Fundamentos de Computación Científica
Facultad de Ciencias del Mar

Introducción a la programación con Matlab

Angelo Santana

1. Matlab como calculadora. Funciones matemáticas elementales

- Elementos Básicos de Matlab
- Matlab como calculadora
- Lista de funciones básicas en Matlab

Elementos Básicos de Matlab

Comenzaremos este capítulo describiendo los elementos básicos del lenguaje de programación de Matlab. Aprenderemos a utilizar Matlab como si fuese una simple calculadora, conociendo así algunas de las funciones con las que este lenguaje viene equipado. A continuación veremos las reglas de sintaxis que permiten estructurar estas funciones en códigos (programas) orientados a resolver tareas concretas.

Matlab como calculadora

La manera más elemental de utilizar Matlab es como si de una simple calculadora se tratara. Podemos utilizar las operaciones básicas (suma, resta, multiplicación, división, exponenciación):

```
>> 2+3
```

```
ans =  5
```

```
>> 4-9
```

```
ans = -5
```

```
>> 6*3
```

```
ans =  18
```

```
>> 9/2
```

```
ans =  4.5000
```

```
>> 3^2
```

```
ans =  9
```

Matlab permite ejecutar fácilmente las funciones matemáticas más usuales (exponencial, trigonométricas, logaritmos …). Hay que destacar que las funciones trigonométricas operan en radianes, y el logaritmo neperiano es `log` y no `ln`. Veamos algunos ejemplos:

- El número π se representa como `pi`:

```
>> pi
```

```
ans =  3.1416
```

- La función *seno* es `sin()`. Así, el seno de $\pi/2$ se calcularía en matlab como:

```
>> sin(pi/2)
```

```
ans =  1
```

- La función exponencial `exp(x)` calcula el valor de e^x. El valor de e^2, por ejemplo, se obtiene como:

```
>> exp(2)
```

```
ans =  7.3891
```

- El logaritmo neperiano de 100 es:

```
>> log(100)
```

```
ans =  4.6052
```

- También están definidas las funciones `log2(x)`, que calcula el logaritmo de `x` en base 2, y `log10(x)` que calcula el logaritmo de `x` en base 10:

```
>> log10(100)
```

```
ans =  2
```

```
>> log2(100)
```

```
ans =  6.6439
```

Lista de funciones básicas en Matlab

Mostramos a continuación un pequeño listado de las funciones matemáticas básicas en matlab

Funciones matemáticas elementales en Matlab

Nombre	Función
cos	Coseno de un ángulo (en radianes)
sin	Seno de un ángulo (en radianes)
tan	Tangente de un ángulo (en radianes)
exp	Función exponencial (e^x)
log	Logaritmo neperiano
log10	Logaritmo en base 10
sinh	Seno hiperbólico
cosh	Coseno hiperbólico
tanh	Tangente hiperbólica

Nombre	Función
acos	Arco coseno
acosh	Arco coseno hiperbólico
asin	Arco seno
asinh	Arco seno hiperbólico
atan	Arco tangente
atanh	Arco tangente hiperbólica
abs	Valor absoluto
sign	Signo de un número (-1 ó +1)
round	Redondeo al entero más cercano
floor	Redondeo por defecto
ceil	Redondeo por exceso
fix	Redondeo hacia cero
rem	Resto de la división entera
sqrt	Raíz cuadrada

2. Variables numéricas: Vectores y matrices en Matlab

- Variables
- Creación de vectores y matrices.
- Dimensión de una matriz
- Concatenar matrices o vectores
- Matriz traspuesta
- Matriz inversa
- Determinante de una matriz
- Selección y/o modificación de elementos o partes de un vector o matriz
- Operaciones con vectores y matrices.
    - Suma
    - Producto
    - Producto de matrices
    - Producto de matriz por vector
    - Producto escalar de dos vectores
    - Operaciones de una matriz con un escalar
    - Operaciones con la matriz inversa

Variables

En un lenguaje de programación una variable es un espacio en la memoria del ordenador al que se le ha asignado un nombre; el contenido de dicho espacio puede cambiar a lo largo de una sesión de trabajo o durante la ejecución de un programa (precisamente por eso se denomina *variable*).

Por ejemplo, la ejecución del comando:

```
>> a = 25
```

```
a =  25
```

crea un espacio llamado `a` en la memoria, en el que se guarda el valor 25. Decimos entonces que hemos creado la variable `a`, cuyo valor inicial es 25. Si ahora hacemos:

```
>> a = 30
```

```
a =  30
```

habremos cambiado el valor de `a` del anterior 25 al nuevo valor 30. Como iremos viendo a lo largo de esta asignatura, las variables pueden ser de distintos tipos (numéricas, de texto, lógicas, de fecha, …). Asimismo el contenido de una variable puede ser un único valor como en este ejemplo, o puede ser un vector, una matriz, o una estructura más compleja.

Creación de vectores y matrices.

En Matlab un vector se crea simplemente escribiendo los valores que lo componen entre corchetes. La siguiente linea crea el vector $(1, 2, 3)$ y lo asigna a la variable `V`:

⊕

```
>> V=[1 2 3]
```

```
V =

   1   2   3
```

Los componentes de un vector se pueden declarar, tal como acabamos de hacer, simplemente separándolos por espacios. Alternativamente, en lugar de espacios podemos

separar los componentes del vector por comas; el resultado es el mismo:

```
>> V=[1, 2, 3]
```

```
V =


   1   2   3
```

Una matriz es igualmente fácil de crear: puede definirse fila por fila en lineas separadas:

⊕

```
>> A=[1 2 3 4 5
    6 7 8 9 10
    11 12 13 14 15]
```

```
A =


    1    2    3    4    5
    6    7    8    9   10
   11   12   13   14   15
```

O también pueden definirse las distintas filas en una sola linea, si bien en tal caso hay que separarlas con punto y coma:

```
>> A=[1 2 3 4 5;  6 7 8 9 10;  11 12 13 14 15]
```

```
A =


    1    2    3    4    5
    6    7    8    9   10
   11   12   13   14   15
```

Por defecto, cada vez que declaramos un vector o una matriz, Matlab nos muestra inmediatamente a continuación la asignación realizada. Si queremos evitar este comportamiento, bastará con terminar la declaración del vector (o matriz) con un punto y coma:

```
>> V=[1 2 3];
```

```
>> A=[1 2 3 4 5
    6 7 8 9 10
    11 12 13 14 15];
```

En general para Matlab los vectores son también matrices con una única fila (o una única columna).

Dimensión de una matriz

⊕ La función `size` nos devuelve el tamaño (número de filas y columnas) de una matriz;

```
>> size(A)
```

```
ans =


   3   5
```

Si especificamos `size(A,1)` obtenemos el **número de filas** de A:

```
>> size(A,1)
```

```
ans =  3
```

Asimismo, `size(A,2)` nos devuelve el **número de columnas** de A:

```
>> size(A,2)
```

```
ans =  5
```

⊕ La función `numel` devuelve el número total de elementos de una matriz:

```
>> numel(A)
```

```
ans =  15
```

⊕ Por último, la función `length` aplicada a un vector nos da su dimensión (número de elementos que lo componen); aplicada a una matriz, nos devuelve su número de columnas:

```
>> length(V)
```

```
ans =  3
```

```
>> length(A)
```

```
ans =  5
```

Concatenar matrices o vectores

Octave es capaz de concatenar o “pegar” matrices o vectores en una nueva matriz o vector siempre que las dimensiones originales encajen adecuadamente. Por ejemplo si hacemos:

```
>> A=[1 2; 3 4]
```

```
A =

   1   2
   3   4
```

```
>> B= [8; 7]
```

```
B =

   8
   7
```

⊕

- “Pegamos” B a la derecha de A:

```
>> C=[A B]
```

```
C =

   1   2   8
   3   4   7
```

- Añadimos una nueva fila a la matriz anterior:

```
>> D=[2 3 5]
```

```
D =
```

```
   2    3    5
```

```
>> [A B; D]
```

```
ans =


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

- “Pegamos” el vector V a la derecha de V:

```
>> [V V]
```

```
ans =


   1    2    3    1    2    3
```

- “Pegamos” el vector V debajo de V, creando una matriz de dimensión 2×3:

```
>> [V; V]
```

```
ans =


   1    2    3
   1    2    3
```

Matriz traspuesta

La traspuesta de una matriz (o vector) se obtiene simplemente añadiendo un apóstrofe (’) al nombre de la matriz o vector:

```
>> A=[1 2 3;  4 5 6;  7 8 9];
```

⊕

```
>> A'
```

```
ans =


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

```
>> V=[1 2 3];
```

⊕

```
>> V'
```

```
ans =

   1
   2
   3
```

Matriz inversa

La inversa de una matriz se obtiene mediante la función `inv()`:

```
>> A=[1 1; 1 2];
```

⊕

```
>> inv(A)
```

```
ans =

   2  -1
  -1   1
```

En caso de que la matriz no tenga inversa, Matlab nos muestra un aviso (*warning*):

```
>> B=[1 1; 2 2];
```

```
>> inv(B)
```

```
warning: matrix singular to machine precision
ans =

   Inf   Inf
   Inf   Inf
```

Determinante de una matriz

El determinante de una matriz se obtiene mediante la función `det()`:

```
>> A=[1 -3; 2 5];
```

⊕

```
>> det(A)
```

```
ans =  11
```

Selección y/o modificación de elementos o partes de un vector o matriz

Una vez que hayamos definido un vector o matriz en Matlab podemos acceder a sus elementos especificando entre paréntesis, a continuación del nombre del vector o matriz, la posición (o posiciones) a que queremos acceder. Si por ejemplo, definimos la matriz A siguiente:

```
>> A=[4 6 8 7 9; 1 -2 -7 8 4; 2 3 5 5 7]
```

```
A =

   4   6   8   7   9
   1  -2  -7   8   4
   2   3   5   5   7
```

podemos seleccionar el término $A_{2,4}$ que ocupa la posición (2,4) (fila 2, columna 4) mediante:

⊕

```
>> A(2,4)
```

```
ans =  8
```

También podemos seleccionar fácilmente una submatriz, indicando las filas y columnas iniciales y finales:

⊕

```
>> A(2:3,3:4)
```

```
ans =

  -7   8
   5   5
```

Podemos hacer selecciones más complicadas; por ejemplo, si queremos seleccionar la submatriz formada por las filas 1 y 3 y las columnas 2 y 5:

```
>> A([1 3],[2 5])
```

```
ans =

   6   9
   3   7
```

Para seleccionar una fila completa especificamos ":" como índice de columna. Por ejemplo, para mostrar las dos primeras filas completas:

⊕

```
>> A([1 2],:)
```

```
ans =

   4   6   8   7   9
   1  -2  -7   8   4
```

Asimismo, para seleccionar una columna completa especificamos ":" como índice de fila. Por ejemplo, para mostrar sólo la cuarta columna:

⊕

```
>> A(:,4)
```

```
ans =

   7
   8
   5
```

Podemos utilizar las selecciones anteriores para modificar los valores correspondientes en la matriz; por ejemplo, si queremos cambiar **todos** los términos de la cuarta columna por unos:

```
>> A(:,4)=1
```

```
A =

   4   6   8   1   9
   1  -2  -7   1   4
   2   3   5   1   7
```

Para cambiar los términos de la segunda fila por los números 5,4,3,2,1 en ese orden:

```
>> A(2,:)=[5 4 3 2 1]
```

```
A =

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

Para borrar una fila (o columna) de una matriz la cambiamos por la **matriz vacía** `[]`. Por ejemplo, para borrar la tercera columna de la matriz anterior:

```
>> A(:,3)=[]
```

```
A =

   4   6   1   9
   5   4   2   1
   2   3   1   7
```

Y si ahora queremos borrar las filas 1 y 3:

```
>> A([1 3],:)=[]
```

```
A =

   5   4   2   1
```

Por último, si se asignan valores a partes de matrices o vectores que no se han definido

previamente, Matlab rellena el resto de la matriz
o vector con ceros:

```
>> d([1 3 5 7])=-1
```

```
d =


  -1   0  -1   0  -1   0  -1
```

```
>> H(3,4)=5
```

```
H =


   0   0   0   0
   0   0   0   0
   0   0   0   5
```

Operaciones con vectores y matrices.

Suma

```
>> V=[1 2 3];
```

```
>> U=[-1 3 4];
```

⊕

```
>> U+V
```

```
ans =


   0   5   7
```

⊕

```
>> A=[1 2 3; 4 5 6; 7 8 9]
```

```
A =


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

```
>> B=[0 1 -2; 3 2 -4; 1 1 1]
```

```
B =

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

```
>> A+B
```

```
ans =

    1    3    1
    7    7    2
    8    9   10
```

Producto

Producto de matrices

⊕

```
>> A*B
```

```
ans =

    9    8   -7
   21   20  -22
   33   32  -37
```

Producto de matriz por vector

⊕

```
>> A=[1 2 3; 4 5 6; 7 8 9];
 V=[1 2 3];
 A*V'
```

```
ans =

   14
   32
   50
```

Producto escalar de dos vectores

⊕

```
>> U*V'
```

```
ans =  17
```

Operaciones de una matriz con un escalar

Cuando se desea multiplicar una matriz o un vector por un escalar se utiliza el símbolo ".*" (punto seguido de asterisco). Así por ejemplo, para multiplicar por 2 todos los valores de la matriz *A* anterior:

```
>> 2.*A
```

```
ans =

    2    4    6
    8   10   12
   14   16   18
```

Si queremos sumar la misma cantidad a todos los términos de una matriz usamos “.+”. Para sumar 3 a todos los términos de la matriz *A* anterior:

```
>> 2.+A
```

```
ans =

    3    4    5
    6    7    8
    9   10   11
```

```
>> 3.*V
```

```
ans =

   3   6   9
```

Operaciones con la matriz inversa

Supongamos ahora que tenemos las siguientes matrices:

```
>> A=[1 3 0; 2 1 -3; 0 1 2];
 B=[2, 1, 4; 6 0 1; 1 0 0];
```

Si queremos multiplicar A por la inversa de B podemos ejecutar:

```
>> A*inv(B)
```

```
ans =

    3  -12   67
    1   -7   42
    1   -2   10
```

Nótese que, de alguna manera, multipicar A por la inversa de B viene a ser como dividir A por B. Matlab nos permite ejecutar esta operación como:

```
>> A/B
```

```
ans =

    3  -12   67
    1   -7   42
    1   -2   10
```

y obtenemos el mismo resultado que antes.

Si quisiéramos realizar el producto $B^{-1} \cdot A$, podemos calcularlo como:

```
>> inv(B)*A
```

```
ans =

    0    1    2
   -7   21   56
    2   -5  -15
```

pero matlab nos permite también utilizar, de manera equivalente, la siguiente notación algo más simple:

```
>> B\A
```

```
ans =


    0     1     2
   -7    21    56
    2    -5   -15
```

Nótese que podemos leer el símbolo “\”, en la operación `B\A` como que la primera matriz “divide” a la segunda.

3. Secuencias, matrices de números aleatorios y matrices especiales

- Secuencias
- Vectores y matrices de números aleatorios:
- Matrices especiales: Identidad, Diagonal, Unos, Ceros
  - Matriz Identidad
  - Matriz diagonal
  - Matriz de unos
  - Matriz de ceros

Secuencias

La sintaxis `a:b:c` genera la secuencia de valores que van de `a` a `c` en incrementos de magnitud `b`. Si se omite el valor de `b` se genera una secuencia de valores de `a` a `c` de uno en uno. En general una secuencia de valores es considerada por Matlab como un vector.

