Un curso de MATLAB

Grupo FMI
Departamento de Matemática Aplicada, Universidad de Zaragoza
http://www.unizar.es/fmi

Versión actualizada en agosto de 2006

Índice general

Antes de comenzar este curso de MATLAB, donde estudiaremos por separado algunas de las posibilidades de este programa–herramienta–lenguaje, hacen falta unos conocimientos rudimentarios sobre el empleo de MATLAB. Para ello, recomendamos que el lector siga previamente las actividades del documento

Una primera sesión en MATLAB
Grupo FMI, Zaragoza, 2006.

El recorrido por esa primera sesión se pueda realizar en el espacio de dos horas como máximo.

Esta guía ha sido desarrollada dentro del marco del Proyecto de Innovación Docente *Guías de aprendizaje de Matlab para alumnos y profesores* por los siguientes miembros de grupo Formación Matemática en Ingeniería: Francisco Javier Sayas (coordinador), Mercedes Arribas, Natalia Boal, José Manuel Correas, Francisco José Gaspar, Dolores Lerís, Andrés Riaguas y María Luisa Sein-Echaluce.

Capítulo 1

Matrices

MATLAB trabaja esencialmente con matrices de números reales o complejos. Las matrices 1×1 son interpretadas como escalares y las matrices fila o columna como vectores.
Por defecto todas las variables son matriciales y nos podemos referir a un elemento con dos índices. Aún así, *conviene saber que la matriz está guardada por columnas y que nos podemos referir a un elemento empleando sólo un índice, siempre que contemos por columnas.* Insistiremos bastante en este detalle, porque tiene fuertes implicaciones para entender el funcionamiento de bastantes aspectos de MATLAB.

```
>> A=[-2 1;4 7] % matriz dos por dos, introducida por filas
A =
    -2    9
     4    7
>> A(2,1)       % fila dos, columna uno
ans =
     4
>> A(3)         % tercer elemento (se lee por columnas)
ans =
     9
>> A(:)         % forma de ver A tal como MATLAB la guarda
ans =
    -2
     4
     6
     7
```

1.1. Aritmética de las matrices

Con las operaciones suma (+) y producto (*) entre matrices hay que poner atención en que las dimensiones de las matrices sean las adecuadas para realizar dichas operaciones.

```
>> A=[-2 3;-4 5;-6 7]  % o tambien A=[-2, 3;-4, 5;-6, 7]
A =
    -2     3
    -4     5
    -6     7
>> B=[1 1;2 0;6 2];
>> A+B  % matriz + matriz de la misma dimension
ans =
    -1     4
    -2     5
     0     9
>> A'*A  % matriz 2x3 * matriz 3x2
ans =
    56   -68
   -68    83
>> A*A'  % matriz 3x2 * matriz 2x3
ans =
    13    23    33
    23    41    59
    33    59    85
>> a=[3 -1];
>> A*a'  % matriz * vector
ans =
    -9
   -17
   -25
>> u=[1 2 3]
u =
     1     2     3
>> v=[4 5 6]
v =
     4     5     6
>> u*v'  % fila por columna = producto escalar
ans =
    32
>> u'*v  % columna por fila = matriz de rango unidad
ans =
     4     5     6
     8    10    12
    12    15    18
>> cross(u,v)  % producto vectorial
ans =
    -3     6    -3
```

El ejemplo anterior muestra cómo se puede hacer el producto escalar de dos vectores. Si ambos son vectores fila `u*v'` realiza la operación, ya que devuelve una matriz 1×1, que en MATLAB es indistinguible de un escalar (un número). La expresión

```
dot(u,v)                % producto escalar
```

realiza la misma operación (`dot` abrevia el concepto de *dot product*, que es como se suele llamar en inglés al producto escalar), sin preocuparse de si los vectores son fila o columna.
Hay que tener también cuidado con el símbolo de trasposición. Cuando una matriz `A` es compleja, con `A'` se calcula su traspuesta conjugada. Para calcular la traspuesta sin conjugar hay que hacer:

```
A.'                % traspuesta sin conjugar
```

El producto de un vector columna por un vector fila (de cualquier tamaño) produce una matriz genérica de rango unidad:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \dots & b_m \end{bmatrix} = \begin{bmatrix} a_1\,b_1 & a_1\,b_2 & \dots & a_1\,b_m \\ a_2\,b_1 & a_2\,b_2 & \dots & a_2\,b_m \\ \vdots & \vdots & & \vdots \\ a_n\,b_1 & a_n\,b_2 & \dots & a_n\,b_m \end{bmatrix}$$

Hay instrucciones para crear matrices llenas de ceros, llenas de unos, diagonales, o la identidad (en inglés la letra I se pronuncia igual que *eye*):

```
>> ones(3)
A =
     1     1     1
     1     1     1
     1     1     1
>> zeros(3,5)
ans =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
>> eye(2)  % identidad 2x2
ans =
     1     0
     0     1
>> diag([1 3 -4]) % matriz diagonal
ans =
     1     0     0
     0     3     0
     0     0    -4

>> A=[1 2; 3 4];
>> diag(A)  % vector de elementos diagonales de A
```

```
ans =

     1
     4
>> diag(diag(A))  % truco para extraer la diagonal de una matriz
ans =

     1     0
     0     4
```

Finalmente, unos pequeños ejemplos de qué se puede hacer con una matriz cuadrada.

```
>> B=[2 1;1 3];
>> B^2     % potencia al cuadrado de la matriz
ans =
     5     5
     5    10
>> B*B     % producto de matrices
ans =
     5     5
     5    10
>> B.^2   % cuadrado de cada elemento de la matriz
ans =
     4     1
     1     9
>> B.*B     % producto elemento a elemento  igual que B.^2
ans =
     4     1
     1     9
>> det(B)  % determinante de B
ans =
     5
>> rank(B)  % rango de B
ans =
     2

>> Bi=inv(B)   % matriz inversa
Bi =
    0.6000   -0.2000
   -0.2000    0.4000
```

Advertencia. Hay que tener una cierta precaución con la función `rank` para calcular el rango de una matriz, ya que lo hace de forma numérica. Para matrices con enteros de tamaño razonable el resultado

suele ser exacto. En otros casos, MATLAB puede obtener un rango incorrecto por culpa de los redondeos; aún así, MATLAB suele avisar si encuentra una cierta *proximidad* a un defecto de rango.

1.2. Acceso a los elementos de una matriz

Ya hemos indicado que las matrices en MATLAB se guardan por columnas y que se puede buscar un elemento siempre con un único índice. Esta es la simple razón por la que en los vectores (donde no hay más que una dimensión a la hora de colocar los elementos, sea en fila o en columna) no hace falta más que un índice para buscar un elemento.

```
>> A=[2 3;4 5;6 7]
A =
     2     3
     4     5
     6     7
>> A(2,1) % se toma el elemento de la fila 2 y la columna 1 de A
ans =
     4
>> A(2,3)
??? Index exceeds matrix dimensions.
>> A(:,1)   % vector columna con la primera columna de A
ans =
     2
     4
     6
>> A(2,:)   % vector fila con la segunda fila de A
ans =
     4     5
>> A(4)     % cuarto elemento de A, contando por columnas
ans =
     3
>> size(A)  % dimensiones de A (devuelve un vector)
ans =
     3     2
>> length (A)   % mayor de las dimensiones de A
ans =
     3
>> v=[-3 4 7];     % vector fila
>> v(2)            % segundo elemento
ans =
     4
>> v(1,2)          % segundo elemento (primera fila, segunda columna)
ans =
     4
```

La referencia a elementos de una matriz permite cambiar el valor de un elemento mediante una sencilla operación de asignación. Esto también se puede hacer con una fila o columna.

```
>> A=[2 -3;-4 5;6 -7];
>> A(3,1)=1/2   % cambio de un elemento de A
A =
    2.0000   -3.0000
   -4.0000    5.0000
    0.5000   -7.0000
>> A(2,:)=[1 1]
A =
    2.0000   -3.0000
    1.0000    1.0000
    0.5000   -7.0000
```

Al definir un nuevo elemento fuera de las dimensiones de la matriz se reajusta el tamaño de la matriz dando el valor 0 a los restantes elementos

```
>> A(3,4)=1     % asignacion fuera del espacio definido
A =
    2.0000   -3.0000         0         0
   -4.0000    5.0000         0         0
    0.5000   -7.0000         0    1.0000
>> size (A)
ans =
     3     4
```

La matriz **vacía** es la matriz que no tiene ningún elemento. Se escribe entre corchetes (es decir, [ ]) y puede ser muy útil a la hora de borrar filas o columnas de una matriz dada, como se ve en el siguiente ejemplo,

```
>> A=[1 -1 2;2 0 1;0 1 -3];
>> A(:,2)=[]     % borramos la segunda columna
A =
    1   2
    2   1
    0   3
```

MATLAB puede trabajar con grupos de filas y columnas (no necesariamente consecutivos) o concatenar matrices para formar matrices más grandes siempre que los tamaños sean compatibles.

```
>> A = diag([1 2 3]);
>> [A,ones(3,2)]   % ampliar con columnas
ans =
     1     0     0     1     1
     0     2     0     1     1
     0     0     3     1     1
>> [A;eye(3)]      % ampliar con filas
ans =
     1     0     0
     0     2     0
     0     0     3
     1     0     0
     0     1     0
     0     0     1
>> A=[1 3;2 1];
>> B=[A eye(2);zeros(2) A] % matriz formada por 4 bloques 2x2
B =
     1     3     1     0
     2     1     0     1
     0     0     1     3
     0     0     2     1
```

En general, empleando listas implícitas o simplemente vectores de índices, se pueden extraer submatrices de una matriz, incluso repitiendo filas y columnas.

```
>> B=[1 3 1 0 -1; 2 1 0 1 7; 0 0 1 3 4;0 0 2 1 9]
B =
     1     3     1     0    -1
     2     1     0     1     7
     0     0     1     3     5
     0     0     2     1     9
>> B(2:3,2:4)   % segunda a tercera filas y segunda a cuarta columnas
ans =
     1     0     1
     0     1     3
>> B(2,1:3)     % segunda fila y columnas de primera a tercera
ans =
     2     1     0
>> B(1,1:2:5)     % columnas impares de la primera fila
```

```
ans =
     2     0     7
>> B(2,:)       % segunda fila (todas las columnas)
ans =
      2     1     0     1     7
>> B(2,4)       % elemento que ocupa el lugar (2,4)
ans =
     1
>> B(:,3)       % tercera columna (todas las filas)
ans =
     1
     0
     1
     2
>> B([3 1 1],:) % filas 3, 1 y 1 (todas las columnas)
ans =
     0     0     1     3     5
     1     3     1     0    -1
     1     3     1     0    -1
>> C= [1 -2;7 -2];
>> C(:)           % lista con todos los elementos de la matriz
ans =
     1
     7
    -2
    -2
```

Este forma de manipular matrices permite un simple método de intercambio de filas o columnas de una matriz sin el tradicional empleo de variables intermedias.

```
>> A=[2 3;1 4;7 6]
A =
     2     3
     1     4
     7     6
>> A([1 3],:)=A([3 1],:)   % intercambio de las filas 1 y 3
A =
     7     6
     1     4
     2     3
```

1.3. Uso del operador \

El operador \ permite resolver, si es posible, un sistema $Ax = b$ mediante la orden $A\backslash b$.

```
>> A=[2 1;1 2];  % Primer ejemplo, solucion unica
>> b=[3 1]';
>> A\b
ans =
    1.6667
   -0.3333
>> A*ans
ans =
    3.0000
    1.0000
```

Si el sistema es compatible determinado, la orden anterior dará la solución. No obstante, hay que tener cuidado con este operador, porque no avisa cuando el sistema no es compatible: se limita a devolver una solución por mínimos cuadrados. Cuando el sistema es indeterminado, compatible o no (esto último quiere decir que hay más de una solución por mínimos cuadrados), MATLAB devuelve una única solución, elegida mediante un criterio que no viene al caso. En esta situación, avisa de la falta de unicidad.
Explicar cómo resuelve MATLAB el sistema es asunto mucho más complejo, ya que emplea una batería de métodos, eligiendo en función de la forma de la matriz.

