Práctica 1

15/03/2024
Fundamentos de Control

Juan López Puebla

Introducción a MATLAB

Matrices y vectores

Table of Contents

MATLAB fue inicialmente diseñado como una herramienta para facilitar el cálculo matricial, de hecho el nombre MATLAB deriva de “MATrix LABoratory” (Laboratorio de Matrices).

Hoy día, MATLAB es un sistema interactivo y un lenguaje de programación de carácter científico y técnico que es utilizado, con éxito, tanto en el ámbito académico como en el ámbito industrial.

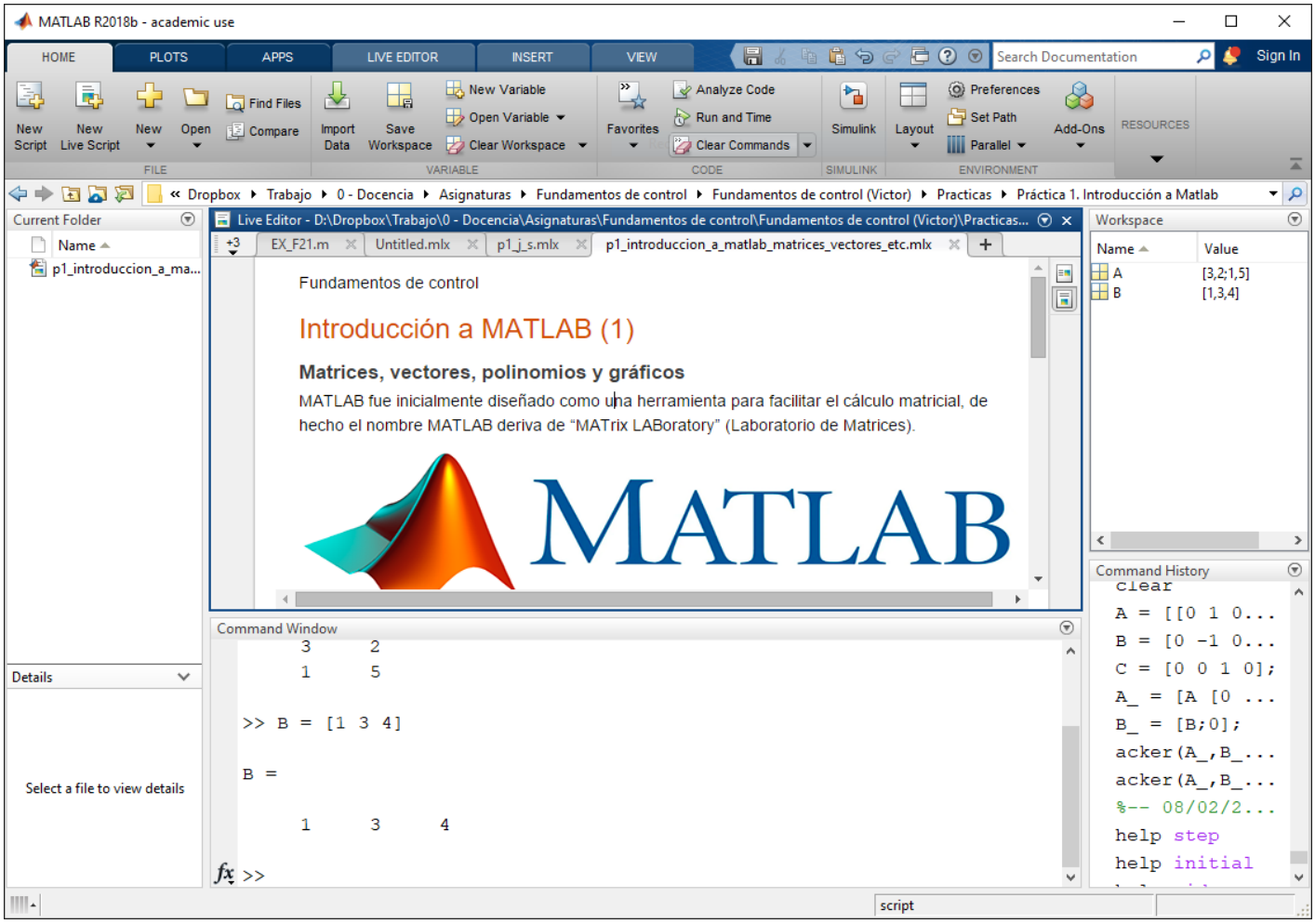
Las características principales de MATLAB son:

- Orientado al cálculo matemático y científico.
- Uso de las matrices como elemento básico.
- Potencia de representación gráfica.
- Sistema abierto.
- Facilidad de uso.
- Lenguaje de programación.
- Aplicable a multitud de campos (versátil).