```
>> U=0:10
```

```
U =

    0    1    2    3    4    5    6    7    8    9   10
```

```
>> V=0:2:10
```

```
V =

    0    2    4    6    8   10
```

```
>> W=0:0.1:0.5
```

```
W =

    0.00000    0.10000    0.20000    0.30000    0.40000    0.50000
```

```
>> X=12:-3:-12
```

```
X =

   12    9    6    3    0   -3   -6   -9  -12
```

Vectores y matrices de números aleatorios:

La función `rand()` crea vectores o matrices de números aleatorios distribuidos uniformemente entre 0 y 1; si se especifica `rand(n)` se crea una matriz de dimensión $n \times n$; si se especifica `rand(m,n)` se genera una matriz de dimensión $m \times n$:

```
>> rand(1)
```

```
ans =  0.38783
```

```
>> rand(2)
```

```
ans =

   0.13337   0.95887
   0.93870   0.22270
```

```
>> rand(1,4)
```

```
ans =

   0.51418   0.57228   0.94543   0.73161
```

```
>> rand(2,3)
```

```
ans =

   0.482729   0.765492   0.448806
   0.075225   0.872439   0.782452
```

La función `randi()` es similar, pero genera números enteros:

- `randi(imax)`: genera al azar un número entero entre 1 e `imax`.
- `randi(imax, n)`: genera una matriz $n \times n$ de números enteros elegidos al azar entre 1 e `imax`.
- `randi(imax, m, n)`: genera una matriz $m \times n$ de números enteros elegidos al azar entre 1 e `imax`.
- `randi ([imin imax], ...)`: Igual que los comandos anteriores, pero eligiendo los números aleatorios entre `imin` e `imax`. Donde están los puntos suspensivos podemos poner la dimensión del vector o matriz a generar.

Ejemplos:

```
>> randi(7)
```

```
ans =  7
```

```
>> randi(7,2,2)
```

```
ans =

   4   4
   7   3
```

```
>> randi([3,9],1,10)
```

```
ans =

   6   9   9   5   8   5   6   3   3   5
```

Matrices especiales: Identidad, Diagonal, Unos, Ceros

Matriz Identidad

La matriz identidad de dimensión n se define en Matlab como `eye(n)`. La razón es que en inglés *eye* y la letra **I** (que es el símbolo utilizado habitualmente para la matriz identidad) suenan igual:

```
>> eye(3)
```

```
ans =

Diagonal Matrix

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

Matriz diagonal

En Matlab una matriz diagonal se construye fácilmente como:

```
>> diag([1 2 3])
```

```
ans =

Diagonal Matrix

   1   0   0
   0   2   0
   0   0   3
```

Se pueden generar matrices con los términos no nulos en alguna diagonal distinta de la principal. Por ejemplo, la siguiente sintaxis genera una matriz cuadrada con los valores (1,2,3) en la quinta diagonal (**NOTA**: téngase en cuenta que para Matlab la diagonal principal es la diagonal 0):

```
>> diag([1 2 3],5)
```

```
ans =

   0   0   0   0   0   1   0   0
   0   0   0   0   0   0   2   0
   0   0   0   0   0   0   0   3
```

```
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0
```

Matriz de unos

Podemos construir una matriz de unos de dimensión $m \times n$ como `ones(m,n)`:

```
>> ones(3,2)
```

```
ans =

   1   1
   1   1
   1   1
```

Si queremos que la matriz de unos sea cuadrada de orden n basta con `ones(n)`:

```
>> ones(3)
```

```
ans =

   1   1   1
   1   1   1
   1   1   1
```

Matriz de ceros

Asimismo una matriz de ceros de dimensión $m \times n$ se define como `zeros(m,n)`:

```
>> zeros(2,3)
```

```
ans =

   0   0   0
   0   0   0
```

Si queremos una matriz cuadrada de ceros de orden n basta con `zeros(n)`:

```
>> zeros(3)
```

```
ans =

   0   0   0
   0   0   0
   0   0   0
```

4. Variables de texto (strings y chars)

- Variables de texto de clase “cadena” (*strings*)
- Variables de texto de clase “carácter” (*char*)
- Funciones más habituales para operar con variables de texto:

Variables de texto de clase “cadena” (*strings*)

Matlab es también capaz de tratar con variables de texto. Por ejemplo, la siguiente sintaxis guarda la frase *“Estás usando el programa Matlab”* dentro de la variable `txt`:

⊕

```
>> txt = "Estás usando el programa Matlab"
```

```
txt =

    "Estás usando el programa Matlab"
```

Nótese que para definir la variable `txt` hemos escrito el texto entre **comillas dobles.** Cuando se usa esta clase de comillas, la variable que se crea es de clase *“string”* (cadena):

⊕

```
>> class(txt)
```

```
ans= string
```

Las variables de texto de tipo cadena (*string*) se pueden organizar en vectores o matrices igual que si fuesen números. Por ejemplo:

⊕

```
>> palabras = ["esto", "es", "un", "vector","de","texto"]
```

```
palabras =

  1×6 string array

    "esto"    "es"    "un"    "vector"    "de"    "texto"
```

Podemos seleccionar elementos de este vector de la manera habitual en matlab:

⊕

```
>> palabras(1)
```

```
ans=
  "esto"
```

```
>> palabras(4)
```

```
ans=
  "vector"
```

```
>> palabras(2:4)
```

```
ans =

  1×3 string array

    "es"    "un"    "vector"
```

La variable MP que definimos a continuación es una **matriz de texto** de clase *string* (pues utilizamos dobles comillas):

⊕

```
>> MP = ["Esto" "es"; "una" "matriz"; "de" "palabras"]
```

```
MP =

  3×2 string array

    "Esto"    "es"
```

```
    "una"     "matriz"
    "de"      "palabras"
```

```
>> MP(2,:)
```

```
ans =

  1×2 string array

    "una"    "matriz"
```

```
>> MP(3,2)
```

```
ans =

    "palabras"
```

Se puede utilizar el operador suma para combinar varias cadenas en una sola:

```
>> "programar " + "es " + "fácil"
```

```
ans =

    "programar es fácil"
```

Obsérvese que para que queden espacios entre las palabras ha habido que dejarlos explícitamente entre comillas.

Variables de texto de clase “carácter” (*char*)

Si queremos acceder individualmente a los caracteres que componen una variable de clase *string*, primero hemos de convertir dicha variable a clase *char* (carácter). Por ejemplo, para convertir nuestra variable inicial `txt` a carácter, basta con emplear la función `char()`:

⊕

```
>> txt = "Estás usando el programa Matlab";
>> t = char(txt)
```

```
t =

    'Estás usando el programa Matlab'
```

En principio lo único que parece haber sucedido es que la variable `t` es igual que la variable `txt` salvo que ahora tiene comillas simples en vez de dobles; ello indica que `t` es de clase `char` como podemos comprobar fácilmente:

```
>> class(t)
```

```
ans =

    'char'
```

Las variables de clase `char` se comportan como si fueran vectores de caracteres (letras, números, espacios u otros símbolos) consecutivos. Para seleccionar parte del texto basta indicar las posiciones a extraer; así, si queremos extraer las letras desde la posición 7 a la 13 del texto guardado en `t` bastaría con ejecutar:

⊕

```
>> t(7:13)
```

```
ans =

    'usando '
```

La función `size` nos da las dimensiones de `t`:

```
>> size(t)
```

```
ans =

     1    31
```

es decir, `t` es una matriz de 1 fila por 31 columnas (que es el número de caracteres que contiene).

Las variables de tipo carácter (`char`) se pueden declarar también directamente usando **comillas**

simples:

⊕

```
>> t2 = 'Matlab es un lenguaje curioso'
```

```
t2 =

    'Matlab es un lenguaje curioso'
```

⊕ Para concatenar dos variables de tipo carácter en un vector fila, basta usar corchetes igual que cuando se concatenan vectores numéricos:

```
>> [t ". " t2]
```

```
ans =

    'Estás usando el programa Matlab. Matlab es un lenguaje curioso'
```

Obsérvese que hemos añadido "." para que aparezca la separación entre las dos frases.

⊕ Para concatenar dos variables de tipo carácter en un vector columna, deben tener la misma longitud; ya sabemos que `t` tiene longitud 31; la variable `t2` tiene dimensiones:

```
>> size(t2)
```

```
ans =

    1   29
```

Como esta segunda frase tiene 29 caracteres, podemos añadirle dos espacios, para que tenga también 31 caracteres y así añadirla en una matriz junto con la frase `t`:

```
>> t2 = 'Matlab es un lenguaje curioso  ';
>> TT=[t;t2]
```

```
TT =

  2×31 char array
```

```
    'Estás usando el programa Matlab'
    'Matlab es un lenguaje curioso  '
```

`TT` es ahora una matriz de dimensión 2x31. Si seleccionamos, por ejemplo, las columnas de la 15 a la 20 obtenemos:

```
>> TT(:, 15:20)
```

```
ans =

  2×6 char array

    'l prog'
    'enguaj'
```

Funciones más habituales para operar con variables de texto:

- `strlength(v)`: muestra el número de caracteres del string `v`:

```
>> strlength("¿Cuantos caracteres tiene esta frase?")
```

```
ans =

    37
```

- `int2str(v)`: convierte valores enteros a texto.

```
>> int2str(123456)
```

```
ans =

    '123456'
```

- `num2str(v,n)`: convierte el valor numérico (o matriz) `v` en texto; si `v` tiene decimales y no se especifica valor de `n`, Matlab por defecto convierte `v` a

texto con 4 decimales como máximo. Si se especifica un valor de n Matlab convierte v en texto con un total de n cifras significativas (incluyendo parte entera y decimal). Si v es un vector (matriz) convierte cada valor numérico del vector (matriz) a texto; también convierte a texto los espacios entre los valores numéricos del vector (matriz).

```
>> num2str(2.13456,2)
```

```
ans = 2.1
```

```
>> num2str(2.13456,4)
```

```
ans = 2.135
```

```
>> num2str(21345.36,2)
```

```
ans = 2.1e+04
```

Podemos extraer los términos del 4 al 8, por ejemplo, de la expresión anterior:

```
>> a=num2str(21345.36);
>> a(4:7)
```

```
ans = 45.3
```

Veamos el resultado de aplicar la función `str2num()` a una matriz numérica:

```
>> a = num2str([1 2 3])
```

```
a = 1  2  3
```

Vemos que a es ahora una matriz de caracteres de dimensión 1$$7

```
>> size(a)
```

```
ans =

   1   7
```

Veamos quienes son los sucesivos términos de a:

```
>> a(1)
```

```
ans = 1
```

```
>> a(2)
```

```
ans =
```

```
>> a(3)
```

```
ans =
```

```
>> a(4)
```

```
ans = 2
```

Nótese que al convertir la matriz `[1 2 3]` en caracteres, Matlab considera que entre cada dos valores hay dos espacios en blanco.

- `str2num(s)`: convierte una cadena de texto en número (esto es, convierte, por ejemplo, la cadena "2" en el número 2, o la cadena "1 2 3" en el vector [1 2 3]; si le pasamos un carácter no numérico, devuelve un vector vacío)

- `str2double(s)`: convierte la cadena de texto `s` en un valor númerico de doble precisión. Al igual que la función anterior, si `s` contiene caracteres no numéricos, esta función devuelve un vector vacío.

- `mat2str(v)`: convierte matrices a caracteres; su funcionamiento es similar a `num2str()`, salvo que también convierte a caracteres los corchetes `[` y `]` que se utilizan para definir la matriz:

```
>> a = mat2str([1 2 3]);
>> a(1)
```

```
ans = [
```

- `ischar(v)`: función que nos indica si `v` es una variable tipo carácter (en cuyo caso `ischar` devuelve un 1) o no (en cuyo caso devuelve un 0).

- `strcmp(a,b)`: compara las cadenas `a` y `b`. Si son iguales devuelve un 1, en caso contrario devuelve un 0. Esta función distingue entre mayúsculas y minúsculas: la misma letra se considera distinta si está en mayúsculas que si está en minúscula.

```
>> a="Hola";
>> b="hola";
>> strcmp(a,b)
```

```
ans = 0
```

- `strcmpi(a,b)`: igual que `strcmp`, pero sin distinguir entre mayúsculas y minúsculas.

```
>> strcmpi(a,b)
```

```
ans = 1
```

- `a==b`: compara las cadenas `a` y `b` carácter a carácter, devolviendo un vector formado por unos y ceros (1 en las posiciones en que `a` y `b` coincidan, 0 cuando no).

```
>> a==b
```

```
ans =

  0  1  1  1
```

5. Variables lógicas e instrucciones condicionales.

Variables lógicas

En matlab las variables lógicas son aquellas que toman dos valores: true (que se codifica como 1) y false (que se codifica como 0).