1.4. Comparaciones, ordenaciones y búsquedas

De una forma muy simple se pueden localizar los valores máximo y mínimo en una matriz, así como su localización.

```
>> x=[1 2 3 5 3 1 -7];
>> max(x)
ans =
     5
>> min(x)
ans =
    -7
>> [cual,donde]=max(x)
cual =
     5
donde =
     4
>> A=[1 2 3;5 7 -1;2 3 4;1 1 1];
```

```
>> max(A)       % da un vector fila con los maximos de cada columna
ans =
     5     7     4
>> [cual,donde]=max(A)  % para cada columna, 'donde' indica la fila
cual =
     5     7     4
donde =
     2     2     3
>> max(max(A))   % o tambien max(A(:))
ans =
     7
```

Veamos algo de ordenación de elementos en una matriz

```
>> x=[1 2 3 5 3 1 -7];
>> sort(x)    % ordena los elementos en orden ascendente
ans =
    -7     1     1     2     3     3     5
>> A=[-2 4 7; 5 -6 -4;-2 -7 -9]
A =
    -2     4     7
     5    -6    -4
    -2    -7    -9
>> sort(A)    % ordena los elementos dentro de cada columna
ans =
    -2    -7    -9
    -2    -6    -4
     5     4     7
```

El orden descendente se puede obtener a partir del ascendente con cambios de posiciones de los elementos, pero también con un doble cambio de signo:

```
-sort(-A)          %   orden descendente
```

La orden `find` sirve para encontrar las posiciones de una matriz que cumplen alguna condición. Al igual que muchas otras funciones de MATLAB, la orden devuelve resultados distintos según el número de argumentos de salida que se soliciten.

```
>> A=[-1 2 5;3 0 -1]
A =
    -1     2     5
     3     0    -1
>> find(A<0)       % posiciones de los elementos negativos de A
```

```
ans =
     1
     6
>> [i,j]=find(A<0)     % filas y columnas de los elementos negativos de A
i =
     1
     2
j =
     1
     3
```

Como se ve en el ejemplo, si se pide sólo un argumento, `find` devuelve las posiciones de los elementos que cumplen la condición, tal y como se guarda la matriz, por columnas. Si se piden dos argumentos, `find` devuelve las posiciones de filas y columnas.

Ejercicios propuestos

1. Sea la matriz cuadrada
$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 3 \\ 3 & 3 & 1 \end{pmatrix}$$
   a) Construye una matriz añadiendo la matriz identidad de rango 3 a la derecha de la matriz A.
   b) Suma a la tercera fila, la primera fila multiplicada por -3.
   c) Cambia la primera columna de A por la tercera.
   d) Construye una nueva matriz cuyas columnas sean las columnas primera y tercera de A.
   e) Construye una nueva matriz cuyas filas sean las columnas primera y tercera de A.

2. Sea la matriz cuadrada
$$A = \begin{pmatrix} 1 & 4 & 0 \\ 0 & 2 & 3 \\ 3 & 3 & -7 \end{pmatrix}.$$
   a) Halla el valor mínimo dentro de cada fila de A.
   b) Ordena los elementos de A en orden descendente dentro de cada columna.
   c) Ordena los elementos de A en orden ascendente dentro de cada fila.
   d) Forma una lista con los elementos de A ordenada de forma ascendente.
   e) Halla el máximo en valor absoluto de los elementos de la matriz A.

3. Suma un mismo escalar a todos los elementos de una matriz.

4. En una sola orden de MATLAB crea una matriz 3×5 cuyo único elemento sea el 7.

5. Con una sola orden de MATLAB crea una matriz aleatoria 4×4 de números reales entre -5 y 5.

   INDICACIÓN: Ejecuta `help rand` para saber cómo generar números aleatorios en distribuciones uniformes (`randn` se emplea para distribuciones normales).

6. Con una sola orden de MATLAB crea una matriz aleatoria 4×4 de números enteros entre -5 y 5.

7. Considera la siguiente orden de MATLAB: `A=magic(5)`. En una sola orden:

   a) Define una matriz `B` formada por las filas pares de la matriz `A`.

   b) Define una matriz `C` formada por las columnas impares de la matriz `A`.

   c) Define una vector `d` formada por la tercera columna de la matriz `A`.

   d) Elimina la tercera fila de la matriz `A`.

8. Sea `x=(0:pi/2:2*pi)`. Con una sola orden de MATLAB crea una matriz cuya primera fila es `x`, su segunda fila es el seno de cada elemento de `x` y cuya tercera fila el coseno de cada elemento de `x`.

9. Define un vector `a` formado por los cuatro primeros números impares y otro `b` formado por los cuatro primeros números pares de varias formas distintas. Empléalos para construir la matriz

$$\begin{pmatrix} 2 & 4 & 6 & 8 \\ 6 & 12 & 18 & 24 \\ 10 & 20 & 30 & 40 \\ 14 & 28 & 42 & 56 \end{pmatrix}.$$

10. Construye una matriz $n \times n$, $C = (c_{ij})$

    a) con $c_{ij} = i\,j$,

    b) con $c_{ij} = \cos(i\,j)$.

11. Construye de distintas formas la matriz

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix}$$

Ejercicios más avanzados

1. En una sola instrucción, cambiar todos los valores de la diagonal de una matriz cuadrada a cero.

2. En una sola instrucción, sustituir todos los valores de la diagonal de una matriz cuadrada por los elementos de un vector dado.

3. Ordenar los elementos de una matriz del menor al mayor manteniendo su forma (indicación: emplear la orden `reshape`.

4. En una sola instrucción, poner a cero todos los elementos negativos de una matriz.

5. En una sola instrucción, poner a cero todos los elementos de una matriz que estén entre -1 y 1. (La conjunción lógica es &).

6. De tres formas distintas (cada una en una sola instrucción), averiguar el número de elementos de una matriz, de forma que al final tengamos un número.

Capítulo 2

Funciones y ficheros

Hasta el momento nos hemos ocupado del trabajo con MATLAB en modo interactivo, sin embargo como ya se ha comentado también es posible trabajar en modo programado utilizando para ello los llamados *m–ficheros*. Éstos son ficheros de texto con extensión `.m` que contienen instrucciones en MATLAB los cuales se ejecutan desde la pantalla de comandos (*workspace*). Hay dos tipos elementales de m–ficheros:

- los *scripts* están formados simplemente por una serie de instrucciones en MATLAB que se ejecutan como si estuviéramos en modo interactivo;
- las *m–funciones* son el equivalente en MATLAB a las subrutinas (procedimientos) de los lenguajes de programación tradicionales.

Hay distintas formas crear y/o editar un nuevo m-fichero, a continuación explicamos cómo se haría utilizando el editor de MATLAB (en un entorno WINDOWS), aunque podría emplearse cualquier editor de ficheros ASCII. Un primera forma de hacerlo sería ejecutando la orden

---

```
>> edit
```

De esta forma se abre una nueva ventana en la que podemos teclear el conjunto de órdenes en MATLAB por ejecutar (véase la Figura 2.1).
Otra forma de generar un nuevo m–fichero sería siguiendo los siguientes pasos:

1. seleccionar del menú principal la opción: *File* → *New* → *M–file* (véase la figura 2.2)
2. escribir el conjunto de instrucciones en MATLAB (véase de nuevo la figura 2.1)
3. guardar el fichero (*File* → *Save as...*) con extensión `.m`.

Podemos elegir la carpeta dónde guardarlo, por defecto se guarda en la carpeta *work* dentro de la carpeta de MATLAB. En general es conveniente guardar los ficheros en la carpeta en la que se está trabajando. En caso contrario, para algunas situaciones que emplearemos posteriormente, hace falta modificar el llamado *path*, que indica al sistema operativo dónde buscar nuevos documentos.

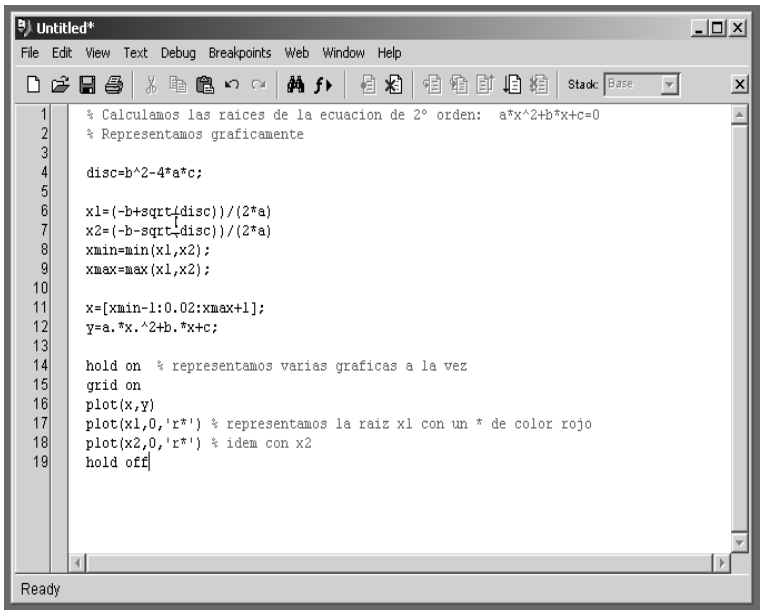

Figura 2.1: Edición un nuevo m–fichero

2.1. Scripts

Un *script* es un m–fichero que agrupa una serie de instrucciones de MATLAB en el que no se requieren ni argumentos de entrada ni de salida y que permite la ejecución repetidas veces de esas órdenes de una forma sencilla y sin ser necesario teclearlas en cada ocasión. En este tipo de m-ficheros se operan con variables declaradas en la pantalla de comandos.
Para calcular las raíces de la ecuación de segundo orden $ax^2 + bx + c = 0$ editamos un script (siguiendo los pasos anteriores) llamado `raices.m` con las siguientes instrucciones de MATLAB:

`raices.m`

```
% Calculamos las raices de la ecuacion de 2 orden:  a*x^2+b*x+c=0
% Representamos graficamente

x1=(-b+sqrt(b^2-4*a*c))/(2*a);
x2=(-b-sqrt(b^2-4*a*c))/(2*a);
xmin=min(x1,x2);
xmax=max(x1,x2);
x=xmin-1:0.02:xmax+1;     % lista de puntos en el eje X
y=a.*x.^2+b.*x+c;         % valores del polinomio en esos puntos

hold on  % representamos varias graficas a la vez
grid on
plot(x,y)
```

