos de línea: sólida (-), discontinua (–), punteada (:), discontinua y punteada (-.)

Tipos de marca: punto (.), mas (+), estrella (*), círculo (o), x (x)

Se pueden especificar colores para los distintos tipos de línea y marca:

Colores: amarillo (y), magenta (m), rojo (r), verde (g), azul (b), blanco (w), negro (k)

El comando **subplot** puede utilizarse para hacer una partición de la terminal gráfica, de modo que pueden situarse varios subgráficos en una misma figura.

11.2 Representaciones en 3D

11.2.1 Gráficos de línea

El comando **plot3** en 3 dimensiones es el análogo al comando **plot** en 2 dimensiones: produce curvas en el espacio tridimensional. Si x, y y z son vectores del mismo tamaño, entonces el comando **plot3(x,y,z)** producirá un gráfico de perspectiva de la curva en el espacio tridimensional que pasa por los puntos especificados por x, y y z. Estos vectores suelen estar definidos de forma paramétrica. Por ejemplo,

```
t=.01:.01:2*pi; x=cos(t); y=sin(t); z=t.^3; plot3(x,y,z)
```

proporciona una hélice que está comprimida cerca del plano x-y.

11.2.2 Gráficos de malla y de superficie

Pueden obtenerse gráficos tridimensionales mallados de superficies utilizando el comando **mesh**. Si tecleamos **mesh(z)** obtenemos un gráfico de perspectiva tridimensional de los elementos de la matriz z. La superficie representada está definida por las coordenadas z de los puntos sobre un retículo rectangular en el plano x-y. El siguiente ejemplo muestra un gráfico de estas características de los elementos de la matriz identidad 10×10 (comando **eye(10)**):

```
mesh(eye(10))
```

Análogamente pueden obtener gráficos tridimensionales "compactos" de superficies utilizando el comando **surf**:

```
surf(eye(10))
```

Para dibujar el gráfico de una función $z = f(x, y)$ sobre un rectángulo debemos, en primer lugar, definir vectores xx e yy que proporcionan particiones sobre los lados del rectángulo. Con la función **meshgrid** creamos una matriz x, cada fila de la misma contiene las componentes del vector xx y cuyo número de columnas es igual a la longitud del vector yy. De manera análoga creamos la matriz y, cuyas columnas contienen las componentes del vector yy. Esto lo conseguimos con la instrucción:

```
[x,y]=meshgrid(xx,yy);
```

Una vez hecho esto, obtenemos la matriz z haciendo actuar f sobre las matrices x e y. La representación de la matriz z se puede hacer acudiendo a los comandos **mesh** y **surf**.

Veamos un ejemplo:

Vamos a dibujar el gráfico de $z = e^{-x^2-y^2}$ sobre el cuadrado $[-2, 2] \times [-2, 2]$ del siguiente modo:

```
xx=-2:.2:2;
yy=xx;
[x,y]=meshgrid(xx,yy);
z=exp(-x.^2-y.^2);
mesh(z)
```

Las características del comando **axis** introducido previamente son aplicables también a los gráficos tridimensionales, así como lo son las de los comandos para títulos, etiquetado de ejes y el comando **hold**.

El color de las superficies se ajusta utilizando el comando **shading**. Hay 3 opciones para este comando: **faceted** (el que está por defecto), **interpolated** y **flat**. Se accede a estas opciones tecleando:

```
shading faceted, shading interp, shading flat
```

El comando **shading** debe introducirse después del comando **surf**. La utilización de **shading interp** y **shading flat** causa la desparición del mallado en la superficie.

El perfil de colores de una superficie se controla mediante el comando **colormap**. Mapas de colores predefinidos son:

```
hsv (por defecto), hot, cool, jet, pink, copper,
flag, gray, bone
```

Por ejemplo, el comando **colormap(cool)** proporciona un determinado perfil de colores en la figura en cuestión.

12 Resumen de funciones elementales y matrices especiales

En la tabla que se muestra a continuación se presentan algunas funciones de MATLAB que nos pueden ser de utilidad:

abs	valor absoluto o módulo
sqrt	raíz cuadrada
real	parte real
imag	parte imaginaria
conj	complejo conjugado
exp	exponencial
log	logaritmo natural
log10	logaritmo en base 10
sin, asin, sinh, asinh	seno, arcseno, seno hiperb., arcseno hiperb.
cos, acos, cosh, acosh	idem para el coseno
tan, atan, tanh, atanh	idem para la tangente
cot, acot, coth, acoth	idem para la cotangente
sec, asec, sech, asech	idem para la secante
csc, acsc, csch, acsch	idem para la cosecante

Finalmente, algunas matrices especiales de MATLAB:

zeros	matriz de ceros
ones	matriz de unos
eye	identidad
diag	diagonal
rand	números aleatorios distribuidos unif.