Ejemplo: el siguiente vector toma valores lógicos:

```
>> A=[true true true]
```

```
A =


  1  1  1
```

En la práctica los valores lógicos se producen al hacer comparaciones. Por ejemplo, si definimos las variables:

```
>> a=7;
 b=5;
 c=7;
```

podemos compararlas entre sí, y la respuesta será un valor lógico. El operador *"¿es igual que?"* es el doble signo igual "=="

```
>> a==b
```

```
ans = 0
```

```
>> a==c
```

```
ans = 1
```

También podemos compararlas para ver si son distintas. El operador *"¿es distinta de"* es el símbolo igual precedido de una tilde: "~="

```
>> a~=b
```

```
ans = 1
```

NOTA: La tilde (~) en matlab significa negación. En la mayoría de los teclados (en español) se obtiene este símbolo pulsando las teclas `Alt Gr-4`. A veces se puede generar este símbolo pulsando la tecla `Alt` y (sin soltarla) tecleando los números 126 en el teclado numérico de la derecha.

Las comparaciones pueden hacerse también sobre vectores o matrices; en este caso la comparación se hace término a término y matlab devuelve un vector (o matriz) de unos y ceros. Por ejemplo:

```
>> u=[1 2 3 2];
 v=[2 2 2 2];
 u==v
```

```
ans =

  0  1  0  1
```

Operadores lógicos

Matlab cuenta con varios operadores que actúan sobre variables lógicas:

all

`all`: comprueba si **todos** los valores de una operación lógica son verdad.

```
>> all(u==1)
```

```
ans = 0
```

```
>> all(v==2)
```

```
ans = 1
```

```
>> all(u==v)
```

```
ans = 0
```

any

any: comprueba si **alguno o algunos** de los valores de una operación lógica son verdad.

```
>> any(u==1)
```

```
ans = 1
```

```
>> any(u==5)
```

```
ans = 0
```

```
>> any(u==v)
```

```
ans = 1
```

and

and: comprueba si dos valores lógicos son ambos verdaderos; se puede para ello usar el operador &:

```
>> a=7; b=5; c=7;
 a==c & b~=3
```

```
ans = 1
```

```
>> a==c & b~=5
```

```
ans = 0
```

o de modo equivalente se puede usar la función and:

```
>> and(a==c, b~=3)
```

```
ans = 1
```

```
>> and(a==c, b~=5)
```

```
ans = 0
```

or

`or`: comprueba si dos valores lógicos son ambos verdaderos; se puede usar para ello el operador `|`:

```
>> a==c | b~=3
```

```
ans = 1
```

```
>> a==c | b~=5
```

```
ans = 1
```

De modo equivalente se puede usar la función `or`:

```
>> or(a==c, b~=3)
```

```
ans = 1
```

```
>> or(a==c, b~=5)
```

```
ans = 1
```

find

`find` encuentra índices y valores de elementos verdaderos; por ejemplo, en el siguiente vector podemos encontrar qué posiciones ocupan los valores mayores que 5:

```
>> A = [2 8 5 9 12 3 2];
 find(A>5)
```

```
ans =

   2   4   5
```

Si aplicamos `find` a una matriz, usaremos la siguiente notación para que nos devuelva las filas y columnas donde se producen valores verdaderos:

```
>> A=[1 2 3; 4 5 6]
 B=[2 7 3; 8 5 9]
 [fila,columna]=find(A==B);
 [fila,columna]
```

```
A =

   1   2   3
   4   5   6

B =

   2   7   3
```

```
   8   5   9

ans =

   2   2
   1   3
```

Vemos que las matrices A y B coinciden en las posiciones (2,2) y (1,3)

Ver más información sobre variables y operaciones lógicas en el sitio web de Mathworks

Sentencias condicionales: `if` … `elseif`

Las sentencias condicionales en Matlab permiten hacer cosas (cálculos, gráficas, salidas por pantalla, …) sólo si se cumple determinada condición. En general son de la forma:

```
if (condición)
  < hacer cosas >
end
```

Ejemplo:

La función `rem(a,b)` calcula el resto de dividir `a` por `b`. La siguiente sintaxis emplea esta función y utiliza el condicional `if` para comprobar si un número es par:

```
a=8

if (rem(a,2)==0)
  disp('a es par')
end
```

El resultado es:

```
a =  8
a es par
```

La sentencia `if` puede acompañarse de `elseif` y de `else`:

```
a=9

if (rem(a,2)==0)
  disp('a es par')
elseif (rem(a,3)==0)
  disp('a es impar y divisible por 3')
else
  disp('a es impar no divisible por 3')
end
```

```
a =  9
a es impar y divisible por 3
```

Sentencias condicionales: `switch`

Cuando hay que comprobar muchas condiciones, en lugar de `if` a veces resulta más cómodo utilizar `switch`, cuya sintaxis es de la forma:

```
switch (X)
  case 1
    Hacer algo;
  case 2
    Hacer algo distinto;
  otherwise
    Hacer otra cosa;
end
```

En cada linea etiquetada como `case` se compara X con el valor que se especifica. La sintaxis de switch es equivalente a una colección de `if` - `elseif` encadenados.

Por ejemplo, si ejecutamos el código:

```
X=2

switch (X)
  case 1
    disp("X vale 1");
  case 2
    disp("X vale 2");
  otherwise
    disp("X no es ni 1 ni 2");
end
```

obtenemos como resultado

```
X =  2
X vale 2
```

Los valores especificados en `case` no tienen que ser necesariamente numéricos, también podrían ser caracteres:

```
X='c'

switch (X)
  case 'a'
    disp("X es la letra a");
  case 'b'
    disp("X es la letra b");
  otherwise
    disp("X no es ni a ni b");
end
```

```
X = c
X no es ni a ni b
```

6. Funciones

Definición de funciones en Matlab

Una función es una colección de operaciones cuyo objetivo es realizar una tarea particular. En general una función puede recibir uno o varios valores de entrada (`input1`, `input2`, …) y producir como salida uno o varios valores (`output1`, `output2`, …). También es posible que la función no reciba ninguna entrada o no produzca ninguna salida.

La sintaxis para definir una función en Matlab/Octave es la siguiente:

```
function [output1, output2, ...] = nombreFuncion(input1, input2, ...)
  < Hacer cosas (cuerpo de la función) >
end
```

Opcionalmente a continuación de la primera linea de una función y antes de su cuerpo podemos insertar una o varias lineas de comentario (precedidas por el símbolo %) que expliquen de manera sucinta qué hace la función. El texto de este comentario servirá como ayuda sobre el funcionamiento de la función y aparecerá en la ventana de comandos si el usuario teclea `help` seguido del nombre de la función.

Ejemplo: Construir una función con el nombre `rango` que reciba como entrada un vector de longitud

arbitraria n y devuelva como salida un vector formado por los valores mínimo y máximo de dicho vector

```
function y = rango(x)
  % Esta función calcula los valores mínimo y máximo de un vector o matriz x
  y = [min(x) max(x)];
end
```

En este ejemplo la función tiene una única salida (`y`) y una única entrada (`x`). Por defecto, Matlab/Octave asume que tanto `y` como `x` pueden ser vectores o matrices. Como hemos añadido una linea de comentario, ésta se muestra si el usuario pide ayuda sobre la función:

```
>> help rango
```

```
Esta función calcula los valores mínimo y máximo de un vector o matriz x
```

Ejecutemos ahora esta función, creando en primer lugar un vector de valores aleatorios:

```
>> v=rand(1,6)
```

```
v =

   0.991870   0.777889   0.593820   0.031225   0.947793   0.351061
```

y ahora utilizando la función `rango()` para encontrar sus valores mínimo y máximo:

```
>> rango(v)
```

```
ans =

   0.031225   0.991870
```

Como vemos, Matlab devuelve un vector con el mínimo y el máximo del vector v.

Alternativamente, esta función podría programarse del modo siguiente:

```
function [minx, maxx] = minmax(x)
  maxx = max(x);
  minx = min(x);
end
```

Nótese que en este caso, la función devuelve **explícitamente** un vector con dos términos, el mínimo y el máximo respectivamente. Al ejecutar esta función obtenemos como resultado:

```
>> minmax(v)
```

```
ans =  0.031225
```

¿Qué ha ocurrido aquí? Pues que, por defecto, cuando la salida de una función explícitamente declara varios términos distintos, al ejecutarla **sin asignar su resultado a ninguna variable** se muestra solamente el primero de sus valores de salida (en este ejemplo, el mínimo). Si queremos obtener todos los valores que proporciona la función la llamada a la misma debe realizarse del modo siguiente:

```
>> [minV, maxV]=minmax(v)
```

```
minV =  0.031225
maxV =  0.99187
```

esto es, asignando la salida de la función **explícitamente** a un vector de dimensión coincidente con la forma en que se ha definido la función; podemos ahora utilizar cada uno de los términos de este vector de salida:

```
>> minV
```

```
minV =  0.031225
```

```
>> maxV
```

```
maxV =  0.99187
```

Importante: para que una función devuelva un valor o valores, **las variables que se han declarado como salida en la propia definición de la función, deben calcularse explícitamente en el cuerpo de la función**. Así, en la función `rango` la salida es la variable `y`, que se calcula explícitamente en el cuerpo de la función. Asimismo en el cuerpo de la función `minmax` se calculan explícitamente las variables `maxx` y `minx` utilizadas en su definición. Si alguna de estas

variables no se calculase en el cuerpo de la función, el valor de dicha variable se devolvería como un vector nulo. Obsérvese lo que ocurre si definimos la función como:

```
function [minx, maxx] = minmax2(x)
  maxx = max(x);
  minimo = min(x);
end
```

Si nos damos cuenta, tal como la hemos declarado, la función debe devolver `minx` y `maxx`. Pero en el cuerpo de esta función no se calcula la variable `minx`. ¿Qué ocurre al ejecutarla?:

- Si la ejecutamos sin asignar su resultado a ninguna variable:

```
>> minmax2(v)
```

no se produce ninguna salida, pues el primer valor que devuelve dicha función es `minx` y esta variable no se ha calculado.

- Si la ejecutamos asignando su resultado al vector `[minV, maxV]`:

```
>> [minV, maxV]=minmax2(v);
```

```
warning: minmax2: some elements in list of return values are undefined
warning: called from
    minmax2 at line 4 column 1
```

```
>> minV
 minV = [](0x0)
```

```
>> maxV
 maxV =  0.99187
```

Guardar una función

En Matlab/Octave es posible declarar una función en la ventana de comandos. No obstante, esto entraña varias dificultades:

- Si la función es medianamente larga o compleja, es difícil escribirla pues la ventana de comandos no permite la edición

de manera sencilla (por ejemplo, no es posible volver a la línea anterior si nos hemos equivocado en una letra)

- La función sólo estará activa mientras dure nuestra sesión. Si la función hace alguna tarea interesante (o complicada) conviene tenerla archivada para poder utilizarla en otras ocasiones.

Por ello la mejor manera de programar funciones es utilizando la ventana de edición, que además ofrece ventajas como distintos colores según que se incluyan comentarios, funciones propias de matlab, etc, o indentación, que hace más claras las distintas partes de la función.

Una vez hayamos terminado de editar la función, debe guardarse en un archivo **con el mismo nombre de la función** y con la extensión .m. Matlab busca siempre los archivos .m en nuestro directorio `home` y en los directorios definidos por defecto para que Matlab busque dichos archivos. Se puede saber cuáles son dichos directorios ejecutando el comando `path`.

Resulta conveniente disponer de una carpeta específica para los archivos `.m` (por una mera cuestión de organización y de facilidad para encontrarlos). Para incluir dicha carpeta dentro de aquellas en las que Matlab busca por defecto los archivos `.m` se utiliza el comando:

```
>> addpath ruta
```

donde `ruta` es la dirección de dicha carpeta (por ejemplo `c:\Documents and settings\usuario\mis archivos\misFunciones`).

Ejemplo: la ecuación de segundo grado

La siguiente función encuentra las raíces de la ecuación de segundo grado:

$$ax^2 + bx + c = 0$$

```
function x = resuelveEc2G(a,b,c)
    discrim=b^2-4*a*c;
    x1=(-b-sqrt(discrim))/(2*a);
    x2=(-b+sqrt(discrim))/(2*a);
    x=[x1;x2];
end
```

Podemos aplicar esta función para hallar las raíces de $x^2 - x - 6 = 0$:

```
>> resuelveEc2G(1, -1, -6)
```

```
ans =

  -2
   3
```

Téngase en cuenta que matlab por defecto es capaz de operar con números complejos; por ello no hay que tomar ninguna precaución respecto a si el discriminante es positivo o negativo. La ecuación $x^2 + 2x + 5 = 0$ tiene dos raíces complejas:

```
>> resuelveEc2G(1,2,5)
```

```
ans =

  -1 - 2i
  -1 + 2i
```

Buenas prácticas de programación de funciones

Es una buena práctica de programación no acumular demasiadas tareas en una función. Si debemos realizar varias tareas independientes es mejor separarlas en varias funciones simples y usar una *"función principal"* que las vaya llamando en el orden adecuado. Ello hace más fácil detectar y corregir posibles fallos en la programación y permite reutilizar la misma función (simple) en varios contextos.

Por ejemplo, supongamos que queremos pedir al usuario que introduzca por pantalla los valores de los coeficientes de la ecuación de segundo grado, para a continuación resolverla y dibujar la parábola resultante entre los puntos de corte. Podemos descomponer este trabajo en tres funciones distintas:

- Una función que pida los valores de los coeficientes y compruebe que los valores son válidos (en este caso, que $a \neq 0$, porque si fuese $a = 0$ no tendríamos una ecuación de segundo grado).

- Una segunda función que resuelva la ecuación.

- Una función que represente la parábola y los puntos de corte.