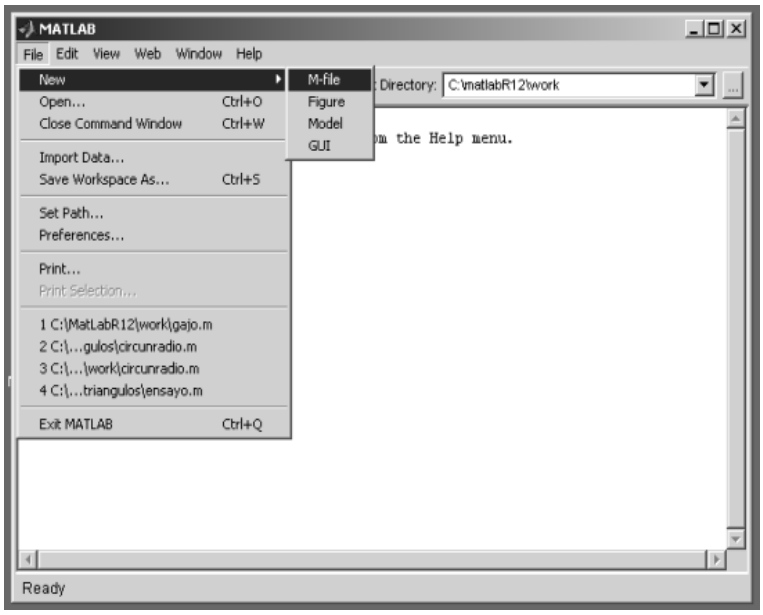

Figura 2.2: Cómo editar un nuevo m–fichero a través del menú principal del editor de MATLAB

```
plot(x1,0,'r*') % representamos la raiz x1 con un * de color rojo
plot(x2,0,'r*') % idem con x2
hold off
```

A la hora de programar es recomendable documentar los programas de forma que se facilite su lectura. Ese es el objeto de las dos primeras líneas de comentarios del script `raices`. En ellas se aclara qué es lo que se hace concretamente, además estas líneas se incorporan al sistema de ayuda de MATLAB, de modo que

---

```
>> help raices
  Calculamos las raices de la ecuacion de 2 orden:  a*x^2+b*x+c=0
  Representamos graficamente
```

Para ejecutar el script de forma que tengamos la representación gráfica y las raíces de $y = 2x^2 + 3x - 5$, se hace simplemente lo siguiente:

```
>> a=2;b=3;c=-5;
>> raices
```

Con esta ejecución se han generado dos nuevas variables, las raíces, `x1` y `x2`.

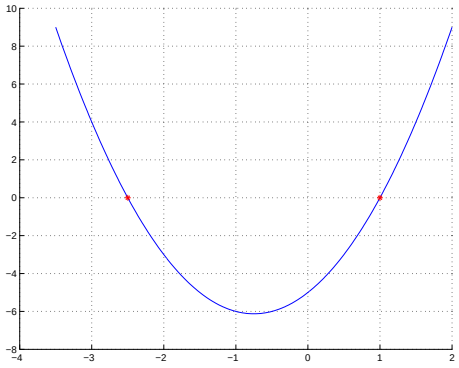

Figura 2.3: Representación gráfica obtenida al ejecutar el script `raices` con `a=2; b=3; c=-5`

```
>> whos x1 x2
  Name      Size          Bytes  Class
  x1        1x1               8  double array
  x2        1x1               8  double array
Grand total is 2 elements using 16 bytes
>> x1
x1 =
    1
>> x2
x2 =
   -2.5000
```

Otra utilidad de los scripts es como fichero de datos o parámetros. Por ejemplo, creamos un m–fichero llamado `datos.m` que contenga las siguientes órdenes de asignación:

`datos.m`

```
% Fichero de datos
tiempo=[0:0.2:1];
posicion=[1.3, 1.4, 1.2, 1.1, 1.0, 1.6];
media=sum(posicion)/length(tiempo);
```

Al ejecutar en MATLAB

```
datos
```

aunque no se muestra en pantalla ningún resultado aparecen tres variables: `tiempo` (vector), `posicion` (vector) y `media` (escalar). Si ya existían, se les asigna un nuevo valor. Si no, se crean nuevas.

2.2. Funciones

Una cualidad de MATLAB es la de permitir generar nuestras propias funciones para un problema específico que queramos resolver. De esta forma ampliamos la potencia de MATLAB ya que estas nuevas funciones adaptadas a nuestras necesidades se pueden utilizar del mismo modo que las que ya tiene MATLAB predefinidas, como son por ejemplo, `det, rank, sum, ...`
Supongamos que para una matriz dada necesitamos saber cuál es su diagonal recorrida de abajo hacia arriba. Vamos a crear una función, llamada `diagonal`, que admita como argumento de entrada una matriz y devuelva un vector que contenga la información buscada. Para ello utilizamos el editor de MATLAB para crear un m–fichero con las siguientes órdenes

`diagonal.m`

```
function x=diagonal(A)

% x=diagonal(A)
% Devuelve un vector con la diagonal de A en orden inverso
%           A : matriz
%           x : vector de la diagonal de A reordenada

n=size(A,1);  % # filas  de A
x=diag(A);
x=x(n:-1:1);  % reordenamos
```

En la primera línea se especifica que el m–fichero es de tipo función, se dice cuál es el nombre y los argumentos tanto de entrada como de salida.

Advertencia. Aunque no es necesario, sí es recomendable asignar al m–fichero el mismo nombre que a la función para evitar confusiones. El nombre de verdad de una función no es la palabra que se coloca en la definición

```
function argSalida=nombre(argEntrada)
```

sino el nombre del fichero con extensión `m` donde se ha guardado. En la versión de MATLAB para Windows, hay que tener en cuenta que el sistema operativo no distingue en estas situaciones entre mayúsculas y minúsculas. Por esta razón, hay que tener en cuenta que variantes de un mismo nombre con mayúsculas y minúsculas no son distinguidas por MATLAB si se emplean para una función guardada en un fichero.

Al igual que antes en las primeras líneas comentadas se explica brevemente el programa y éstas son las que pasan a formar parte del sistema de ayuda de MATLAB. Estas líneas pueden colocarse también antes de la instrucción de declaración de la función, aunque el formato mostrado en el ejemplo es preferible, pues deja la primera línea, más visible, para la declaración.

```
>> help diagonal
  x=diagonal(A)
  Devuelve un vector con la diagonal de A en orden inverso
```

```
            A : matriz
            x : vector de la diagonal de A reordenada
>> B=[1 2 3;4 5 6;7 8 9];
>> v=diagonal(B)
v =
     9
     5
     1
```

Aunque en este sencillo ejemplo se tienen un solo argumento de entrada y un solo argumento de salida, en general, se pueden tener o no varios argumentos de entrada como de salida. Veamos algunos ejemplos.

`mifuncion.m`

```
function f=mifuncion(x,y)
f=y*(1-y)+x;
```

Ahora vamos a emplearla:

```
>> mifuncion(0,1)
ans =
     0
>> mifuncion(3,2)
ans =
     1
>> mifuncion(4,1)
ans =
     4
```

Observamos que hay un cierto inconveniente con que la función no esté vectorizada. Vamos a hacer un pequeño arreglo, vectorizamos para poder evaluar la función para vectores.

`mifuncion.m`

```
function f=mifuncion(x,y)
f=y.*(1-y)+x;
```

En este caso sólo debemos vectorizar el producto `(.*)` pues la suma ya es una operación que se realiza de forma vectorizada. Ahora si hacemos lo siguiente

```
>> mifuncion([0 3 4],[1 2 1])
ans =
     0     1     4
```

obtenemos un vector con el valor de la función en $(0,1)$, $(3,2)$ y $(4,1)$, respectivamente.

Advertencia. Cuando una función de dos variables se vectoriza, hay que enviar como argumentos de entrada matrices de la misma dimensión (o de dimensiones compatibles si la definición es más compleja). La vectorización supone que se evalúa en pares de valores que ocupan la misma posición. Es decir, si enviamos dos vectores

$$(x_1,\ldots,x_n), \qquad (y_1,\ldots,y_n)$$

estaremos evaluando en los pares (x_i, y_i). Si queremos cruzar todos los x_i con todos los y_j, hay que emplear la instrucción `meshgrid`. Veremos este tipo de evaluaciones frecuentemente en los gráficos de funciones bidimensionales.

Modificamos ahora la función de la siguiente forma

`mifuncion.m`

```
function [f,promedio]=mifuncion(x,y)

f=y.*(1-y)+x;
promedio=sum(f)/length(x);
return              %  opcional
```

Ahora tenemos dos argumentos de salida, `f` y `promedio`. El primero de ellos siempre es un dato que se devuelve. Para el segundo (y en general para los siguientes), podemos elegir si devolverlos o no.

```
>> valor=mifuncion([0 3 4],[1 2 1])
valor =
     0     1     4
>> [valor,media]=mifuncion([0 3 4],[1 2 1])
valor =
     0     1     4
media =
    1.6667
```

Observa también que en esta última modificación de `mifuncion.m` hemos añadido la orden `return` con lo que se vuelve al modo interactivo. A diferencia de lo que ocurre con otros lenguajes de programación esta instrucción no es necesaria al final de la definición de la función. Su empleo

es común cuando se quiere salir de la función si se cumple algún tipo de condición (lo veremos en la sección sobre estructuras de programación).

Advertencia. Los argumentos de salida que se exportan (asignan a variables en consola o en otras funciones) van siempre por orden, de izquierda a derecha. No se pueden exportar el primer y el tercer argumento sin exportar el segundo.

No hay ningún problema en que una m–función llame a otra m–función, siempre que no se entre en un ciclo recursivo, es decir, que siguiendo el rastro de llamadas de una función a otra función regresemos a la original. La recursividad existe en MATLAB pero hay que tratarla de una manera especial.

2.3. Vectorización de funciones con condicionales

Cuando una función está definida a trozos, no es obvio cómo vectorizarla. Mira el siguiente ejemplo. Vamos a definir la función característica de la bola unidad cerrada en dimensión dos

$$f(x,y) = \begin{cases} 1, & x^2+y^2 \leq 1 \\ 0, & \text{en otro caso} \end{cases}$$

`f.m`

```
function z=f(x,y)

z=zeros(size(x));
cond=find(x.^2+y.^2-1<=0);
z(cond)=1;
```

El vector `cond` contiene las posiciones de `x` e `y` (matrices del mismo tamaño) donde se cumple la condición $x^2 + y^2 \leq 1$. Las posiciones se entienden viendo todas las matrices como vectores, leídas por columnas como de costumbre.

2.4. Variables globales

En la definición de una función (al igual que en cualquier lenguaje de programación) los nombres de las variables son mudos, es decir, se entienden y definen únicamente dentro del fichero, sin relación con las variables exteriores. La única excepción la forman las variables declaradas como globales en ambos contextos.
Veamos esto con un ejemplo. Construimos y guardamos la siguiente función, completamente trivial. La instrucción `a=4` no hace nada relevante, ya que realiza una asignación interna que no se exporta.

f.m

```
function y=f(x)
a=4;
y=2*x;
```

Ahora hacemos una prueba de ejecución. En el contexto de la consola, `'a'` es una variable global. No obstante, como ésta no ha sido declarada así en la función `f`, ni el valor en consola se importa a la función, ni el valor dentro de la función se exporta.

```
>> global a
>> a=[2 4];
>> y=3;
>> x=f(y)
x =
     6
>> a
a =
     2  4
```

Ahora cambiamos la función para que `a` sea variable global dentro de la función también.

f.m

```
function y=f(x)
global a
a=4;
y=2*x;
```

En este caso, tras ejecutar la función, el valor de la variable `'a'` pasa a ser `a=4`.

```
>> global a
>> a=3;
>> f(3.5)
ans =
      7.0
>> a           % la ejecucion de f ha cambiado el valor de a
a =
     4
```

Las variables globales son variables de entrada y salida simultáneamente sin que aparezcan en ninguna de las dos listas (listas de argumentos de entrada ni de salida).
Una utilidad típica de las variables globales es poder transmitir listas largas de parámetros de la consola a una subrutina o al revés, sin incluirlas en las definiciones de las funciones.
Las variables globales se definen y emplean únicamente en el contexto donde estén declaradas. Pueden estar perfectamente definidas en un grupo de m–funciones y no en la consola. En tal caso, no podemos acceder a su valor en consola, pero sí en cualquier función que las declare.
Por ejemplo, además de la función `f` anterior, construimos otra función, llamada `g`.