La función que resuelve la ecuación ya la hemos construido antes. Para pedir al usuario que introduzca los valores de los coeficientes podemos usar la siguiente función:

```
function [a,b,c,error]=pideCoeficientesEc2G()
      % Esta función solicita los coeficientes de una
      % ecuación de segundo grado
      clc; % Limpia la pantalla
      disp('============================================');
      disp('programa para resolver ecuaciones de 2º grado');
      disp('============================================');
      disp(' ');
      % Introducción de los coeficientes desde teclado:
      disp('Introduzca los coeficientes de la ecuación');
      disp(' ');
      a = input('Valor del coeficiente a: ');
      b = input('Valor del coeficiente b: ');
      c = input('Valor del coeficiente c: ');
      if a==0
          error=true;
          disp('ERROR: El valor del coeficiente "a" no puede ser 0')
      else
          error=false;
      end
end
```

Asimismo, la siguiente función se encargaría de representar la parábola con los puntos de corte:

```
function dibujaParabola(coefs,raices)
% Esta función dibuja una parábola y muestra
% sus puntos de corte con el eje X
   % Se asignan las raíces a las variables x1 y x2:
    x1=min(raices);
    x2=max(raices);
    % Se fijan los límites de la gráfica de forma
    % que contengan las raices y quede margen a
    % los lados. Si hay una raíz única se fija
    % un margen de 2 unidades
    if x1==x2
        margen=2;
    else
        margen=(x2-x1)/4;
    limInf=x1-margen;
    limSup=x2+margen;
    % Se calculan los valores de la parábola entre
    % los dos límites elegidos
    x=linspace(limInf, limSup, 200);
    y=coefs(1)*x.^2+coefs(2)*x+coefs(3);
    % Se dibuja la parábola
    plot(x,y);
    % Se centran los ejes en el origen de coordenadas
    ax = gca;
    ax.XAxisLocation = 'origin';
    ax.YAxisLocation = 'origin';
    box off
end
```

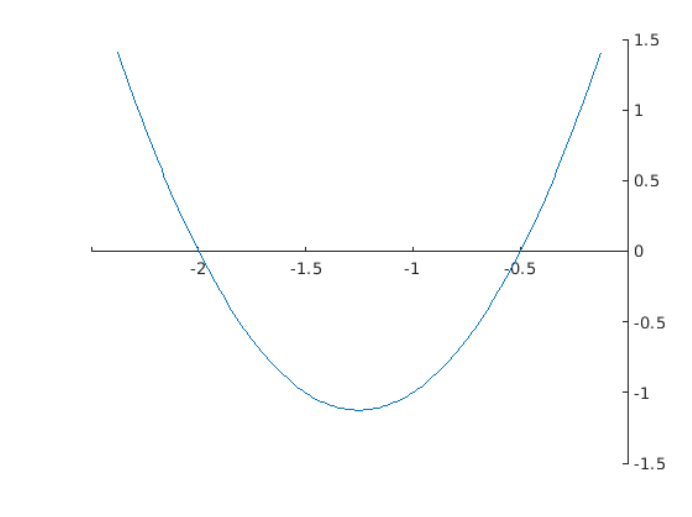
Por último, la *"función principal"* siguiente se encargaría de integrar las tres funciones anteriores: la que pide los datos, la que soluciona la ecuación y la que dibuja la parábola cuando las raíces son reales:

```
function ec2G()
    [a,b,c,error]=pideCoeficientesEc2G();
    if ~error
        raices=resuelveEc2G(a,b,c);
        disp('Las raices de la ecuación son:');
        disp(raices);
        if imag(raices(1))==0
            % Solo se dibuja la parábola si
            % las raices son reales
            dibujaParabola([a,b,c],raices);
        end
    end
end
```

```
>> ec2G()
```

Cuando ejecutamos la función, nos pide los valores de los coeficientes y a continuación nos muestra la solución (o un mensaje de error) y la gráfica en su caso:

```
==========================================
programa para resolver ecuaciones de 2º grado
==========================================

Introduzca coeficientes de la ecuación

Valor del coeficiente a: 2
Valor del coeficiente b: 5
Valor del coeficiente c: 2

ans =

   -2.0000
   -0.5000
```



Identificadores de funciones (*function handles*)

En matlab un *identificador de función* es un tipo de dato cuyo nombre se asocia de manera unívoca con una función. En la práctica ello permitirá usar funciones en algunos contextos en los que normalmente se usan variables (por ejemplo, cuando el argumento de una función sea otra función, o cuando se quiere crear una función sencilla dentro de un script).

A modo de ejemplo, supongamos que hemos creado la función `cuadrado`, encargada simplemente de

elevar al cuadrado un valor o todos los valores de un vector:

```
function x2 = cuadrado(x)
    x2=x.^2;
end
```

Crear un identificador de función consiste en asignarle a esta función un nombre de variable; para ello el nombre de la función debe ir precedido por el símbolo `@` ; el siguiente ejemplo asigna el identificador `f` a la función `cuadrado`.

```
>> addpath programas;
 f=@cuadrado;
```

A partir de ahora la variable `f` se comportará como si fuera la función original:

```
>> addpath programas
 f = @cuadrado;
 a = 2;
 b = f(a)
```

```
b =  4
```

Más información aquí

Usar una función como argumento de otra función

El uso más habitual de los identificadores de función es poder pasar una función como argumento de otra función. Así, por ejemplo, si queremos calcular:

$$\int_1^3 x^2 dx$$

podemos utilizar la función `integral`. Para ello esta función debe recibir como argumentos la función `cuadrado` (x^2) y los límites de integración 1 y 3. Si tratamos de hacer directamente la integral de nuestra función obtenemos un error:

```
>> integral(cuadrado,1,3)
```

```
>> Not enough input arguments.
```

```
Error in cuadrado (line 2)
    x2=x.^2;
```

Sin embargo, si utilizamos el identificador de función `f` que hemos creado más arriba, matlab realiza el cálculo sin problemas:

```
>> integral(f,1,3)
```

```
>> ans =

    8.6667
```

De manera equivalente, también podíamos haber definido directamente el identificador de función al calcular la integral:

```
>> integral(@cuadrado,1,3)
```

```
>> ans =

    8.6667
```

Funciones anónimas

Una *función anónima* es una función cuyo código consiste en una única expresión matlab escrita en una sola linea. Normalmente son funciones simples que no requieren ser guardadas en un archivo .m. Su sintaxis es de la forma:

```
h = @(argumentos) código de la función
```

Por ejemplo, queremos utilizar la función:

$$g(x) = 2x^2e^{-x}$$

y no queremos crear un archivo `.m` que la contenga. Bastaría entonces con definir:

```
>> g = @(x) 2*(x.^2)*exp(-x);
```

y ya podríamos utilizarla en nuestros cálculos:

```
>> g(1)
```

```
ans =  0.73576
```

```
>> g(2)
```

```
ans =  1.0827
```

```
>> g([1 2 3])
```

```
ans =

   0.73576   1.08268   0.89617
```

Más información sobre funciones anónimas aquí

7. Bucles

Es frecuente que al realizar un algoritmo sea preciso repetir un paso durante un número determinado de veces. Estas repeticiones se conocen como *bucles* y en Matlab se implementan mediante los comandos `for` y `while`, dependiendo de si se conoce o no por anticipado el número de repeticiones del bucle.

Bucle For:

El comando `for` permite repetir el bucle el número de veces que se especifique mediante los términos de un vector. Su sintaxis es de la forma:

```
for variable=vector
    < Hacer cosas >
end
```

Por ejemplo, el siguiente código genera los cuadrados de los números del 1 al 10:

```
cuadrados=[];
for i=1:10
    cuadrados=[cuadrados,i^2];
end
cuadrados
```

cuyo resultado es:

```
cuadrados =

     1     4     9    16    25    36    49    64    81   100
```

NOTA: Hemos usado un bucle `for` en este caso simplemente para ilustrar como funciona dicho bucle. En realidad Matlab/Octave nos permite calcular los cuadrados de los números del 1 al 10 de una manera más sencilla simplemente mediante:

```
>> [1:10].^2
```

```
ans =

    1    4    9   16   25   36   49   64   81  100
```

En este caso estamos haciendo lo que se conoce como *aritmética vectorial*. En lugar de repetir mediante un bucle la misma operación para cada término de un vector, simplemente aplicamos dicha operación al vector completo. No todos los problemas prácticos son vectorizables como en este caso.

El vector de índices sobre los que se ejecuta el `for` no tiene por qué ser necesariamente una sucesión de valores correlativos. Por ejemplo, supongamos que se desea construir una función tal que, dado un valor x, devuelva:

$$f(x) = x + \frac{x^2}{2} + \frac{x^4}{4} + \frac{x^5}{5} + \frac{x^7}{7}$$

Una forma de hacerlo utilizando un bucle `for` sería la siguiente:

```
function y=f(x)
  y=0;
  for k=[1 2 4 5 7]
    y=y+(x^k)/k;
  end %for
end %function
```

```
>> f(3)
```

```
ans =  388.78
```

Aunque, al igual que en el caso anterior existe la posibilidad de utilizar aritmética vectorial para realizar el cálculo; en tal caso la función podría programarse así:

```
function y=g(x)
  k=[1 2 4 5 7];
  y=sum((x.^k)./k);
end %function
```

Vemos que el resultado es el mismo que utilizando el bucle `for`:

```
>> g(3)
```

```
ans =  388.78
```

En general siempre resulta más eficiente desde el punto de vista del tiempo de cómputo utilizar la aritmética vectorial antes que un bucle de programación.

Ejemplo: Sucesión de Fibonacci

La sucesión de Fibonacci es una sucesión clásica en matemáticas que se utiliza en múltiples aplicaciones. Comienza con los números 0 y 1 y a partir de ahí cada término se obtiene como suma de los dos anteriores. A continuación mostramos una función que implementa un bucle `for` para generar los primeros n términos de la sucesión de Fibonacci, con $n \geq 2$:

```
function [F]=Fibonacci(n)
  % Esta función calcula los n primeros términos de la sucesión de Fibonacci
  if n!=floor(n)
    F=[];
    disp('Aviso: n debe ser un  valor entero')
  elseif n<=0
    F=[];
    disp("Aviso: n debe ser mayor o igual que cero");
  elseif n==1
    F=[0];
  elseif n==2
    F=[0 1];
  else
    F=[0 1];
    for k=3:n
      F(k)=F(k-1)+F(k-2);
    end %for
  end %if
end %function
```

Para construir la sucesión de Fibonacci de esta forma (es decir, de manera recurrente, calculando cada término como la suma de los dos anteriores) no es posible utilizar aritmética vectorial.

Bucle While:

El comando `while` hace que se repita un bucle mientras se cumpla una condición; el bucle se detiene cuando dicha condición deja de cumplirse. Su sintaxis es de la forma:

```
while condicion
    < Hacer cosas >
end
```

Usaremos un bucle `while` en lugar de un bucle for cuando no sea posible determinar **a priori** el número de veces que hay que iterar el bucle.

Ejemplo: Supongamos que dado un valor x_{max} se desea construir una función que determine el mayor valor de n tal que

$$0^2 + 1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2 \leq x_{max}$$

La función podría programarse usando un bucle while del siguiente modo:

```
function [n]=cuadradosNmax(xmax);
  % Esta función determina el mayor entero n tal que la suma de los
  % cuadrados de los valores enteros de 0 a n es menor o igual que xmax
  if xmax<0
     disp("Aviso: ha especificado una suma máxima negativa");
     n=[];
   else
     n=0;
     suma=0;
     while (suma<xmax)
       n=n+1;
       suma=suma+n^2;
       if (suma>xmax) n=n-1;
      end %while
  end %if
end %function
```

Así, por ejemplo, el mayor valor de n tal que $\sum_{k=1}^{n} k^2 < 100$ sería

```
>> cuadradosNmax(100)
```

```
ans =  6
```

Ejemplo: Construir una función que devuelva todos los términos de la sucesión de Fibonacci menores

que un valor x preespecificado.

La función podría implementarse como:

```
function [F]=FibonacciMenorQue(x)
  % Esta función calcula los términos de la sucesión de Fibonacci
  % menores que x
  if x<=0
    F=[];
    disp("Aviso: x debe ser mayor que cero");
  elseif x<=1
    F=[0];
  else
    F=[0 1];
    k=2;
    while F(k)<x
      k=k+1;
      F(k)=F(k-1)+F(k-2);
    end %while
    F=F(1:(k-1));
  end %if
end %function
```

```
>> FibonacciMenorQue(100)
```

```
ans =

    0    1    1    2    3    5    8   13   21   34   55   89
```

Otra implementación un poco más elegante sería la siguiente:

```
function [F]=FibonacciMenorQueB(x)
  % Esta función calcula los términos de la sucesión de Fibonacci
  % menores que x utilizando un bucle while
  if x<=0
    F=[];
    disp("Aviso: x debe ser mayor que cero");
  else
    F=[0];
    k=1;
    siguiente=1;
    while siguiente<x
      k=k+1;
      F(k)=siguiente;
      siguiente=F(k)+F(k-1);
    end %while
  end %if
end %function
```

```
>> FibonacciMenorQueB(100)
```

```
ans =

    0    1    1    2    3    5    8   13   21   34   55   89
```

8. Gráficos

Gráficos simples: la función `plot`

En Matlab/Octave es muy sencillo generar gráficos de funciones. Supongamos que queremos dibujar la función:

$$f(x) = 4x^3 + 10x^2 + 6$$

en el intervalo $[-3, 1]$. Para ello comenzamos por definir la función; podemos hacerlo creando el archivo `f.m` con el código de la función:

```
function y=f(x)
  y=4*x.^3+10*x.^2+6;
end
```

o de manera equivalente, teniendo en cuenta que en este caso la función es sencilla y se puede definir en una linea, podemos definirla directamente como una función anónima:

```
>> f = @(x) 4*x.^3+10*x.^2+6;
```

Ahora construimos una malla sobre el intervalo $[-3, 1]$. Esto significa que vamos a definir una colección de puntos equiespaciados en este intervalo; si entre punto y punto dejamos una distancia de 0.1 unidades, el vector de puntos de la malla se obtiene mediante:

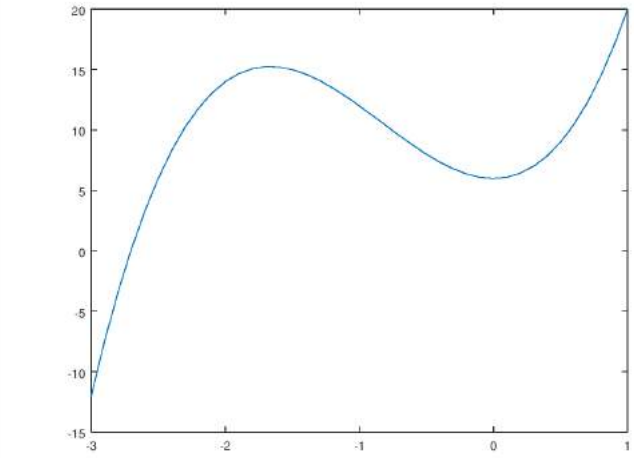
```
>> x=[-3 : 0.1 : 1];
```

De manera alternativa podemos usar la función `linspace(a,b,n)` que crea una malla de `n` puntos equiespaciados entre `a` y `b`. Por ejemplo, si queremos crear una malla de 50 puntos entre -3 y 1 ejecutaríamos el siguiente comando:

```
>> x=linspace(-3,1,50);
```

Por último utilizamos la función `plot()` indicando cuáles son los valores `x` y `f(x)` que describen el recorrido de la función:

```
>> plot(x,f(x))
```



Por cierto, que Matlab/Octave cuentan con una función para definir polinomios. La función que hemos dibujado en el ejemplo anterior, $f(x) = 4x^3 + 10x^2 + 6$ es un polinomio de tercer grado cuyos coeficientes, de mayor a menor grado son [4, 10, 0, 6]. La función `polyval(coefs,x)` calcula los valores de un polinomio con coeficientes `coefs` sobre una malla `x`. Por tanto, obtenemos el mismo resultado de antes si utilizamos la sintaxis:

```
>> f=polyval([4, 10, 0, 6],x);
 plot(x, f)
```

y así nos ahorramos tener que definir la función `f(x)`.

Como modificar el estilo de un gráfico.

En Matlab/Octave existen múltiples opciones para modificar el estilo de un gráfico. En general estas opciones se especifican añadiendo dentro de la llamada a la función `plot()` las propiedades del gráfico que se quieren modificar y a continuación de cada propiedad el valor que se quiere dar a la misma. A continuación se listan las propiedades que se pueden modificar y sus posibles valores:

- **Color:** color de linea. Posibles valores: “red”, “green”, “blue”, “yellow”, “magenta”, “cyan”, “white”, “black”. Los colores se pueden acortar a su primera letra (“r”,“g”,“b”,“y”,“m”,“c”,“w”,“b”). También se puede especificar un color distinto mediante un vector [r g b] donde r, g y b son valores reales entre 0 y 1 que especifican, respectivamente, las proporciones de rojo, verde y azul en la construcción del nuevo color.

- **LineStyle:** estilo de línea. Posibles valores: “-” (linea continua), “–” (linea discontinua), “:” (linea de puntos), “-.” (linea discontinua de guiones y puntos), “none” (sin linea)

- **LineWidth:** Ancho de linea

- **Marker:** Símbolo para el punto. Posibles valores: “none” (ningún punto), “o”, “+”,"*“,”.“,”x“,”s" (square, cuadrado), “d” (diamond, rombo), “^”, “v”, “<”, “>”, “p” (estrella de 5 puntas), “h” (estrella de 6 puntas).

- **MarkerEdgeColor:** Color del borde del punto. Posibles valores: los mismos que para el color de linea.

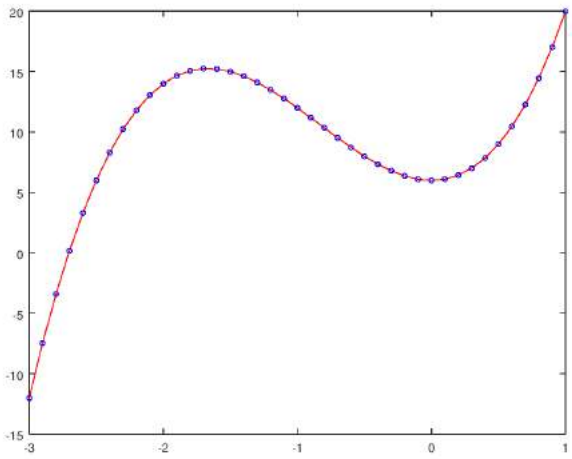
- **MarkerFaceColor:** Color del interior del punto.

- **MarkerSize:** Tamaño del punto. Por ejemplo, si en el gráfico anterior se desea que solamente se dibujen los puntos de la función correspondientes a los valores x de la malla, que dichos puntos sean círculos, que su tamaño sea 4 y que su color sea rojo:

Ejemplo:

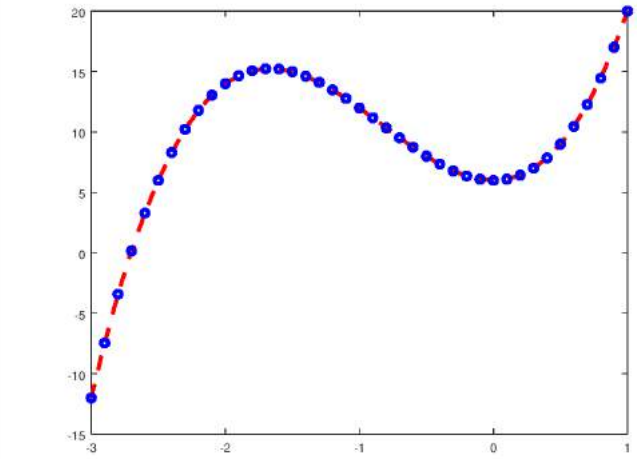
El siguiente código dibuja la misma función $f(x)$ anterior utilizando ahora un círculo para los puntos de la malla, que se dibujan a tamaño 4 y de color azul, y se unen con una linea de color rojo. Obsérvese que la sintaxis comienza con el plot seguida de la sucesión de propiedades y valores; éstas se pueden poner en cualquier orden, si bien detrás de cada propiedad debe ir su valor:

```
plot(x, f, "marker", "o", "markerEdgeColor", "b", ...
     "markersize", 4, "color","red")
```



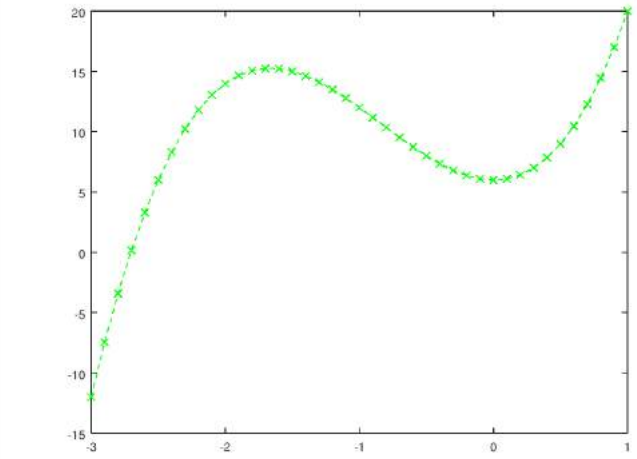
Si se quieren unir los puntos con lineas discontinuas de anchura 3, haciendo los puntos más grandes (tamaño 6), la sintaxis a seguir sería:

```
plot(x, f, "marker", "o", "markerEdgeColor", "b", ...
     "markersize", 6, "linewidth", 3, "linestyle", ...
     "--", "color","red")
```



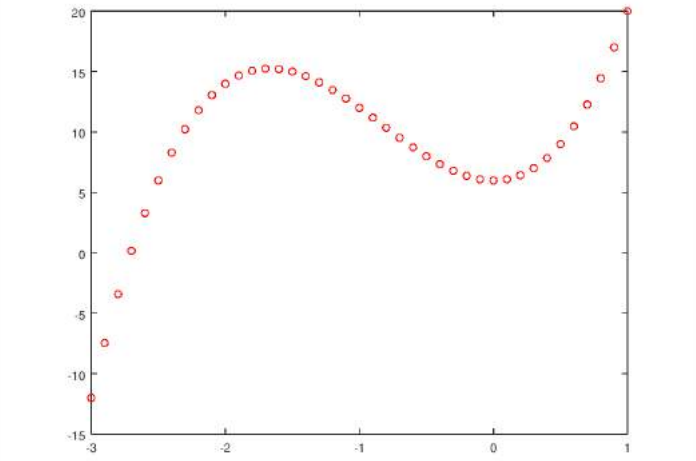
Estilo de linea y puntos resumido: se puede especificar el tipo de punto, el tipo de linea y su color mediante una cadena de caracteres en la que se concatenen los identificadores de los valores elegidos para estas tres propiedades; los identificadores pueden ir en cualquier orden y no necesariamente tienen que estar los tres. Por ejemplo, la siguiente sintaxis especifica que los puntos se representan con el símbolo x, se unen con una linea discontinua y se dibuja todo de color verde:

```
>> plot(x, f, "x--g")
```



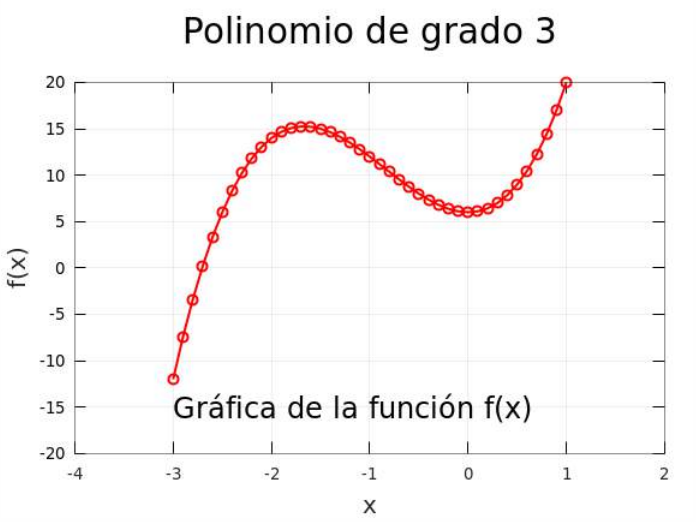
Si especificamos solo el tipo de punto y el color, no se dibuja la linea:

```
>> plot(x, f, "or")
```



Ejes, títulos, etc: También podemos modificar los ejes (etiquetas, tamaño de letra, grosor de la linea de los ejes, límites de los ejes …) y poner un título al gráfico. Para ello primero se crea el gráfico tal como acabamos de hacer y luego se fijan las propiedades de los ejes mediante la función `set(gca, ...)` (la palabra `gca` es un acrónimo de *get current axis*). Además podemos añadir texto en los ejes o en cualquier lugar del gráfico mediante la función `text`. Veamos a modo de ejemplo el resultado de la siguiente sintaxis:

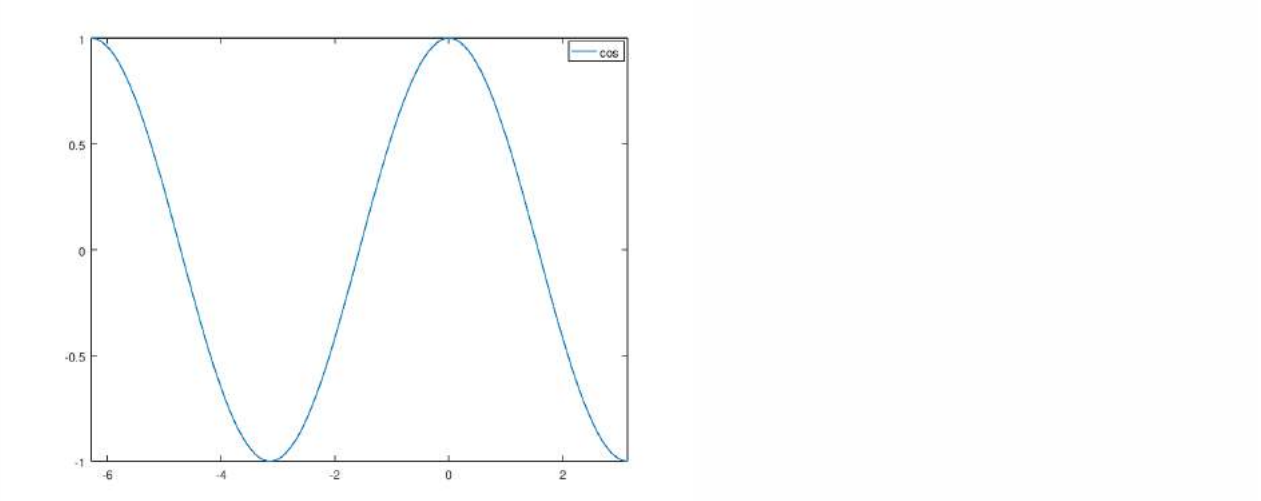
```
plot(x, f, "o-", "markersize", 4, "linewidth", 2, ...
     "color","red")
set(gca, "xlabel", ...
    text("string", "x", "fontsize", 15), ...
    "xlim",[-4,2], "fontsize",10)
set(gca, "ylabel", ...
    text("string", "f(x)", "fontsize", 15), ...
    "ylim",[-20,20], "fontsize", 10, "ytick",[-20:5:20])
set(gca, "title", ...
    text("string","Polinomio de grado 3","fontsize", 22))
text(-3,-15,"Gráfica de la función f(x)", "fontsize", 18)
grid on
```



La función `fplot()`

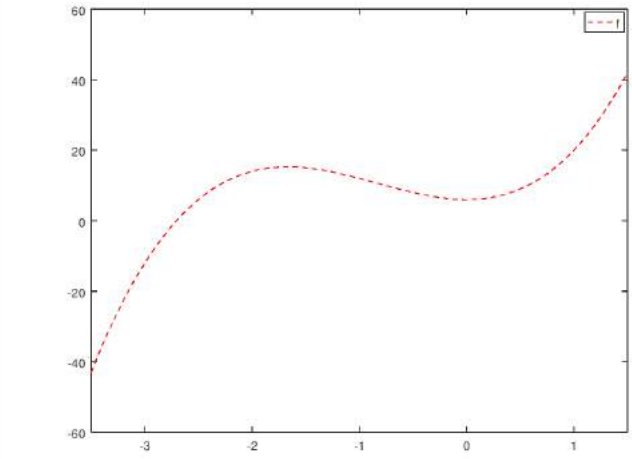
Esta función permite dibujar funciones definidas por el usuario (o ya existentes) de manera muy simple, indicando el recorrido de la función y el número de puntos que se quieren emplear para hacer la gráfica. Por ejemplo, para dibujar la función coseno (`cos`) entre -2π y pi, utilizando 50 puntos, podemos utilizar la sintaxis:

```
>> fplot("cos", [-2*pi, pi], 50)
```



También podemos dibujar el polinomio que ya definimos más arriba, utilizando la función *"f"* que hemos creado:

```
>> fplot("f", [-3.5, 1.5], 50, "--r")
```



Superposición de varias funciones en un mismo gráfico.