`g.m`

```
function y=g(x)
global a
y=a*x;
```

En la siguiente cadena de ejecuciones (donde nos preocupamos de que `a` quede borrada en la consola), se ve claramente cómo se transmite el valor de `a` de una función a otra sin que pase por consola.

```
>> clear a
>> f(2);      % a ha quedado declarado globalmente como a=4;
>> a          % ... pero no en consola
??? Undefined function or variable 'a'.
>> g(3)       % matem'aticamente g(x)=a*x
ans =
    12
```

Al igual que todas las variables empleadas en una sesión están almacenadas en la memoria a la que se accede desde la consola (en el *workspace*), todas las variables globales están almacenadas en una memoria global a la que se puede acceder parcialmente a través del comando `global`

2.5. Funciones como argumento de otras funciones

El nombre de una función `f.m` es la cadena de caracteres anterior a la extensión y con comillas:

```
'f'
```

El *handle* es la siguiente expresión

```
@f
```

Es un tipo específico de datos de MATLAB. Para el siguiente ejemplo suponemos que tenemos definida una función `f.m`

```
>> u=@f;
>> whos u
  Name      Size                    Bytes  Class

  u         1x1                        16  function_handle array

Grand total is 1 element using 16 bytes
```

Para evaluar una función se puede emplear `feval` (*function evaluation*) que permite evaluar una función en uno o varios puntos: el primer argumento de `feval` es el nombre de la función o su *handle*. Después se dan los argumentos donde se quiere evaluar esta función. La siguiente función, con dos argumentos, está vectorizada.

`f.m`

```
function z=f(x,y)
z=x.*y;
```

Las instrucciones siguientes dan evaluaciones de esta función en los puntos $(2, 3)$ y $(1, 4)$. Nótese que para que MATLAB encuentre la función a la hora de evaluarla, el fichero correspondiente debe estar en la carpeta donde estemos trabajando o bien la carpeta donde se guarda la función debe ser añadida al *path*. Para modificar el *path*, se hace dentro del menú *File*.

```
>> f([2 1],[3 4])
ans =
     6     4
>> feval(@f,[2 1],[3 4])
ans =
     6     4
>> feval('f',[2 1],[3 4])
ans =
     6     4
>> feval(@sin,pi/4)      % tambien feval('sin',pi/4)
ans =
    0.7071
```

En muchas circunstancias puede resultar conveniente emplear una función como argumento de otra. La siguiente función realiza una fórmula del punto medio compuesta para una integral

$$\int_a^b f(x)dx \approx h\sum_{j=1}^{n} f(h\,(j-1/2)), \qquad h=\frac{b-a}{n}.$$

Dentro de la función `puntomedio`, la variable `f` es muda y representa a cualquier función que nos pasen como argumento, sea dándonos su nombre o su *handle*.

`puntomedio.m`

```
function y=puntomedio(f,a,b,n)

% function y=puntomedio(f,a,b,n)
%         formula del punto medio compuesta para \int_a^b f(x)dx
%         con n subintervalos de igual longitud
%         f = nombre de la funcion o handle asociado

h=(b-a)/n;
puntos=a+h*((1:n)-1/2);
y=h*sum(feval(f,puntos));
```

Ahora podemos ejecutar esta función en cualquiera de las siguientes formas.

```
>> puntomedio(@sin,0,pi/2,4);
>> puntomedio('sin',0,pi/2,4);
```

2.6. Comunicación entre una función y el usuario

Si la ejecución de una función que está en un m–fichero necesita de la introducción de algún dato, se puede realizar mediante la orden `input`. Por ejemplo

```
a=input('Introduce el valor de a: ')
```

solicita un dato que es introducido en la variable `a`. El dato puede ser de cualquier tipo, sea un escalar, una matriz, etc. La introducción del dato se interrumpe una vez se pulse la tecla *Enter*, salvo que se hayan introducido puntos suspensivos antes.

```
>> b=input('Matriz: ')
Matriz: [3 2 1;...           %  no hemos terminado (puntos suspensivos)
        1 -1 2]
b =
     3     2     1
     1    -1     2
```

Con la siguiente variante

```
input('Nombre:    ','s')
```

nos podemos ahorrar teclear las comillas antes de introducir una variable carácter, ya que fijamos el tipo de la variable que pedimos a carácter (`'s'` hace referencia al tipo *string*).
Cuando se quiere escribir un mensaje en pantalla durante la ejecución de una función, se puede emplear

```
disp('Mensaje')
```

(`disp` es abreviatura de *display*). La función de `disp` admite como argumento una variable de cualquier tipo. Nota la diferencia

---

```
>> b=[2 3];
>> disp(b)
     2     3
>> b               % vuelve a escribir el nombre de la variable
b =
     2     3
```

La orden

```
pause
```

detiene la ejecución del programa, hasta que se pulsa la tecla *Enter*. Como ya hemos mencionado,

```
return
```

interrumpe en esa línea la ejecución de la función, regresando a donde se la hubiera llamado. Finalmente, la función

```
error('Mensaje')
```

escribe el mensaje en pantalla y finaliza la ejecución del m–fichero, regresando a donde se la hubiera llamado (la consola u otra función). Es, por tanto, equivalente a un `disp` del mensaje y un `return` después.

2.7. Gestión de ficheros

La gestión de los m–ficheros se hace con los siguientes comandos

Comando	Descripción
`edit nombrearchivo.m`	abre la ventana de edición y el fichero indicado
	si el fichero no existe, se crea en blanco
`edit nombrearchivo`	(igual)
`type nombrearchivo.m`	visualiza en pantalla el contenido del fichero
	sin editarlo (así no se puede modificar)
`type nombrearchivo`	(igual)
`del nombrearchivo.m`	borra el m–fichero indicado;
	requiere que no esté abierto en la ventana de edición
`what`	lista todos los m–ficheros en el directorio actual
`dir`	lista todos los archivos en el directorio actual
`ls`	(igual)
`cd`	muestra el directorio actual
`cd path`	cambia al directorio dado por path
`which ejem`	muestra el directorio de ejem.m

2.8. Ficheros de datos y resultados

La gestión de ficheros de datos y resultados se realiza en MATLAB con ficheros de formato binario, a los que sólo se accede abriendo el MATLAB. Los ficheros con extensión `mat` son ficheros con formato interno de MATLAB. Si se intentan editar no se puede ver qué hay guardado de una forma clara.
Las instrucciones más elementales son:

- Para guardar en el fichero `nombre.mat` todas las variables en uso, se escribe

  ```
  save nombre
  ```

- Para guardar las variables `x`, `y`, `z` en el fichero `nombre.mat`, se escribe

  ```
  save nombre x y z
  ```

  En ambos casos, si hay alguna versión previa del fichero, se borra.

- Para recuperar el contenido del fichero `nombre.mat`, se escribe

  ```
  load nombre
  ```

Los ficheros no sólo guardan los datos, sino también el nombre y tipo de las variables. Cuando se carga un fichero `.mat` (con la orden `load`), MATLAB recupera todas las asignaciones que se habían realizado allí. Esto es, el fichero incluye las sentencias de asignación.
Las instrucciones `save` y `load` se emplean como comandos: sus argumentos no van entre paréntesis. Esto se debe a que están programados con lo que se llama **duplicidad comando–función**. En el fondo

```
save fichero x y
```

es lo mismo que

```
save('fichero','x','y')
```

En esta instrucción los argumentos de `save` son los nombres del fichero y de las variables, no el fichero y las variables en sí. Otro ejemplo ya empleado de comando que no es más que una función con una grafía simplificada es `whos`. Así

```
whos('x')
```

se escribe como comando

```
whos x
```

El siguiente ejemplo muestra cómo se pueden añadir variables al fichero una vez ya escrito. En caso de que volviéramos a guardar una variable que ya hemos guardado, se sobreescribiría sobre ella y sólo tendríamos la última versión.

```
>> a=[pi 0.4; 2 1]; b=[2 3 4];
>> save fichero a
>> save fichero b -append    % a\~{n}adir al final
>> clear all
>> load fichero
>> a        % estamos en format long para ver todas las cifras
a =
   3.14159265358979   0.40000000000000
   2.00000000000000   1.00000000000000
>> b
b =
     2     3     4
>> a=3*a;
>> save fichero a -append
>> clear a
>> load fichero
>> a
a =
   9.42477796076938   1.20000000000000
   6.00000000000000   3.00000000000000
```

En caso de que queramos guardar las variables en un fichero con formato texto, que podamos leer desde un editor cualquiera o importar desde un lenguaje de programación más tradicional, hay que emplear la opción `-ascii`. Al cargar el fichero en formato texto, MATLAB entiende que hay una única matriz en el fichero y la asigna a una matriz que se llamará como el fichero, ya que no se han guardado más que los números. Además, el defecto es que los números sólo se guardan en precisión simple. Si se quieren guardar todas las cifras hay que forzar a MATLAB a que lo haga, con la opción `-double`.

```
>> a=[pi 0.4; 2 1];
>> save fichero.txt a -ascii
>> type fichero.txt
  3.1415927e+000  4.0000000e-001
  2.0000000e+000  1.0000000e+000
>> load fichero.txt -ascii
>> fichero       % el contenido se asigna a la variable fichero
fichero =
   3.14159270000000   0.40000000000000
   2.00000000000000   1.00000000000000
>> save fichero.txt a -ascii -double
>> type fichero.txt      % ahora si se han guardado todas las cifras
  3.1415926535897931e+000  4.0000000000000002e-001
  2.0000000000000000e+000  1.0000000000000000e+000
```

En principio, en formato texto se pueden guardar más de una matriz, pero luego no se podrán recuperar, ya que al cargar el fichero la asignación a una matriz no será válida.

```
>> a=[pi 0.4; 2 1]; b=[2 3 4];
>> save fichero.txt a b -ascii
>> type fichero.txt
  3.1415927e+000  4.0000000e-001
  2.0000000e+000  1.0000000e+000
  2.0000000e+000  3.0000000e+000  4.0000000e+000
>> load fichero.txt -ascii
??? Error using ==> load
Number of columns on line 3 of ASCII file
C:\Documents and Settings\invitado-1\Escritorio\fichero.txt
must be the same as previous lines.

See FILEFORMATS for a list of known file types
and the functions used to read them.
```

Para escribir en ficheros de texto expresiones más complicadas, con texto y variables mezclados, véase en las ayudas el manejo de la función

```
fprintf
```

Ejercicios propuestos

1. El siguiente ejemplo muestra el empleo de una subfunción:

```
function y=f(x)
y=2*ff(x);

function z=ff(u)
z=u*u;
```

Copia el ejemplo en un fichero `f.m` y pruébalo. ¿Qué hace exactamente? ¿Puedes llamar a la función `ff` desde fuera de la función `f` (por ejemplo desde la consola)?

2. Averigua qué hacen las siguientes instrucciones:

```
pause(3)
cputime
tic
toc
clock
```

Nota la diferencia entre el empleo de `cputime` (que mide tiempo de ejecución) y el par `tic-toc` (que mide tiempo transcurrido).

3. La matriz de Hilbert $n \times n$ viene dada por su elemento genérico

$$\frac{1}{i+j-1}$$

Empleando `cputime`, haz un script que mida el tiempo que necesita MATLAB para invertir la matriz de Hilbert $n \times n$. (MATLAB tiene implementada la matriz de Hilbert en la instrucción `hilb(n)` y la inversión de matrices se realiza con `inv`).