Supongamos que queremos superponer en un mismo gráfico la representación de las siguientes funciones entre $-\pi$ y 2π:

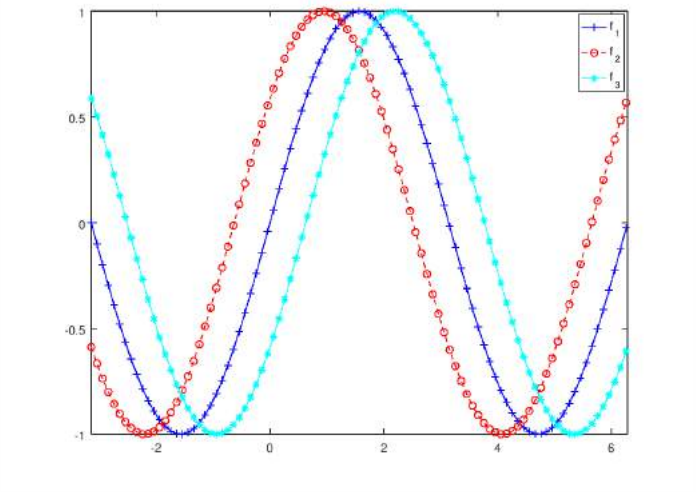
$$f_1(x) = sin(x) \qquad f_2(x) = sin(x + \frac{\pi}{5}) \qquad f_3(x) = sin(x - \frac{\pi}{5})$$

En primer lugar debemos definir las funciones $f_2(x)$ y $f_3(x)$:

```
>> f2 = @(x) sin(x+pi/5);
 f3 = @(x) sin(x-pi/5);
```

y ahora podemos representarlas con una única llamada a la función `plot`. Nótese que al final añadimos el comando `legend` para indicar que coloque en la esquina superior derecha (noreste) una leyenda especificando qué gráfica corresponde a cada función:

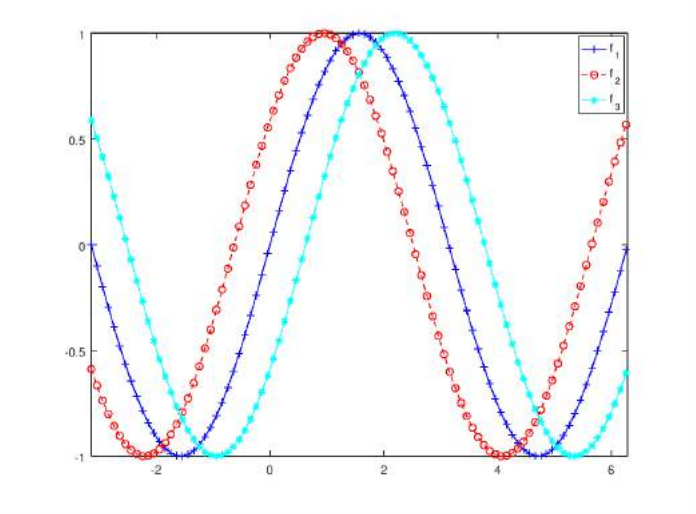
```
x=[-pi:0.1:2*pi];
plot(x,sin(x),"+-b",x,f2(x),"o--r",x,f3(x),"*-.c");
set(gca, "xlim",[-pi,2*pi]);
legend ({"f_1", "f_2", "f_3"}, "location", "northeast");
```



Nótese que a continuación de cada par `x, f(x)` se incluyen las propiedades de esa función; en este caso hemos usado la versión abreviada del estilo de linea, pero podríamos usar la versión completa e ir especificando propiedades (markerSize, markerEdgeColor, lineWidth, …) distintas para cada gráfica.

Otra manera de superponer dos o más funciones en un mismo gráfico consiste en dibujar la primera función seguida del comando `hold on`. A continuación se escribe el código para el resto de las funciones y se termina con `hold off`. El siguiente ejemplo aclara como se realiza el proceso:

```
plot(x,sin(x),"+-b")
hold on
plot(x,f2(x),"o--r")
plot(x,f3(x),"*-.c")
hold off
set(gca, "xlim",[-pi,2*pi])
legend ({"f_1", "f_2", "f_3"}, "location", "northeast");
```



Pueden combinarse también varias gráficas en la misma figura utilizando `fplot()` en lugar de `plot()`.

Borrado de un gráfico

La función `clf` borra el gráfico actual, dejando la ventana de gráficos en blanco.

Guardar gráficos

Una vez que hemos generado un gráfico, podemos guardarlo en un archivo png, jpg, pdf o eps (postscript) entre otros formatos, mediante la sintaxis:

```
>> print("miGrafico.png", "-dpng");
 print("miGrafico.jpg", "-djpg");
 print("miGrafico.pdf", "-dpdf");
 print("miGrafico.eps", "-deps");
```

Tecleando `help print` en la consola veremos más opciones para guardar (o imprimir) gráficos.

Gráficos 3D

Matlab/Octave ofrecen también la posibilidad de dibujar superficies 3D. Supongamos que queremos dibujar la función:

$$f(x,y) = x^2 - y^2, \quad x \in [-2,2],\ y \in [-2,2]$$

Comenzamos definiendo el dominio de la función:

```
>> x = [-2:0.1:2];
 y = x;
```

Ahora generamos la malla (x, y) sobre la que se va a dibujar la superficie $f(x, y)$:

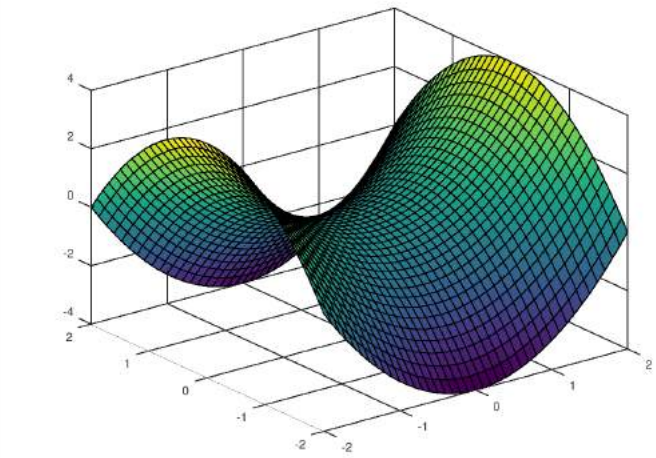
```
>> [X Y] = meshgrid(x,y);
```

Ahora calculamos los valores $z = f(x, y) = x^2 - y^2$ para todos los puntos de esta malla:

```
>> Z = X.^2 - Y.^2;
```

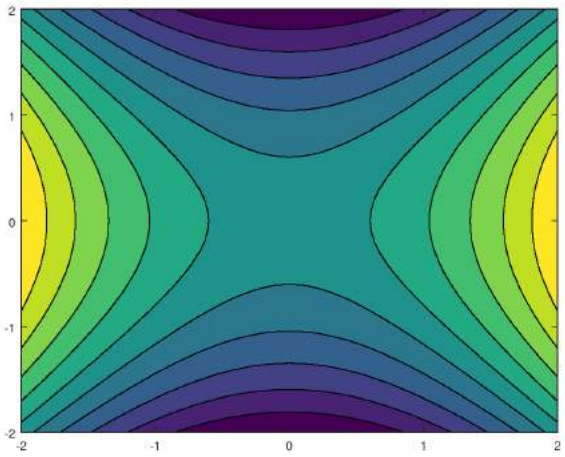
Por último usamos la función `surf()` para dibujar la superficie $f(x, y)$:

```
>> surf(X, Y, Z)
```



La función `contourf()` traza las curvas de nivel correspondientes a la figura anterior:

```
>> contourf(X, Y, Z)
```



Se puede consultar la ayuda de Matlab (o cualquiera de los numerosos tutoriales que hay en linea) para ver las opciones existentes para modificar los gráficos 3D: modificaciones en los ejes, colores, orientación, punto de vista, …

9. Manejo de datos en Matlab: Estructuras, Matrices de celdas y Tablas

Hasta ahora hemos visto que matlab permite manipular con facilidad vectores y matrices. Ambos tipos de variables se caracterizan porque sus elementos son siempre de la misma clase: son todos valores numéricos o son todos valores de tipo texto (string o char). No es posible en una misma matriz combinar números y texto. Para ello en matlab existen tipos de variables más sofisticados (*estructuras* y *celdas*) cuya finalidad es precisamente almacenar datos de distintas clases.

Estructuras (`Struct`)

Una *estructura* en matlab es una variable que agrupa varios *contenedores de datos* llamados campos. Cada campo puede contener cualquier tipo de datos (números, cadenas de texto, vectores, matrices …), y puede tener cualquier dimensión. Por ejemplo, si la información que se recoge para cada alumno de la universidad está formada por su nombre, apellidos, dni, año de nacimiento y cursos en que está matriculado, dicha información podría almacenarse

en una estructura que llamaremos `alumno` definida del siguiente modo:

⊕

```
>> alumno.nombre="María";
 alumno.apellido="Rodríguez";
 alumno.DNI=12345678;
 alumno.anioNac=2000;
 alumno.cursosMatriculado=[1 2 3];
 alumno
```

```
alumno =

  scalar structure containing the fields:

    nombre = María
    apellido = Rodríguez
    DNI =  12345678
    anioNac =  2000
    cursosMatriculado =

       1   2   3
```

Obsérvese que los campos de la estructura `alumno` se especifican mediante su nombre separado por un punto del nombre de la estructura. En la misma estructura hay variables de tipo *string* (nombre y apellidos), variables numéricas (DNI y año de nacimiento) y vectores (cursos en que está matriculado)

En el ejemplo, hemos creado la estructura definiendo en cada linea el contenido de un campo. De manera equivalente podemos definir la estructura en una sola línea utilizando la función `struct`:

```
>> alumno=struct("nombre", "María","apellido","Rodríguez","DNI",12345678,"anionac",2000,"
 alumno
```

```
alumno =

  scalar structure containing the fields:

    nombre = María
    apellido = Rodríguez
    DNI =  12345678
    anionac =  2000
    cursosMatriculado =

       1   2   3
```

Nótese que para utilizar esta sintaxis los nombres de los campos deben ir entrecomillados; a continuación de cada campo se introduce su valor, que sólo se entrecomilla si es de tipo carácter.

Es posible definir *vectores de estructuras* (*struct array*); así, por ejemplo, si quisiéramos guardar la lista de alumnos de la clase podríamos seguir añadiendo términos a la estructura anterior como si de un vector se tratase:

```
>> alumno(2)=struct("nombre","Pedro","apellido","González","DNI",87654321,"anionac",1999,
 alumno
```

```
alumno =

  1x2 struct array containing the fields:

    nombre
    apellido
    DNI
    anionac
    cursosMatriculado
```

```
>> alumno.nombre
```

```
ans = María
ans = Pedro
```

```
>> alumno.apellido
```

```
ans = Rodríguez
ans = González
```

Existe la posibilidad de convertir todos los valores del mismo campo en un vector; por ejemplo, si queremos extraer todos los nombres de los alumnos a un único vector empleamos la sintaxis:

```
>> [alumno.nombre]
```

```
ans =

  1×2 string array

    "María"    "Pedro"
```

Para acceder a los datos de algún campo particular de un alumno concreto utilizamos la siguiente sintaxis:

```
>> alumno(1).apellido
```

```
ans = Rodríguez
```

```
>> alumno(2).DNI
```

```
ans =  87654321
```

Ver más información sobre estructuras aquí

Vectores y matrices de celdas (`cells arrays`)

Al igual que las estructuras, las matrices de celdas son variables formadas por varios *contenedores de datos* (campos). A diferencia de las estructuras, cuyos campos tienen un nombre asignado (especificado a continuación del nombre de la estructura separado por un punto), en el caso de las celdas los campos que las forman no tienen nombre sino que se indexan numéricamente del mismo modo que los elementos de una matriz.

Creación de una matriz de celdas

Por ejemplo, supongamos que queremos crear una tabla en la que figuren los nombres de las montañas más altas del mundo junto a su altura:

Las cuatro montañas más altas del planeta

nombre	altura
Everest	8848
K2	8611
Kanchenjunga	8586
Llhotse	8516

Para crear una matriz de celdas con esta información, usamos la misma sintaxis que para las matrices, con la única diferencia de que deberemos usar llaves { } en lugar de corchetes [ ]:

⊕

```
>> mountains={"Everest" "K2" "Kanchenjunga" "Llhotse";
   8848 8611 8586 8516 }
```

```
mountains =

  2×4 cell array

  Columns 1 through 4

    {["Everest"]}    {["K2"]}    {["Kanchenjunga"]}    {["Llhotse"]}
    {[     8848]}    {[8611]}    {[           8586]}    {[     8516]}
```

Las matrices de celdas, igual que las matrices numéricas, se pueden trasponer:

```
>> mountains'
```

```
ans =

  4×2 cell array

    {["Everest"     ]}    {[8848]}
    {["K2"          ]}    {[8611]}
    {["Kanchenjunga"]}    {[8586]}
    {["Llhotse"     ]}    {[8516]}
```

Dado que las matrices de celdas se indexan igual que las matrices numéricas, podemos extraer sus términos indicando índice de fila y columna, si bien en el caso de las matrices de celdas estos índices deben especificarse entre llaves:

```
>> mountains{1,2}
```

```
ans = K2
```

Si especificáramos el índice entre paréntesis, en lugar de obtener el **contenido** de la celda, obtendríamos **una nueva celda** que contiene ese valor:

```
>> mountains(1,2)
```

```
ans =

  1×1 cell array

    {["K2"]}
```

Podemos seleccionar una o varias filas o columnas pero, y **esto es importante**, si queremos que el resultado de la selección sea una nueva matriz de celdas, debemos especificar los índices entre

paréntesis. Si no, lo que se seleccionan son los valores de las celdas y se pierde la estructura de celda:

Obsérvese la diferencia entre usar paréntesis en la selección (se obtiene una matriz de celdas como resultado):

```
>> mountains(1,:)
```

```
ans =

  1×4 cell array

  Columns 1 through 4

    {["Everest"]}    {["K2"]}    {["Kanchenjunga"]}    {["Llhotse"]}
```

y usar llaves (se obtienen los valores que estaban dentro de las celdas)

```
>> mountains{1,:}
```

```
ans =
    "Everest"

ans =
    "K2"

ans =
    "Kanchenjunga"

ans =
    "Llhotse"
```

Añadir nuevos términos a una matriz de celdas

Si queremos añadir, por ejemplo, el año de la primera ascensión a cada una de estas montañas (dichos años fueron respectivamente 1953, 1954, 1955 y 1956) lo haríamos del siguiente modo:

```
>> mountains(3,:)={1953, 1954, 1955, 1956}
```

```
mountains =

  3×4 cell array

    {["Everest"]}    {["K2"]}    {["Kanchenjunga"]}    {["Llhotse"]}
    {[     8848]}    {[8611]}    {[          8586]}    {[     8516]}
    {[     1953]}    {[1954]}    {[          1955]}    {[     1956]}
```

Conversión de matrices de celdas en matrices o vectores numéricos o de texto.