4. Construye una función vectorizada

$$f(x) = \begin{cases} x+1, & -1 \leq x \leq 0, \\ 1-x, & 0 \leq x \leq 1, \\ 0, & \text{en otro caso.} \end{cases}$$

5. Crea un script que haga lo siguiente: solicita del usuario un nombre de fichero y una extensión; con ellos más un número guardado en una variable `n` crea un fichero `nombreVersion.ext` donde guarda todas las variables. Por ejemplo, si `n=3` y el usuario teclea (así, sin comillas)

```
resultados
res
```

MATLAB debería guardar todas las variables en un fichero `resultados3.txt`

INDICACIONES: La concatenación de expresiones carácter se realiza como si fuera la concatenación de vectores

```
>> ['uno' ' y dos']
ans =
uno y dos
```

Para pasar un argumento numérico a carácter, se hace con `num2str` (abreviatura de *number to string*).

6. Crea una función que dados dos números enteros devuelva el número que se obtiene concatenándolos, de manera que si disponemos de los números `34` y `101`, devuelva `34101`.

Capítulo 3

Gráficos

MATLAB ofrece numerosas oportunidades para emplear rutinas gráficas en dos y tres dimensiones. En la ventana gráfica hay una paleta de comandos que permiten:

- añadir texto en posiciones deseadas,
- añadir flechas o líneas,
- seleccionar alguna de las componentes del gráfico y desplazarla en su caso,
- rotar el gráfico.

Las gráficas de MATLAB se pueden exportar a multitud de formatos gráficos puntuales y vectoriales (jpg, bmp, tiff, eps, png). Además está la posibilidad de guardarlos con la extensión `fig`. En ese caso, cuando se abre la figura se inicia la ejecución de MATLAB y se ofrece al usuario la figura tal y como estaba cuando la guardó, incluyendo modificaciones realizadas directamente sobre la ventana gráfica.
En lugar de dar instrucciones que se puedan ejecutar en la ventana de comandos, en este capítulo principalmente daremos scripts que generen uno o varios gráficos al ejecutarse.

3.1. Gráficos bidimensionales

La instrucción básica para dibujo de curvas planas es `plot`. La orden `plot` dibuja vector contra vector, siempre que tengan la misma longitud (da igual que uno sea fila y el otro columna). Es una instrucción a la que se le pueden añadir al final nuevos argumentos, como si estuviera comenzando.
También se puede emplear con un vector contra una matriz. En tal caso, se van dibujando, en distintas gráficas (con distintos colores), los pares que va formando el vector con las filas o columnas de la matriz. *En caso de duda, MATLAB siempre opta por leer las matrices por columnas.*
Se puede escoger el color para una gráfica lineal entre una lista habitual: `b` es azul (blue), `r` es rojo (red), `k` es negro (blac**k**), `g` es verde (green), etc. (Ver el final de la sección obtener la lista completa). Se pueden emplear líneas (`-`), líneas partidas (`--`), guiones y puntos (`.-`) y otro tipo de marcadores. (De nuevo, se puede encontrar una lista al final de esta sección).
Copia el siguiente script y ejecútalo. Al final, no cierres la figura.

script1.m

```
x=0:0.01:2*pi;
y=cos(x);
z=sin(2*x);
plot(x,y,'r-',x,z,'b--'); % dos dibujos
```

Si se quiere seguir dibujando, pero no queremos borrar la figura anterior, con `figure` se genera una figura nueva. El número de figura se establece correlativamente por MATLAB, sin *dejar huecos*. Aquí podemos además observar varias de las opciones de dibujo y etiquetado de MATLAB. Las opciones modifican el dibujo pero no suponen que se dibuja de nuevo.

script1b.m

```
figure         % en una nueva ventana
x=0:0.01:2*pi;
Y=[cos(x);sin(2*x)];
plot(x,Y)
axis([0 2*pi -1.5 1.5])     % ejes
xlabel('ejex')
ylabel('ejey')
title('titulo')
grid on
```

En la siguiente gráfica, vemos cómo emplear `ylim` para recortar los límites en el rango de la variable vertical. Esto es muy útil cuando se quieren dibujar gráficas de funciones que tienden a infinito en algún punto, ya que el escalado lo estropea todo.

script1c.m

```
close                       % cierra la ultima figura
figure(4)                   % abre la Figura 4
x=linspace(0,3,200);        % 200 puntos equiespaciados de 0 a 3
plot(x,1./(x-1).^2)
ylim([0 10]), box off       % quitamos la caja de alrededor
pause
close(1)                    % cerramos la primera figura
```

La orden

```
close
```

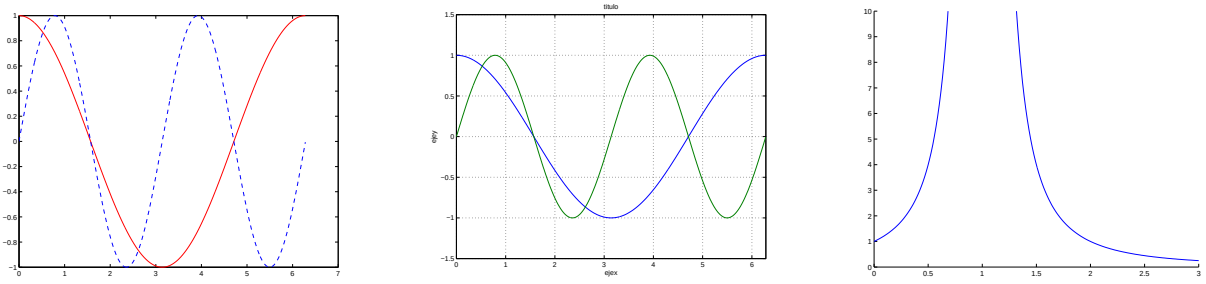

Figura 3.1: Gráficas respectivas de los tres primeros scripts.

cierra la última figura creada, modificada o seleccionada. Si queremos cerrar un figura concreta se le da como argumento el número de figura (escribiendo `close(3)` cerramos la figura 3). Todas las figuras se cierran con `close all`.

`script2.m`

```
x=linspace(0,pi,200);
y=1-x.^2/2+x.^4/16;
plot(x,cos(x),x,y)
legend('cos','\itTaylor')     % \it=cursiva
title('titulo con \alpha, \infty, \int_a^b')
pause
figure(2)                     % acceso a la segunda grafica
plot(x,cos(x),x,1-x.^2/2+x.^4/16)
legend('\bfcos','\itTaylor',2) % en otra esquina
text(0,0.5,'texto en grafica')  % (0,0.5) son las coordenadas
```

Por defecto, si no se escogen los marcadores, todas las gráficas son con línea continua. Los colores dentro de una misma gráfica `plot` se van rotando de una lista que comienza con azul, verde y rojo (en este orden).

Los macros de escritura (para escribir integrales, letras griegas, para cambiar tipos de letra) son de TEX. Así, se dispone de todas las letras griegas (poniendo su nombre en inglés), del símbolo para infinito (ver ejemplo) y de símbolos básicos como la integral, etc. Con `\it` se pasa el tipo a cursiva (*italic*) y con `\bf` a negrita (**boldface**). Notar la grafía `\itTaylor` o `\bfcos`, con todo seguido.

La instrucción `gtext` sirve para colocar algún tipo de comentario o texto en una gráfica viendo dónde queremos colocarlo. No obstante, eso se puede hacer manualmente, empleando la paleta de comandos de la ventana gráfica de MATLAB. Todos los cambios realizados manualmente sobre la gráfica se guardan cuando se exporta la gráfica a cualquier formato externo.

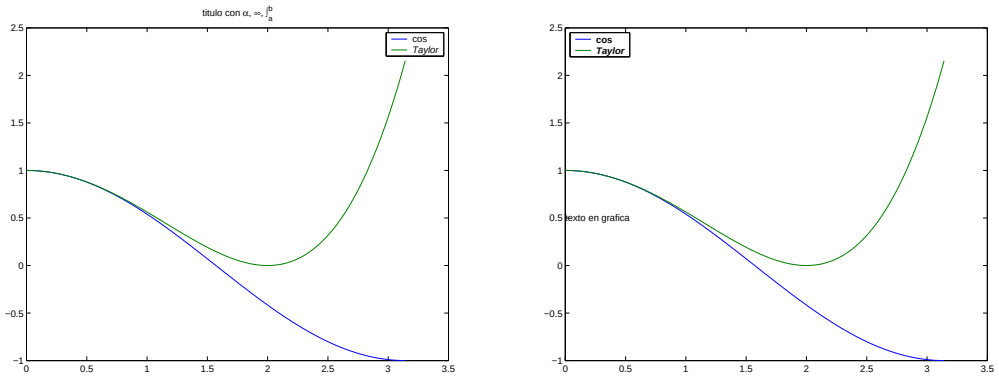

Figura 3.2: Las dos gráficas producidas por el `script2`

`script2b.m`

```
plot(x,cos(x)), axis equal   % misma escala en ambos ejes
gtext('texto para poner')    % hay que pinchar en el dibujo
pause
close all                    % cierra todos los dibujos existentes
```

Vamos seguidamente a ver cómo hacer una gráfica rellena de un color. La orden `fill` necesita tres argumentos: lista de coordenadas horizontales de los puntos, lista de coordenadas verticales, color de relleno. Si la lista de puntos no es cerrada (el último punto coincide con el primero), MATLAB los une automáticamente. En el siguiente ejemplo, vemos tres gráficas realizadas con `fill`.

El argumento que da el color se puede escoger con uno de los colores básicos o empleando el *estándar RGB*: $[\alpha\ \beta\ \gamma]$, con los tres parámetros en $[0, 1]$. Los colores [1 0 0], [0 1 0] y [0 0 1] corresponden exactamente a los que MATLAB denota como `'r'`, `'g'` y `'b'`: son las versiones más puras del rojo, el verde y el azul. $[\alpha\ \alpha\ \alpha]$ con $\alpha \in [0, 1]$ es un tono de gris, siendo [0 0 0] el negro y [1 1 1] el blanco.

`script3.m`

```
t=linspace(0,2*pi,20);
fill(cos(t),sin(t),'r',1+0.5*cos(t),0.5*sin(t),[0 0.5 0]);
hold on
pause
fill(0.3*cos(t),1.5+0.3*sin(t),[0.8 0.8 0.8])
```

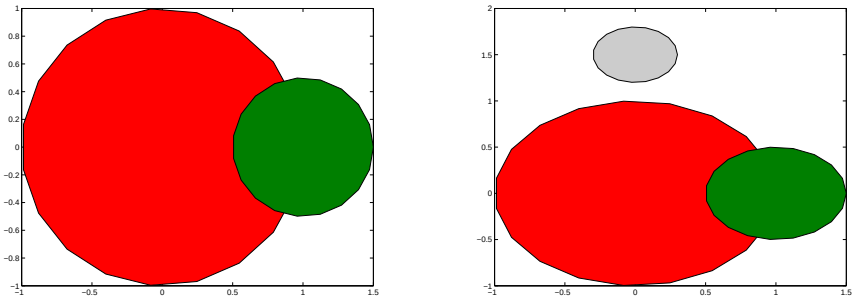

Figura 3.3: Gráficas producida por el `script3`. En la segunda se deforman los círculos para que quepan más cosas.

3.2. Disposiciones de múltiples gráficas

Exploramos ahora la orden `subplot`, que permite colocar varias gráficas dentro de la misma figura. Aprovechamos para emplear la orden `fplot` en su versión más simple: dando una cadena de caracteres (que debe estar vectorizada), que MATLAB convertirá en función para dibujar. Si la función tiene varias componentes, MATLAB las dibuja por separado, con distintos colores.