Muchas veces resultará conveniente convertir una matriz de celdas (o una parte de la misma) a vector o matriz (numéricos o de texto). Por ejemplo, si de nuestra matriz de montañas queremos extraer simplemente los nombres (la primera fila), ya hemos visto que `mountains{1,:}` nos devuelve los valores en dicha fila. Para guardar estos valores en un vector bastará con poner esta expresión entre corchetes:

```
>> nombres = [mountains{1,:}];
 nombres
```

```
ans =

  1×4 string array

    "Everest"    "K2"    "Kanchenjunga"    "Llhotse"
```

De igual modo podemos extraer las alturas:

```
>> alturas = [mountains{2,:}];
 alturas
```

```
ans =

        8848        8611        8586        8516
```

Para más conversiones, ver la función cell2mat en la web de Mathworks.

Construcción de una matriz de celdas vacía

La función `cell(m,n)` permite crear una matriz de celdas vacía de dimensión $m \times n$. Si solo se especifica `cell(m)` se crearía una matriz de celdas vacía de dimensión $m \times m$.

```
>> A=cell(3)
```

```
A =

  3×3 cell array

    {0×0 double}    {0×0 double}    {0×0 double}
    {0×0 double}    {0×0 double}    {0×0 double}
    {0×0 double}    {0×0 double}    {0×0 double}
```

Una vez definida una matriz de celdas vacía, podemos rellenar sus celdas. En el ejemplo anterior, en la matriz de celdas `mountain` la primera fila eran nombres de montañas y la segunda eran alturas, pero en general en una matriz de celdas, cada celda puede contener cualquier tipo de datos: números, texto, matrices, incluso otras matrices de celdas. Así, en el siguiente ejemplo, la matriz de celdas `A` tiene su primera fila compuesta por números, la segunda por matrices y la tercera por una mezcla de variables de distinto tipo.

```
>> A(1,:)={1 2 3};
 A(2,:)= {zeros(2), ones(3), [1:5]};
 A(3,:)= {"Texto", [1 1; 2 3], ["Matriz"; "de";"texto"]};
 A
```

```
A =

  3×3 cell array

    {[       1]}    {[       2]}    {[       3]}
    {2×2 double}    {3×3 double}    {1×5 double}
    {["Texto" ]}    {2×2 double}    {3×1 string}
```

Ver más información sobre matrices de celdas aquí

Tablas

En matlab una *tabla* es una matriz especial, caracterizada porque sus columnas tienen nombre, y cada columna puede ser de un tipo distinto (string o numérico). Los datos de alturas de montañas que vimos en la sección anterior también podrían codificarse en matlab como una tabla. En esta sección utilizaremos como ejemplo los siguientes datos, referidos al año 2016 en Canarias:

Datos de Canarias, 2016

Isla	Extensión (km2)	Población	Municipios	Capital
Lanzarote	846	145084	7	Arrecife
Fuerteventura	1660	107521	6	Puerto del Rosario
Gran Canaria	1560	845195	21	Las Palmas de GC

Isla	Extensión (km2)	Población	Municipios	Capital
Tenerife	2034	891111	31	Santa Cruz de Tenerife
La Gomera	370	20940	6	San Sebastián
La Palma	708	81486	14	Santa Cruz de La Palma
El Hierro	269	10587	3	Valverde

Para crear una tabla con estos datos, definimos primero los vectores que conforman cada una de las variables (columnas) de la tabla:

```
isla = ["Lanzarote" "Fuerteventura" "Gran Canaria" ...
      "Tenerife" "La Gomera" "La Palma" "El Hierro" ]';
extension = [846 1660 1560 2034 370 708 269]';
poblacion = [145084 107521 845195 891111 20940 81486 ...
             10587]';
municipios = [7 6 21 31 6 14 3]';
capital = ["Arrecife" "Puerto del Rosario" ...
           "Las Palmas de GC" ...
           "Santa Cruz de Tenerife" "San Sebastián" ...
           "Santa Cruz de La Palma" "Valverde"]';
```

NOTA 1: cuando el comando que escribimos en matlab no cabe en una linea y debe continuar en la linea siguiente, hay que utilizar tres puntos suspensivos … para indicarlo.

NOTA 2: Nótese que hemos definido cada vector como vector fila y al final le hemos añadido un apóstrofe para que se guarde como vector columna. **Es importante que todas las columnas de una tabla se definan como vectores columna**.

Una vez definidos estos vectores creamos la tabla usando la función `table()`:

```
>> canarias = table(isla, extension, poblacion, municipios,capital)
```

```
canarias =

  7×5 table

        isla          extension    poblacion     municipios            capital
    ---------------    ---------    ----------    ----------    ------------------------

    "Lanzarote"           846       1.4508e+05         7        "Arrecife"
    "Fuerteventura"      1660       1.0752e+05         6        "Puerto del Rosario"
    "Gran Canaria"       1560        8.452e+05        21        "Las Palmas de GC"
    "Tenerife"           2034       8.9111e+05        31        "Santa Cruz de Tenerife"
    "La Gomera"           370            20940         6        "San Sebastián"
    "La Palma"            708            81486        14        "Santa Cruz de La Palma"
    "El Hierro"           269            10587         3        "Valverde"
```

Igual que en las estructuras, podemos acceder a cada variable simplemente añadiendo su nombre a continuación del nombre de la tabla, separando ambos con un punto:

```
>> canarias.isla
```

```
ans =

  1×7 string array

  Columns 1 through 5

    "Lanzarote"    "Fuerteventura"    "Gran Canaria"    "Tenerife"    "La Gomera"

  Columns 6 through 7

    "La Palma"    "El Hierro"
```

En la práctica, la mayor parte de las veces las tablas se leerán desde archivos de texto externo mediante la función `readtable` (ver la sección 6-Lectura de datos desde archivos csv).

Más información sobre el uso de tablas aquí

10. Lectura de datos desde archivos .csv

Archivos csv

Con frecuencia deberemos "cargar" datos almacenados en archivos externos para utilizarlos en cálculos o programas realizados en Matlab/Octave. En la sección *"Descargas"* de la web de la asignatura se encuentran accesibles una colección de archivos en formato `.csv` con datos de diversas variables medidas durante una campaña oceanográfica. El formato `.csv` (*"comma separated values"*) es un formato de texto estándar para el intercambio de datos. Cada fila de un archivo `.csv` corresponde a los valores de distintas variables; dichos valores se encuentran separados por comas; el símbolo del separador decimal es el punto; y las variables tipo texto se recogen entre comillas (dobles o simples).

Prácticamente todos los programas para el tratamiento de hojas de cálculo (Excel, libreOffice Calc, …) son capaces de leer un archivo `.csv` y mostrarlo como hoja de cálculo. A su vez, si hemos rellenado una hoja de cálculo con datos utilizando uno de estos programas es siempre posible exportar los datos a formato `.csv`.

⊕ Los datos que vamos a utilizar como ejemplo han sido medidos con boyas ARGO. En concreto, los archivos *temperatura.csv* y *salinidad.csv* contienen valores de temperatura y salinidad obtenidos a 100 profundidades distintas, medidos en los 109 puntos de muestreo que se muestran en la figura al margen.

Cada uno de estos archivos contiene 100 filas y 109 columnas. Cada columna corresponde a una posición longitud-latitud y cada fila corresponde a una profundidad. El archivo *depth.csv* contiene también 100 filas y 109 columnas que indican la profundidad a que se ha medido cada uno de los valores de temperatura y salinidad de los otros dos archivos. Los archivos *latitud.csv* y *longitud.csv* contienen, cada uno, 109 filas y una columna. La combinación de cada valor de longitud y latitud (de 1 a 109) especifica la localización de cada punto de muestreo. A su vez, el archivo *time.csv* tiene también 109 filas (un valor por fila) que especifican el momento en que la boya comenzó a tomar datos en cada punto.

Si abrimos, por ejemplo, el archivo *temperatura.csv* con LibreOffice (o Excel) veremos los datos de temperatura dispuestos como una hoja de cálculo:

	A	B	C	D	E	F	G	H	I
1	23.2	23.26	23.32	23.3	23.22	23.21	23.17	23.17	23.16
2	23.2	23.26	23.32	23.3	23.23	23.21	23.17	23.17	23.16
3	23.19	23.24	23.32	23.3	23.24	23.21	23.17	23.18	23.16
4	23.19	23.22	23.3	23.29	23.24	23.21	23.18	23.18	23.16
5	23.19	23.22	23.27	23.29	23.24	23.21	23.18	23.17	23.16
6	23.19	23.21	23.24	23.28	23.24	23.21	23.18	23.18	23.16
7	23.19	23.2	23.22	23.27	23.24	23.21	23.18	23.18	23.16
8	23.19	23.2	23.22	23.26	23.24	23.21	23.19	23.18	23.16
9	23.19	23.2	23.22	23.26	23.24	23.21	23.19	23.18	23.16
10	23.19	23.2	23.21	23.25	23.24	23.22	23.19	23.18	23.16
11	23.19	23.2	23.21	23.25	23.24	23.22	23.19	23.18	23.16
12	23.19	23.2	23.21	23.23	23.24	23.22	23.19	23.18	23.16
13	23.19	23.2	23.21	23.22	23.24	23.21	23.18	23.18	23.16
14	23.19	23.2	23.21	23.21	23.24	23.22	23.19	23.18	23.16
15	23.19	23.19	23.21	23.21	23.24	23.22	23.19	23.18	23.16
16	23.19	23.19	23.21	23.2	23.23	23.22	23.19	23.18	23.16
17	23.19	23.19	23.2	23.2	23.23	23.21	23.2	23.18	23.16
18	23.19	23.19	23.2	23.2	23.22	23.21	23.2	23.19	23.16
19	23.19	23.19	23.2	23.2	23.2	23.22	23.2	23.18	23.17
20	23.19	23.19	23.2	23.2	23.2	23.22	23.2	23.18	23.17
21	23.19	23.19	23.2	23.2	23.19	23.22	23.2	23.18	23.17
22	23.19	23.19	23.2	23.19	23.19	23.22	23.2	23.18	23.17
23	23.2	23.19	23.2	23.19	23.18	23.22	23.19	23.18	23.17
24	23.2	23.19	23.19	23.18	23.18	23.22	23.2	23.18	23.17
25	23.2	23.19	23.19	23.17	23.16	23.2	23.2	23.18	23.17
26	23.2	23.19	23.19	23.17	23.14	23.19	23.2	23.18	23.17
27	23.2	23.19	23.19	23.17	23.13	23.17	23.2	23.18	23.17
28	23.2	23.19	23.19	23.16	23.13	23.17	23.19	23.18	23.17

Si abrimos el mismo archivo con el bloc de notas veremos que sus valores están simplemente separados por comas (el símbolo decimal es el punto):

Nótese que en algunos lugares aparece escrito `NaN`. Esta expresión es un acrónimo de *"Not a Number"* indicando que en ese punto no se pudo medir la temperatura, y por tanto falta el valor numérico correspondiente. En tales casos es equivalente escribir `NaN` en el archivo, o dejar la casilla en blanco. Escribir otra cosa daría lugar a errores de lectura del archivo.

Lectura de archivos csv con `csvread`

Para leer estos archivos en Matlab en primer lugar hemos de colocarnos en la carpeta que los contiene. Supongamos que los archivos anteriores están en una carpeta llamada `datos`, que a su vez está dentro de la carpeta `c:\users\Mis documentos`. Nos colocamos en primer lugar en esta carpeta mediante:

⊕

```
>> cd "c:\users\Mis documentos"
```

(o bien nos situamos en ella utilizando el navegador que aparece a la izquierda en la pestaña "HOME" de la aplicación de matlab)

Ahor, para leer los datos contenidos en los archivos utilizamos la función `csvread()`, asignando el contenido de cada archivo a una variable:

⊕

```
>> T=csvread("datos/temperatura.csv");
 S=csvread("datos/salinidad.csv");
```

```
lon=csvread("datos/longitud.csv");
lat=csvread("datos/latitud.csv");
z=csvread("datos/depth.csv");
```

De esta forma hemos creado las matrices T, S, lon, lat, y Z, que contienen los datos que se han leido desde los archivos csv originales. Podemos ver, por ejemplo, la dimensión (el tamaño) de la matriz T donde se han cargado los datos de temperatura:

```
>> size(T)
```

```
ans =

   100   109
```

Como señalábamos más arriba, cada una de las 109 columnas de T corresponde a una localización de la boya en un día concreto, y cada fila corresponde a una profundidad. Así, por ejemplo, la columna 15 de la matriz T correspondería a las temperaturas medidas por la boya en la longitud:

```
>> lon(15)
```

```
ans = -38.484
```

y en la latitud:

```
>> lat(15)
```

```
ans =  25.154
```

Podemos combinar en una única matriz los datos de profundidad (`z`) con los de temperatura (`T`) en esa posición:

```
>> zt=[z(:,15),T(:,15)]
```