`script4.m`

```
subplot(221),fplot('cos(x)',[0 4*pi]);     % fplot=dibuja funciones
subplot(222),fplot('abs(cos(x))',[0 4*pi]),grid on;
subplot(223),fplot('[sin(x),sin(2*x),sin(3*x)]',[0 2*pi]);
subplot(224),fplot('[sin(x),cos(x)]',[0 2*pi])
legend('sen','cos');
pause
subplot(212),fplot('(2+cos(x))*sin(10*x)',[0 8*pi])
```

Por ejemplo, `subplot(324)` es una disposición de 3×2 dibujos. El siguiente número escoge cúal de los seis. El orden es por filas, luego el cuarto está en la segunda fila, primera columna. Salvo que se dé una nueva orden `subplot` se seguirá dibujando en el escogido anteriormente.
La última parte del script muestra que `subplot(212)` es compatible con `subplot(22x)`. Junta las dos posiciones inferiores en un único dibujo. Si el dibujo no se puede hacer uniendo un número exacto de posiciones del `subplot` precedente, se borra lo que haga falta para que quepa el nuevo dibujo, pero nada más.

3.3. Superficies

MATLAB dispone de una gran variedad de formatos para dibujar gráficas de funciones de dos variables y una componente. En general se emplean colores para resaltar las alturas, en una gradación típica de cálculo científico que escala las alturas del azul al rojo (de menor a mayor).

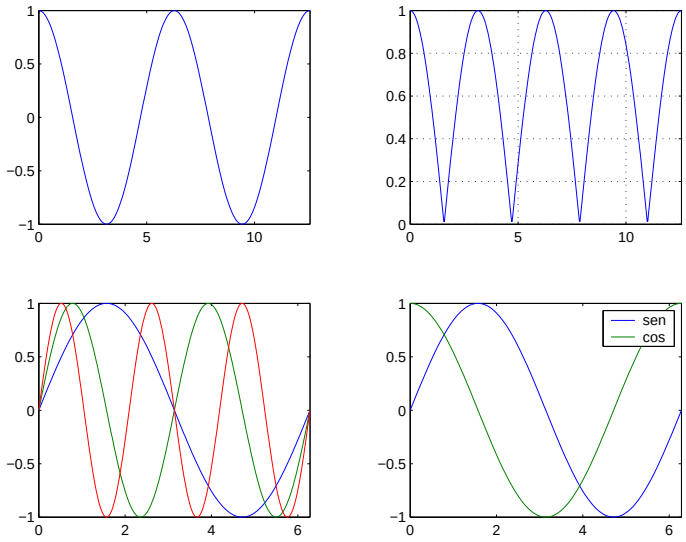

Figura 3.4: Primera gráfica producida por el `script4`

Cuando se van a emplear funciones de dos variables, necesitaremos cruzar una lista de valores $(x_1, \ldots, x_n)$ con otra $(y_1, \ldots, y_m)$. Esto lo hace la orden `meshgrid`. Si

$$\mathtt{x} = (x_1, \ldots, x_n), \qquad \mathtt{y} = (y_1, \ldots, y_m)$$

la instrucción

```
[u v]=meshgrid(x,y)
```

devuelve dos matrices con m filas y n columnas

$$\mathtt{u} = \begin{bmatrix} x_1 & x_2 & \ldots & x_n \\ x_1 & x_2 & \ldots & x_n \\ \vdots & \vdots & & \vdots \\ x_1 & x_2 & \ldots & x_n \end{bmatrix} \qquad \mathtt{v} = \begin{bmatrix} y_1 & y_1 & \ldots & y_1 \\ y_2 & y_2 & \ldots & y_2 \\ \vdots & \vdots & & \vdots \\ y_m & y_m & \ldots & y_m \end{bmatrix}$$

Así `u(i,j)`=x_j y `v(i,j)`=y_i. Recorriendo en paralelo las matrices `u` y `v` se obtienen, por tanto, todos los pares (x_i, y_j).

`script5.m`

```
t=linspace(0,2*pi,200);
subplot(221),plot3(sin(t),cos(t),sin(10*t)); % curvas en espacio
[X,Y]=meshgrid(-1:0.1:1,-1:0.1:1);          % =meshgrid(-1:0.1:1);
subplot(222),surf(X,Y,X.^2+Y.^2);
subplot(223),surfc(X,Y,X.^2+Y.^2);
subplot(224),contour(X,Y,X.^2-Y.^2);
```

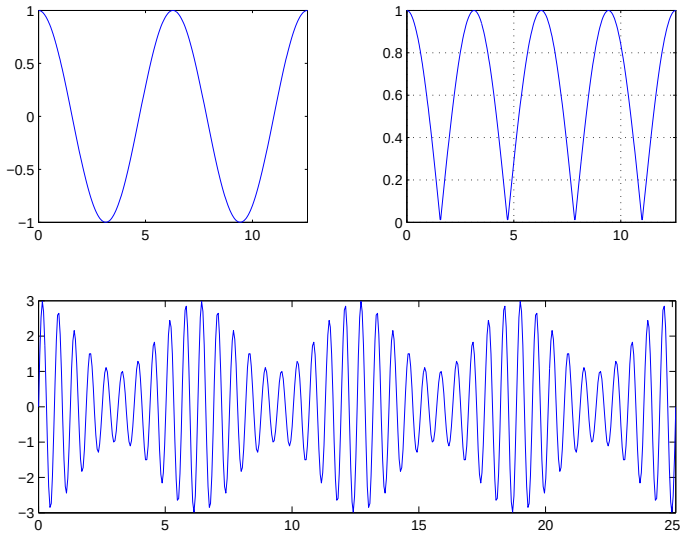

Figura 3.5: Segunda gráfica producida por el `script4`

Se puede hacer `[X,Y]=meshgrid(lista1)` con resultados evidentes. También existe

```
pcolor
```

para dar una vista zenital coloreada, equivalente a `surf`. El punto de vista de las gráficas tridimensionales siempre es con el observador en una esquina con (x, y) negativos y z positivo. El punto de vista se puede rotar directamente sobre la figura con las utilidades gráficas de los menús.
En todas las gráficas tridimensionales, por defecto el color es proporcional a la altura. La gama de colores es la llamada `jet` que organiza los colores del azul al rojo pasando por el verde. En el fondo es una gama de grises pero coloreada, donde el color se controla por un único parámetro.

`script5b.m`

```
subplot(221),mesh(X,Y,X.^2-Y.^2);
subplot(222),surf(X,Y,X.^2-Y.^2),shading flat;
subplot(223),surf(X,Y,X.^2-Y.^2),shading interp,colorbar;
subplot(224),contourf(X,Y,X.^2-Y.^2);
```

Este último script muestra algunas utilidades más. El comando `shading` permite controlar el tipo de rellenado en una superficie dibujada con `surf`. El rellenado plano (`flat`) quita las curvas paramétricas, que pueden llegar a ennegrecer mucho una figura, pero asigna un color plano a cada recuadro contenido entre las líneas paramétricas. El rellenado interpolado (`interp`), suaviza este resultado. Aún así, la calidad visual de la gráfica dependerá del número de puntos escogido para realizar la gráfica.

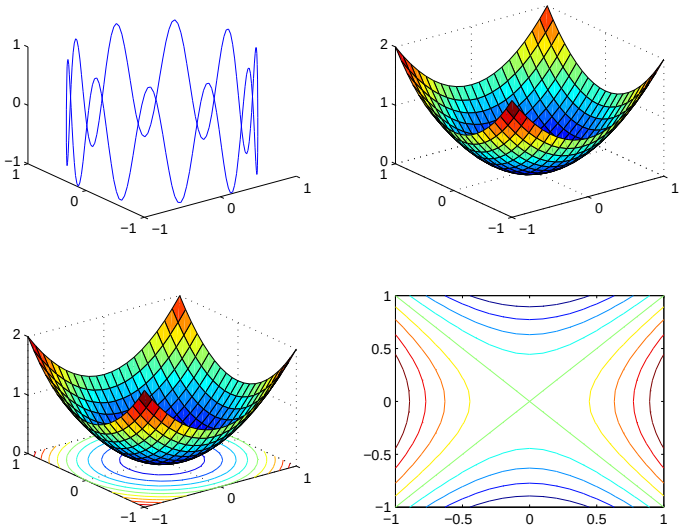

Figura 3.6: Gráfica producida por el `script5`

3.4. Cuestiones avanzadas sobre gráficos

3.4.1. Triangulaciones

La orden `trisurf` sirve para representar funciones definidas sobre una triangulación. Una triangulación consiste de:

- Dos vectores de igual longitud, correspondientes a las coordenadas horizontal y vertical de un conjunto de puntos del plano.
- Una matriz con tres columnas. En cada fila se escriben los índices de puntos que constituyen cada triángulo.

La triangulación no tiene que estar construida de forma que los triángulos no se crucen o corten. Es simplemente una lista de recorridos de tres en tres de puntos de una lista más grande. El siguiente ejemplo muestra cómo se emplea la orden `trisurf` para dibujar funciones sobre triangulaciones.

`script6.m`

```
[x,y]=meshgrid(0:2,0:2)
trii=[1 2 4;2 4 5;2 3 5;3 5 6;4 5 7;7 5 8;5 6 8;6 8 9];
                                % triangulaci'on
z=x.^2+y.^2;
x=x(:);y=y(:);z=z(:);           % lista de nodos
trisurf(trii,x,y,z),shading interp
```

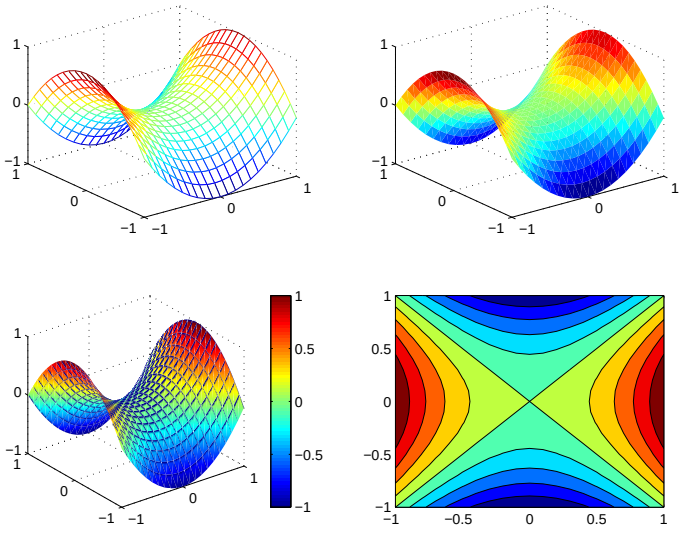

Figura 3.7: Gráfica producida por el `script5b`

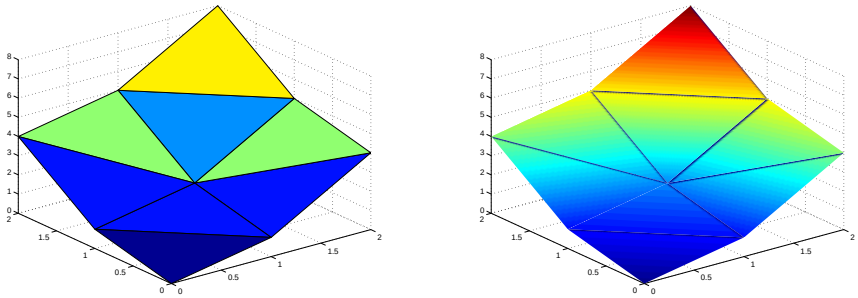

Figura 3.8: Dos versiones del `script6`. En la primera hemos eliminado el suavizado de color y se ve la triangulación subyacente. Nótese que no se suaviza la gráfica, sólo el coloreado de la misma

3.4.2. Lectura de puntos con ratón

En ocasiones resulta útil leer puntos pinchados con el ratón sobre una gráfica existente (que podría estar en blanco). La orden `ginput` lee las coordenadas locales (respecto del rango que existiera en la gráfica) del punto pinchado con el ratón y permite emplearlas para cálculos o gráficas.