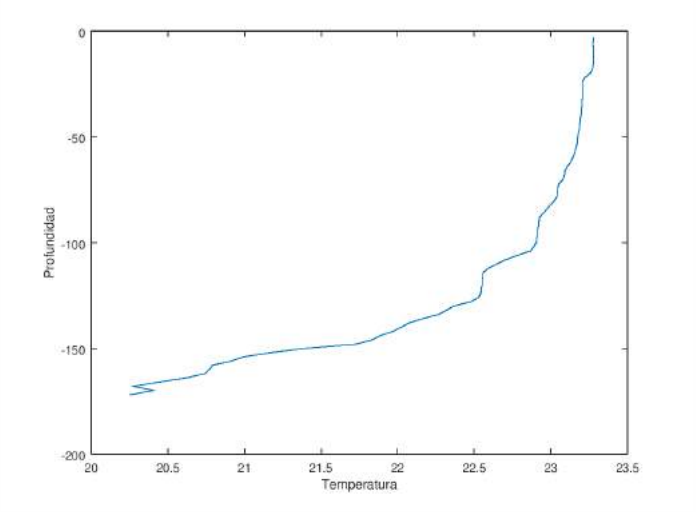
```
zt =

    2.9000   23.2790
    3.9000   23.2770
    5.9000   23.2750
    7.9000   23.2770
    9.9000   23.2780
   11.9000   23.2780
   13.9000   23.2780
   15.9000   23.2760
   17.9000   23.2710
   19.9000   23.2560
```

```
 21.9000    23.2230
 23.9000    23.2060
 25.9000    23.2070
 28.0000    23.2070
 29.9000    23.2060
 31.9000    23.2050
 33.9000    23.2030
 35.9000    23.2010
 37.9000    23.1990
 39.9000    23.1960
 41.9000    23.1910
 43.9000    23.1890
 45.9000    23.1890
 47.9000    23.1800
 49.9000    23.1750
 51.9000    23.1720
 53.9000    23.1700
 55.9000    23.1610
 57.9000    23.1540
 59.9000    23.1420
 61.9000    23.1290
 63.9000    23.1080
 65.9000    23.0910
 67.9000    23.0880
 69.9000    23.0800
 71.9000    23.0550
 73.9000    23.0460
 75.9000    23.0430
 77.9000    23.0410
 79.9000    23.0250
 81.9000    23.0000
 83.9000    22.9760
 85.9000    22.9530
 87.9000    22.9260
 89.9000    22.9220
 91.9000    22.9210
 93.9000    22.9110
 95.9000    22.9080
 97.9000    22.9070
 99.9000    22.9040
101.9000    22.8890
103.9000    22.8670
105.9000    22.7850
107.9000    22.7120
109.9000    22.6510
111.9000    22.5950
113.9000    22.5590
115.9000    22.5540
117.9000    22.5530
119.9000    22.5510
121.9000    22.5430
123.9000    22.5420
125.9000    22.5290
127.9000    22.4780
129.9000    22.3700
```

```
  131.9000    22.3190
  133.9000    22.2660
  135.9000    22.1650
  137.9000    22.0760
  139.9000    22.0260
  141.9000    21.9720
  143.9000    21.8850
  146.0000    21.8320
  148.0000    21.7270
  149.9000    21.4130
  151.9000    21.1790
  153.9000    20.9950
  155.9000    20.9170
  157.9000    20.7930
  159.9000    20.7730
  161.9000    20.7440
  163.9000    20.6220
  165.9000    20.4420
  167.9000    20.2770
  169.9000    20.4070
  172.0000    20.2500
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
       NaN        NaN
```

`z` es la profundidad en metros y `T` es la temperatura (en ºC) medida a esa profundidad; como podemos apreciar, a partir de 172 metros no hay datos disponibles. En matlab es sencillo hacer una representación gráfica de estos datos. El código siguiente representa la temperatura en el eje X y la profundidad en el eje Y (nótese que cambiamos el signo del vector `z`, de forma que la profundidad se mida “hacia abajo”):

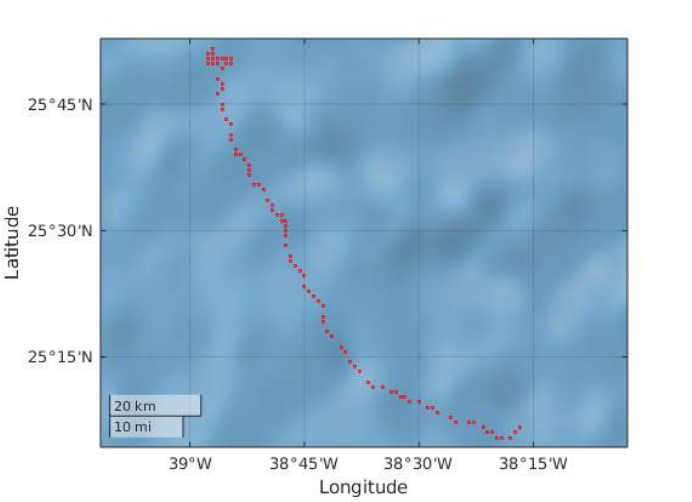
```
>> plot(T(:,15),-z(:,15))
 xlabel("Temperatura")
 ylabel("Profundidad")
```



En esta gráfica podemos apreciar claramente que a medida que aumenta la profundidad disminuye la temperatura.

Matlab ofrece también la posibilidad de dibujar mapas. Para ello es preciso descargar el *Mapping Toolbox* (paquete de aplicaciones para representación geográfica) y al menos un mapa base. La descarga de ambos componentes puede hacerse directamente desde la aplicación matlab (al menos a partir de la versión R2017b): dentro de la pestaña “HOME” pinchamos en Add-Ons -> Get Add-Ons y en el buscador de la ventana que aparece escribimos “basemap”. Para que funcione tenemos que estar conectados a internet, en cuyo caso el servidor de Mathworks nos presentará diversas aplicaciones y complementos, entre los que se encuentran el *Mapping Toolbox* y los mapas base que necesitamos. Basta pinchar en el nombre de la aplicación o mapa que queramos descargar para que matlab nos ofrezca la opción de descargar e instalar. Matlab tiene cinco mapas base (*landcover, colorterrain, grayterrain, grayland* y *bluegreen*). Podemos descargarlos los cinco (tarda bastante) o uno solo de ellos. El mapa que veremos a continuación usa como mapa base el *landcover*. Una vez descargados e instalados el *Mapping Toolbox* y el mapa base, la siguiente sintaxis dibuja los 109 puntos de muestreo *(lat,lon)* en los que la boya ha medido la temperatura y la salinidad a distintas profundidades:

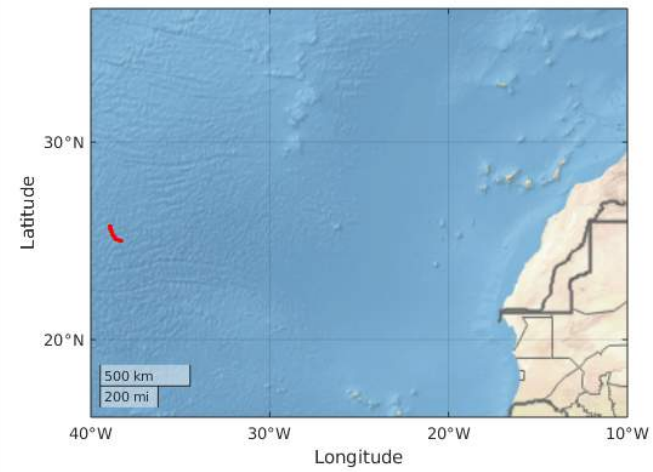
```
>> geoscatter(lat, lon, 3, 'red')
 geobasemap('landcover')
```



En la función `geoscatter` hemos especificado los vectores `lat` y `lon` con las latitudes y longitudes respectivas de los puntos de muestreo. El valor 3 es simplemente el tamaño de cada punto, y `'red'` es el color que usamos para la representación. Por defecto matlab centra el mapa en la región en la que se observan los puntos.

Si queremos ampliar el mapa anterior para situar la región de muestreo y poder hacernos una mejor idea de donde se encuentra, podemos cambiar los límites de latitud y longitud con los que se ha trazado el mapa, de forma que se incluyan las islas Canarias y parte de la costa africana. Para ello basta ejecutar la linea siguiente a continuación de las anteriores.

```
>> geolimits([25,28],[-40,-10])
```



Observamos, pues, que los puntos de muestreo se encuentran al este y ligeramente al sur de las islas Canarias.

Tratamiento de los valores perdidos

Como señalábamos más arriba, en los archivos de datos de temperatura y salinidad faltan valores que han sido sustituidos por el código de valor perdido `NaN`. Estos valores pueden no haberse medido por fallos del sensor, por fallos de la batería de la boya o por cualquier otra causa. La presencia de valores perdidos en un conjunto de datos generalmente es causa de problemas. Si quisiéramos, por ejemplo, calcular la temperatura media en la columna de agua correspondiente al punto de muestreo número 15, podríamos utilizar la función `mean`:

```
>> mean(T(:,15))
```

```
ans = NaN
```

pero, cotras omo vemos, la función nos devuelve `NaN` (*Not a Number*). Ello se debe a que la presencia de valores perdidos en el conjunto de datos impide calcular la media directamente (como hay valores que no se conocen, matlab no puede calcular la media usando dichos valores). Esto ocurre con muchas funciones. Por ejemplo, si quisiéramos calcular la media sumando los valores y dividiendo por el número total de valores válidos (no perdidos) y aplicamos para ello la función `sum`, obtenemos el mismo resultado que antes:

```
>> sum(T(:,15))
```

```
ans = NaN
```

Para poder calcular la media (o la suma) en presencia de valores perdidos, hay que utilizar funciones que ignoren dichos valores y calculen la media (o la suma) con los valores no perdidos. Las funciones `nanmean` y `nansum` hacen exactamente eso:

```
>> nanmean(T(:,15))
```

```
ans =

   22.5908
```

```
>> nansum(T(:,15))
```

```
ans =

   1.9428e+03
```

Si quisiéramos saber cuántos valores perdidos hay en el vector `T(:,15)` podemos utilizar la función `isnan()`. Esta función vale 1 para los valores perdidos y 0 en otro caso. Si, por ejemplo consideramos el vector:

```
>> v=[1 2 3 NaN 4 5 NaN];
```

al aplicar la función `isnan()` obtenemos:

```
>> isnan(v)
```

```
ans =

  0  0  0  1  0  0  1
```

es decir, un 1 donde había `NaN`s y un 0 donde no había `NaN`s. Por tanto, si queremos contar el número de valores perdidos en un vector, bastará con hacer:

```
>> sum(isnan(v))
```

```
ans =  2
```

y, análogamente, si queremos contar el número de valores no perdidos:

```
>> sum(~isnan(v))
```

```
ans =  5
```

Asimismo, si queremos seleccionar sólo aquellos valores de `v` que no están perdidos, podemos ejecutar:

```
>> v(~isnan(v))
```

```
ans =

   1   2   3   4   5
```

(podemos leer esta última expresión como que le estamos pidiendo a matlab: *"muéstrame los valores*

de `v` tales que no están perdidos")

En el caso de nuestro vector de temperaturas `T(:,15)`, el número de valores perdidos que contiene es:

```
>> sum(isnan(T(:,15)))
```

```
ans =  14
```

y el número de valores no perdidos:

```
>> sum(~isnan(T(:,15)))
```

```
ans =  86
```

Combinando todo lo anterior, si quisiéramos calcular la media de `T(:,15)` sin utilizar las funciones `nanmean()` ni `nansum()` podemos utilizar la siguiente sintaxis:

```
>> Temp= T(:,15);            % Valores de temperatura en el punto 15
 noPerdidos=~isnan(Temp);  % Vector que identifica los valores no perdidos
 n=sum(noPerdidos);        % Total de valores no perdidos
 media=sum(Temp(noPerdidos))/n
```

```
media =  22.591
```

Lectura de archivos csv con `readtable`

Para leer el archivo *time.csv* hemos de proceder de manera diferente. Si lo abrimos con LibreOffice o Excel vemos que su contenido son fechas y horas:

	A	B
1	2013-02-18 12:28:44	
2	2013-02-18 14:34:01	
3	2013-02-18 16:59:21	
4	2013-02-18 19:25:21	
5	2013-02-18 21:54:55	
6	2013-02-19 00:18:42	
7	2013-02-19 02:23:02	
8	2013-02-19 04:27:01	
9	2013-02-19 06:36:58	
10	2013-02-19 08:43:33	
11	2013-02-19 11:15:33	
12	2013-02-19 13:15:51	
13	2013-02-19 15:41:59	
14	2013-02-19 17:48:14	
15	2013-02-19 19:34:56	
16	2013-02-19 21:43:29	
17	2013-02-19 23:30:37	
18	2013-02-20 01:45:57	
19	2013-02-20 03:31:01	
20	2013-02-20 05:12:26	
21	2013-02-20 06:54:10	
22	2013-02-20 09:28:25	

En particular, podemos observar que aparecen los símbolos "-" y ":" como separadores para marcar las fechas y las horas. La función `csvread` es capaz de leer ficheros que contengan solamente números; si el archivo contiene algún otro símbolo como en este caso, matlab nos devuelve un error:

```
>> file=fopen("datos/time.csv")
 time=csvread(file);
```

```
Error using dlmread (line 147)
Mismatch between file and format character vector.
Trouble reading 'Numeric' field from file (row number 1, field number 1) ==>
"2013-02-18 12:28:44"\n

Error in csvread (line 48)
    m=dlmread(filename, ',', r, c);
```

Para leer archivos que contengan caracteres no numéricos podemos utilizar la función `readtable`:

⊕

```
>> fh=readtable("datos/time.csv");
```

Cuando se usa esta función, el objeto que se crea (`fh` en nuestro caso) es un objeto de clase `table`, que tiene algunas particularidades específicas de esta clase (en esencia una tabla en matlab es un contenedor con variables de distinta naturaleza ordenadas a modo de hoja de cálculo). Para convertir una tabla en una matriz estándar usamos la función `table2array`:

```
>> FechaHora=table2array(fh);
```

Como ventaja adicional, la función `table2array` identifica que la variable *FechaHora* que hemos creado es de clase `datetime`, que es la clase específica que le permite a matlab manejar adecuadamente variables de fecha y hora. Si le pedimos a matlab que nos muestre los diez primeros valores de esta variable obtenemos la salida siguiente:

```
>> FechaHora(1:10)
```

```
ans =

  10×1 datetime array

   2013-02-18 12:28:44
   2013-02-18 14:34:01
   2013-02-18 16:59:21
   2013-02-18 19:25:21
   2013-02-18 21:54:55
   2013-02-19 00:18:42
   2013-02-19 02:23:02
   2013-02-19 04:27:01
   2013-02-19 06:36:58
   2013-02-19 08:43:33
```

y en particular podemos comprobar que la fecha y hora a que se tomaron los datos en el punto 15 de muestreo fue:

```
>> FechaHora(15)
```

```
ans =

  10×1 datetime array

   2013-02-19 19:34:56
```