`script7.m`

```
figure
axis([0 1 0 1]);                    % crea el cuadro de dibujo
[x,y]=ginput(4);                    % lee cuatro puntos
plot(x,y),axis([0 1 0 1]);
[x y]
```

3.4.3. Líneas de nivel sobre la superficie

Este es un pequeño ejemplo más avanzado para dibujar una superficie y las líneas de nivel sobre ella.

`script8.m`

```
[x,y]=meshgrid(-1:0.03:1,-1:0.03:1);
z=x.^2-y.^2;
surf(x,y,z),shading flat
hold on
contour3(x,y,z,'k')                % 'k' es el negro
```

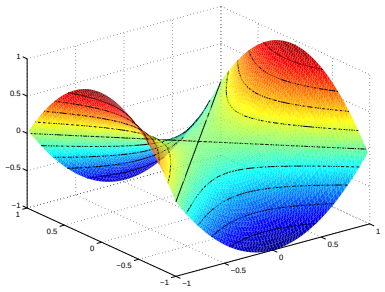

Figura 3.9: Gráfica producida por el `script8`

3.5. Tablas de instrucciones y opciones para gráficos

Instrucciones de dibujo bidimensional

Función	Utilidad
`plot(y)`	dado un vector y, dibuja los puntos $(1, y_1) \cdots (n, y_n)$ y los enlaza con segmentos.
`fplot('f',[a,b])`	dibuja la función $f(x)$ para valores de x entre a y b.
`ezplot('f')`	dibuja la función $f(x)$ para valores de x entre -2π y 2π.
`polar(t,r,opciones)`	gráfica en polares, t vector de ángulos en radianes, r radio vector.
`fill(x,y,'c')`	dibuja un polígono 2-D definido por los vectores columna x e y con el color indicado por `c`.
`bar(x,y)`	gráfico de barras.
`stairs(x,y)`	gráfico de escaleras.
`hist(y)`	histograma
`stem(y,opciones)`	gráfica stem

Colores y marcadores

Símbolo	Color	Símbolo	Estilo de línea
`y`	amarillo	$\cdot$	punto
`m`	magenta	$\circ$	círculo
`c`	cian	$\times$	marca-x
`r`	rojo	$+$	más
`g`	verde	$*$	estrella
`b`	azul	$-$	línea continua
`w`	blanco	:	línea punteada
`k`	negro	$-.$	línea punto-raya
		$--$	línea de trazos
		`'p'`	estrella de cinco puntas

Además de los colores listados, los demás colores se pueden obtener con un vector de tres componentes, que dan el estándar RGB para obtenerlo.

Opciones para dibujos bidimensionales

Función	Utilidad
`xlabel('texto')`	texto en el eje de abscisas
`ylabel('texto')`	texto en el eje de ordenadas
`title('texto')`	texto en la parte superior de la gráfica
`text(x,y,'texto')`	texto en el punto (x,y)
`gtext('texto')`	utiliza el ratón para colocar el texto
`legend('texto1','texto2'...,p)`	permite poner leyendas en la gráfica en el lugar de la ventana indicado por p 1 superior derecha (defecto), 2 superior izquierda, 3 inferior izquierda...
`grid on`	añade una rejilla a la gráfica actual
`grid off`	elimina la rejilla
`grid`	cambia el modo de rejilla
`axis([xmin,xmax,ymin,ymax])`	establece valores mínimo y máximo de los ejes
`axis on`	incluye los ejes (por defecto)
`axis off`	elimina la rejilla y los ejes
`axis(axis)`	una vez activado `hold on`, fija el escalado de los ejes para todas las gráficas posteriores
`axis equal`	aplica el mismo escalado para los ejes
`axis square`	crea la ventana de gráficos como un cuadrado
`axis auto`	vuelve al escalado para los ejes por defecto
`hold on`	mantiene activa la ventana gráfica actual
`hold off`	borra la figura y crea una nueva (por defecto)
`hold`	cambia el modo `hold`
`subplot(m,n,p)`	divide la ventana gráfica en `m` por `n` subventanas y coloca la gráfica actual en el lugar `p`-ésimo (de izquierda a derecha y de arriba a abajo).
`subplot(1,1,1)`	devuelve al modo por defecto.

Nótese que tanto `grid` como `hold` tienen una versión sin opción `on` u `off`. Corresponde a cambiar de una a la otra, sea cual sea la que esté escogida. Las dimensiones de los ejes de las gráficas siempre se ajustan (salvo que se empleen opciones de `axis`) para que la figura esté en proporción áurea, esto es, que

$$\frac{\text{longitud}}{\text{altura}} = \frac{1+\sqrt{5}}{2} \approx 1{,}6180.$$

Gráficas tridimensionales

Función	Utilidad
`plot3(X,Y,Z,'opciones')`	dibuja una gráfica línea si `X`, `Y`, `Z` son matrices, dibuja por columnas
`mesh(X,Y,Z,'opciones')`	dibuja una gráfica de malla
`meshc(X,Y,Z,'opciones')`	dibuja una gráfica de malla con curvas de nivel en 2-D
`surf(X,Y,Z)`	gráfica de superficie, ahora se rellenan los espacios entre líneas
`surfc(X,Y,Z)`	gráfica de superficie con curvas de nivel en 2-D
`fill3(X,Y,Z,'c')`	dibuja un polígono 3-D con el color c
`contour(X,Y,Z)`	gráfica de contorno en 2-D
`contour3(X,Y,Z)`	gráfica de contorno en 3-D
`clabel(contour())`	añade etiquetas de altura a los gráficos de contorno.

Ejercicios propuestos

1. La instrucción `surf` admite un cuarto parámetro, que da el color (por defecto el color es proporcional a la altura). Haz una gráfica de la función

$$x^2 - y^2 + 2\,z$$

definida sobre la superficie de la esfera $x^2 + y^2 + z^2 = 1$. Para ello sigue los siguientes pasos. (a) Se considera un mallado uniforme de las variables longitud y latitud (emplea `linspace` y `meshgrid`).

$$\phi \in [0, 2\pi], \qquad \theta \in [-\pi/2, \pi/2]$$

(b) Con ello y las ecuaciones

$$x = \cos\phi\,\cos\theta, \qquad y = \operatorname{sen}\phi\,\operatorname{sen}\theta, \qquad z = \operatorname{sen}\theta$$

se construyen tres matrices, correspondientes a las líneas paramétricas sobre la esfera (meridianos y paralelos). (c) Se evalúa la función de tres variables $c = f(x, y, z)$ (hay que definir la función de forma vectorizada). (d) Finalmente se emplea la instrucción de dibujo.

2. Repite el proceso anterior construyendo una función

```
function dibujoEsfera(f)
```

que reciba como argumento una función vectorizada de tres variables.

3. Construye la función vectorizada

$$f(x) = \begin{cases} x + 1, & -1 \le x \le 0, \\ 1 - x, & 0 \le x \le 1, \\ 0, & \text{en otro caso.} \end{cases}$$

Dibuja su gráfica en $[-2, 2]$ empleando `plot` y `fplot`.

4. Averigua el funcionamiento de la opción `view` para tener una vista cenital de una figura. Pruébala con un dibujo tipo `surf` y con otro tipo `mesh`.

5. La bola unidad en norma p ($1 \leq p < \infty$) se define como el conjunto de los puntos (x, y) tales que
$$(|x|^p + |y|^p)^{1/p} < 1.$$
Haz un script que permita, para un valor arbitrario de p dibujar la forma del conjunto. Puedes emplear instrucciones de tipo `surf` siempre que cambies el punto de vista a que sea zenital. Otra opción es emplear `pcolor`.

6. Averigua qué hace la instrucción `colorbar` cuando se añade a una serie de órdenes gráficas.

7. Las instrucciones `image` e `imagesc` dibujan los elementos de una matriz, asignándoles un color en función de la altura de su valor. Averigua cómo funcionan. Nota que una de las dos escala las alturas entre la mínima y la máxima. Nota además que los elementos se dibujan en el mismo orden en el que se escriben en la matriz. Si la matriz la has obtenido evaluando una función de dos variables sobre una malla obtenida con `meshgrid`, el eje Y sale del revés. ¿Por qué?

8. Averigua cómo funciona `quiver` para dibujar campos vectoriales en el plano. Haz una representación del campo vectorial
$$\left(\frac{-y}{\sqrt{x^2+y^2}}, \frac{x}{\sqrt{x^2+y^2}} \right).$$

9. Si en la instrucción `plot(X,Y)` ambos argumentos son matrices de la misma dimensión, ¿qué se dibujan? ¿Las filas o las columnas?

Ejercicios más avanzados

1. Se da una triangulación cualquiera, por ejemplo
```
[x,y]=meshgrid(0:2,0:2);
trii=[1 2 4;2 4 5;2 3 5;3 5 6;4 5 7;7 5 8;5 6 8;6 8 9];
```
¿Cómo se puede dibujar una vista plana de esta triangulación de una forma simple?

2. Si haces una disposición `subplot(44x)`, ¿cómo accedes a las posiciones a partir de la décima sin crear confusión?

3. Intenta reproducir la gráfica de la Figura 3.10. Los dibujos no son relevantes, sino las proporciones que ocupan. En la columna de la derecha las proporciones son 1:2:1. (**Nota.** Se pueden fusionar posiciones de un `subplot` con vectores en vez de índices).

4. Haz un script que solicite cuántos puntos se quieren introducir, luego lea de pantalla las coordenadas de esos puntos y termine dibujando un polígono cerrado (relleno de algún color) con esos puntos.

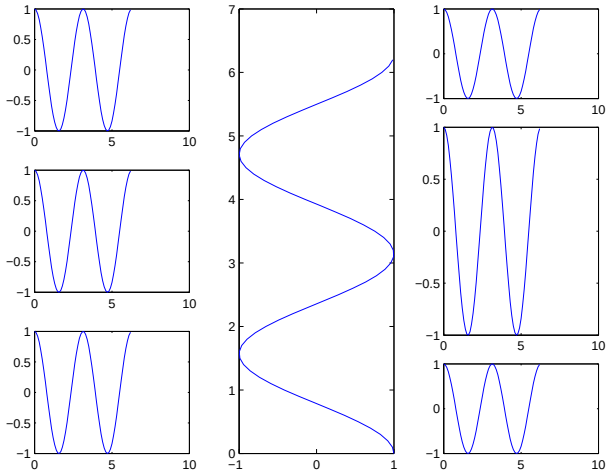

Figura 3.10: Intenta reproducir esta gráfica

Capítulo 4

Programación y tipos de datos

4.1. Estructuras de repetición

La instrucción de repetición más empleada es `for`, que se utiliza para crear bucles. Nota, no obstante, que MATLAB permite hacer bucles implícitos gracias a la creación automática de listas y a la vectorización de muchas operaciones. Esto permite que muchas operaciones que en un lenguaje de programación tradicional se realizarían con bucles, en MATLAB se pueden hacer mucho más eficientemente sin ellos.
La estructura de un `for` es:

```
for var=inicio:incremento:fin
     sentencias
end
```

Como de costumbre si no ponemos el incremento, se toma el valor por defecto, que es la unidad. La lista

```
inicio:incremento:fin
```

se crea al inicio de la ejecución del `for` y seguidamente se va recorriendo por la variable `var`. La lista puede ser de cualquier tipo de números.

`fibo.m`

```
function a=fibo(n)
% Calcula n terminos de la sucesion de Fibonacci

a=ones(1,n);        % reserva ya la memoria necesaria
for i=3:n
    a(i)=a(i-1)+a(i-2);
end
```

Ahora para calcular unos cuantos términos se ejecuta como sigue.

```
>> fibo(3)
ans =
     1     1     2
>> fibo(10)
ans =
     1     1     2     3     5     8    13    21    34    55
```

Para convencernos de que la lista no tiene porqué ser de números distintos, podemos emplear el siguiente script. Su funcionamiento es simple: se crea una lista de valores y luego se va recorriendo de uno en uno; en cada caso se muestra el cuadrado del número que está en el iterador `i`.

`script.m`

```
lista=fibo(n);
for i=lista
     disp(i.^2)
end
```

El iterador puede ser un vector. Por ejemplo, prueba a ejecutar el siguiente script.

`script.m`

```
lista=[1 2 3;2 3 4];
for j=lista
    disp(j)
end
```

En cada avance del iterador, dispones en `j` de un vector columna (la matriz `lista` se recorre mirándola como una lista de columnas). Esto es muy útil para recorrer triangulaciones: en lugar de numerar los triángulos, vas recorriendo la lista de los índices de los vértices que componen los triángulos.
Algo menos común, pero aún así muy útil, es la estructura `while`.

```
while condicion
    sentencias
end
```

4.2. Condicionales

Las construcciones condicionales son las siguientes, de más simple a más compleja:

- Condicional simple

```
if condicion
    sentencias
end
```

- Condicional simple con caso por defecto

```
if condicion
    sentencias
else
    otras sentencias
end
```

- Condicional de decisión múltiple (el `else` final, que cubre los demás casos, es opcional)

```
if cond_1
    sentencias
elseif cond_2       % se cumple cond_2 pero no cond_1
    otras sentencias
elseif cond_3
    ...
else
    ...
end
```

script.m

```
% Script donde se usa tambien la lectura y escritura
a=input('introduce el valor de a:')
b=a^2;
if b==0
    disp('sale 0')
else
    disp('sale distinto de 0')
end
```

- La orden `break` se utiliza para terminar la ejecución de los bucles for y while. Sale del bucle más interno en el que está contenido.
- Una orden `continue` dentro de un conjunto de sentencias de un bucle pasa directamente a la siguiente iteración, sin completar el conjunto de sentencias para ese valor del iterador.

Los símbolos de comparación son:

==	igual
~=	distinto
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que

El resultado de una operación de comparación es una variable lógica (con valores 0 ó 1). Las conectivas lógicas son

&	y (and)
`and(a,b)`	otra forma de escribir `a` & `b`
|	o (or)
`or(a,b)`	otra forma de escribir `a` | `b`
`xor(a,b)`	o exclusivo de las expresiones `a` y `b`
~	negación lógica

4.3. Estructuras de datos

Una estructura de datos es una agrupación de arrays del mismo tipo dentro de un array superior. La idea básica de una estructura de datos (existen en muchos lenguajes de programación de alto nivel) es la de un fichero, donde cada ficha contiene varios registros. El número de registros por ficha es fijo y el tipo de registro guardado también, aunque en el caso de que un registre guarde una matriz, distintas fichas pueden contener esos registros con distinta longitud.
Los siguientes ejemplos pueden emplearse para hacerse una idea de cómo funcionan las estructuras de datos en MATLAB a su nivel más básico. La orden `save` puede guardar estructuras de datos

`script1.m`

```
persona.nombre='Perico';
persona.edad=24;
persona.notas=[3, 5.4, 7, 9.5];

persona
persona.notas
media=sum(persona.notas)/length(persona.notas)
pause
                          %  a\~{n}adimos una segunda persona
persona(2).nombre='Fred';
persona(2).edad=26;
persona(2).notas=[4 5 2];  % puede tener otra longitud

persona(2)
persona
disp(persona(2).notas(3))  % tercera nota de la segunda persona
```

```
disp('numero de elementos'),length(persona)
pause

persona(:).edad            % son varias respuestas; no un array
edades=cat(1,persona(:).edad)  % lista de edades (hay que pegar)
edades=cat(2,persona(:).edad)  % 2=en horizontal
calificaciones=cat(2,persona(:).notas) % horizontal
```

- En cuanto se añade un segundo elemento con la misma estructura, una asignación de la forma `persona.nombre='Perico'` ya no funcionará.
- `cat(1,persona(:).notas` sólo funcionaría si las listas de notas tuvieran la misma longitud, ya que intentamos pegar varias filas.

genera.m

```
function u=genera(m)
% u=genera(m) \qquad m=numero de elementos no nulos
for i=1:m
     disp(i);
     aux(1)=input('posicion del elemento (enter tras cada valor)');
     aux(2)=input('');
                  % pido dos numeros y los empaqueto en un vector
     u(i).posicion=aux(:);     % columna
     u(i).elemento=input('elemento: ');
end
```

Cuando se llama a esta función creará una estructura de datos con el nombre correspondiente y las extensiones `matriz.posicion` y `matriz.elemento`.

```
>> matriz=genera(5)
```

4.4. Cell arrays

Un *cell array* es una estructura parecida a una matriz con la diferencia de que cada elemento puede ser una variable de un tipo distinto: un elemento puede ser a su vez una matriz numérica, una variable carácter, un vector. En el siguiente script se muestra un ejemplo muy simple de *cell array*, incluyendo cómo puede uno acceder a un elemento de una matriz que está guardada dentro de un cell array.

Para distinguir los cell arrays de las matrices se emplean llaves, tanto sustituyendo a los corchetes a la hora de construirlos como a los paréntesis a la hora de referirse a una posición dentro del cell array.

`script.m`

```
A=[2 3; 7 1];
b=[1; 3];
C=eye(3)+ones(3);
d=[7 7 1]';
ejemplos={'ej1','ej2'; A, C; b,d}; % cell array
ejemplos{1,1}
ejemplos{2,1}
pause
cellplot(ejemplos) % dibujo de la 'estructura'
pause
ejemplos{1,1}='ejemplo #1';
ejemplos{2,1}(2,2)=2; % posicion (2,2) en la matriz {2,1}
ejemplos{:,1}
ejemplos{1,:}
```

4.5. Número variable de argumentos de entrada

Es muy común en funciones de MATLAB que el número de argumentos de entrada sea variable. Por ejemplo, aquí damos una función que dibuja en un intervalo $[a, b]$ tantas funciones como queramos. A la hora de definirlo, el conjunto de argumentos variables se refieren siempre como `varargin` y constituyen un `cell array`. Lo primero que hay que hacer es averiguar su longitud.

`dibuja.m`

```
function dibuja(a,b,varargin)

k=nargin-2;      % nargin=numero argumentos de entrada
hold on
for i=1:k
     fplot(varargin{i},[a b]);
end
hold off
return
```

Ahora construimos dos funciones simples. Tienen que estar vectorizadas porque vamos a emplear `fplot`

`f.m`

```
function y=f(x)
y=cos(x)+sin(x);
```

g.m

```
function y=g(x)
y=cos(x);
```

A la hora de emplear `dibuja` podemos dar tantas funciones como queramos. El argumento que enviamos es el nombre de la función (¡por eso va entre comillas!)

```
>> dibuja(0,2*pi,'f')
>> dibuja(0,2*pi,'f','g')
```

Resolución de algunos ejercicios del Capítulo 1

Los ejercicios listados como [A1], etc son los de la lista de ejercicios más avanzados.

2. (b) `-sort(-A)` ordena por columnas de forma descendente
   (c) `(sort(A'))'` es el orden ascendente por filas
   (e) `max(abs(A(:)))` o bien `max(max(abs(A)))`

4. `7*ones(3,5)` o también `7.*ones(3,5)`

5. Para hacer una distribución uniforme en $[-5, 5]$ basta hacer

   ```
   10*rand(4,4)-5
   ```

6. Si se hace `floor` en el anterior ejercicio, se trunca el número a su parte entera por debajo. Esto genera en esencia una distribución uniforme que toma los valores entre -5 y 4. Para que sea de -5 a 5, se haría

   ```
   floor(11*rand(4,4)-5)
   ```

A1. `A-diag(diag(A))`

A2. `A-diag(diag(A))+diag(vector)`

A3. `reshape(sort(A(:)),size(A))`

A4. `A(find(A<0))=0` o también la forma más matemática `(A+abs(A))/2`

A5. `A(find(A>-1 & A<1))=0`

A6. Se pueden hacer las siguientes opciones

```
length(A(:)) max(size(A(:)))
prod(size(A))           % numero de filas por numero de columnas
size(A(:),1)            % size(B,1) es el numero de filas de B
[n,m]=size(A(:))        % en este caso el resultado esta en n
```

Resolución de algunos ejercicios del Capítulo 3

Los ejercicios listados como [A1], etc son los de la lista de ejercicios más avanzados.

2. La construcción más simple es la siguiente:

```
function dibujoEsfera(f)

phi=linspace(0,2*pi,150);
theta=linspace(-pi/2,pi/2,150);
[phi,theta]=meshgrid(phi,theta);
x=cos(phi).*cos(theta);
y=sin(phi).*cos(theta);
z=sin(theta);
c=feval(f,x,y,z);
surf(x,y,z,c),axis equal, shading interp
```

4. La opción `view(2)` pone el punto de vista cenital. Con `view(3)` se tiene al habitual punto de vista tridimensional.

5. Para un `p` dado, se puede aplicar el siguiente script:

```
[x,y]=meshgrid(linspace(-2,2,200));
z=zeros(size(x));
z(find(abs(x).^p+abs(y).^p<1))=1;
surf(x,y,z),view(2),axis equal,shading flat
```

7. Probar el siguiente experimento

```
imagesc([1 2 3;4 5 6]), colorbar
```

   Queda claro que dibuja en el mismo orden que la matriz. La esquina superior derecha de la matriz (el elemento (1,1)) está en la esquina superior derecha del dibujo. Al dibujar gráficas tendemos a poner el origen abajo y el eje Y en dirección ascendente, pero en las matrices numeramos en sentido contrario.

8. Lleva cuatro argumentos: eje horizontal, vertical, componente horizontal, componente vertical

```
[x,y]=meshgrid(linspace(-1,1,14));  % tomo 14 para evitar (0,0)
u=-y./sqrt(x.^2+y.^2);
v=x./sqrt(x.^2+y.^2);
quiver(x,y,u,v);
```

9. Las columnas.

A1. Hay varias formas:

```
trisurf(trii,x,y,zeros(size(x))), view(2)
```

   la dará en el color por defecto (que es verde).

A2. ¡Con comas! Por ejemplo: `subplot(4,4,11)`.

A3. Un pequeño script

```
x=0:0.1:2*pi;y=cos(2*x);
subplot(331),plot(x,y)
subplot(334),plot(x,y)
subplot(337),plot(x,y)
subplot(132),plot(y,x)
subplot(433),plot(x,y)
subplot(4,3,12),plot(x,y)
subplot(4,3,[6,9]),plot(x,y)
```

Obsérvese la fusión de las posiciones sexta y novena de una disposición gráfica cuatro por tres. El efecto de zona vacía a la derecha de la gráfica se puede corregir imponiendo los límites de los ejes. Aquí está tal y como sale por defecto en una versión 6.5.

A4. Por ejemplo,

```
n=input('Cuantos puntos: ');
[x,y]=ginput(n);
fill(x,y,'b')
```

Material por venir

En fechas próximas, ampliaremos este curso con una introducción a los siguientes tópicos:

- Empleo del paquete simbólico de MATLAB
- Empleo de las rutinas numéricas más comunes