1. El entorno MATLAB

La ventana de MATLAB se encuentra dividida en cuatro partes fundamentales (subventanas) que inicialmente muestran la siguiente información:

- Subventana izquierda: muestra el conteido del **directorio de trabajo actual** y, en la parte inferior, detalles del fichero que tengamos seleccionado (versión del fichero, autor, etc.).
- Subventana central-arriba: contiene el **editor de código** (ficheros con extensión *.m*) o de scripts vivos (*live scripts*, extensión *.mlx*).
- Subventana central-abajo: **ventana de comandos** (*Command Window*) donde se introducen los comandos propios de MATLAB. Esta ventana tiene capacidad de "memorizar" los comandos que han sido introducidos con anterioridad y consultarlos pulsando la tecla arriba.
- Subventana superior derecha: es el denominado **espacio de trabajo** (*Workspace*) y en ella aparecen todas las variables que se han utilizado en la ventana de comandos. Esta ventana tiene la capacidad de crear nuevas variables, eliminar variables y de visualizar y editar los valores de las variables existentes.



Además, en la parte superior se muestra la ruta del directorio de trabajo actual, la cual puede cambiarse para guardar y ejecutar comandos desde una carpeta definida por el propio usuario.

Una vez nos han presentado el entorno, vamos a empezar a jugar con él.

Pista: para cambiar esta distribución de ventanas inicial, o mostrar algunas ocultas por defecto (por ejemplo, el historial de comandos, *Command history*), puede emplearse la opción *Layout* dentro del menú *Home*.

2. Matrices y vectores

El elemento básico o primitiva de trabajo de MATLAB son las **matrices**. Para MATLAB, por defecto, cualquier variable es considerada como una matriz rectangular, y esta no necesita ser dimensionada previamente para ser usada. La mayoría de las funciones de MATLAB están diseñadas para operar directamente con matrices.

Los elementos de una matriz pueden ser números enteros, reales, complejos, o expresiones matemáticas, entre otras muchas cosas.

2.1 Definición de matrices

Las matrices en MATLAB pueden ser creadas de diversas formas, pero la más común es emplear una lista explicita de elementos. Por ejemplo:

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

```
A =

     2     4     5
     1     3     8
```

Pista: si una operación de asignación (=) no incluye un punto y coma al final (;), MATLAB mostrará el contenido de la variable asignada tras ejecutarla.Si lo incluye, a esto se le denomina modo *no echo*.

Esta creación tiene varios aspectos a destacar:

- La matriz se denomina `A`,
- se incluye un símbolo de asignación `=`,
- la definición de la matriz comienza con un corchete `[`,
- cada columna se separa por una coma `,`,
- cada fila se separa empleando un punto y coma `;`, y
- la definición de la matriz finaliza con un corchete `]`.

MATLAB utiliza la notación matemática para referenciar un elemento de una matriz. Es decir, para referenciar el elemento de la fila `i` y la columna `j` de la matriz `A` se usaría la expresión `A(i,j)`.

Tarea 1: Crea la matriz `A` tal que:

$$A = \begin{bmatrix} -1 & 2 & 3 & -4 \\ 5 & 6 & -7 & 8 \\ 9 & -10 & 11 & 12 \\ 13 & -14 & -15 & 16 \end{bmatrix}$$

y consulta su elemento en la fila 2 y columna 3.

```
% Tu código aquí, para ejecutarlo puedes pulsar el botón Run o usar la
% combinación de teclas Ctrl+Enter
A = [-1 2 3 -4;5 6 -7 8;9 -10 11 12;13 -14 -15 16]
```

```
A = 4×4
    -1     2     3    -4
     5     6    -7     8
     9   -10    11    12
```

```
    13   -14   -15    16
```

```
elemnt = A(2,3)
```

```
elemnt = -7
```

```

```

Tarea 2: Crea la matriz B tal que:

$$B = \begin{bmatrix} 1+i & 2+2i \\ 3+3i & 4+4i \end{bmatrix}$$

y consulta su elemento en la fila 2 y columna 1.

Pista: En MATLAB la unidad imaginaria básica puede representarse empleando `i` o `j` indistintamente.

```
% Tu código aquí
B = [1+i 2+2*i;3+3*i 4+4*i]
```

```
B = 2×2 complex
   1.0000 + 1.0000i   2.0000 + 2.0000i
   3.0000 + 3.0000i   4.0000 + 4.0000i
```

```
element = B(2,1)
```

```
element = 3.0000 + 3.0000i
```

Matrices de uso frecuente

MATLAB incorpora una serie de comandos para crear matrices de uso frecuente. En las siguientes celdas de código puedes jugar con ellas. Fijaté en los parámetros de entrada:

```
% Matriz de ceros de dimensión 2x2
M = zeros(2)
```

```
M = 2×2
     0     0
     0     0
```

```
% Matriz de ceros de dimensión 2x3
M = zeros(2,3)
```

```
M = 2×3
     0     0     0
     0     0     0
```

```
% Matriz de unos (identidad) de dimensión 3x3
M = eye(3)
```

```
M = 3×3
     1     0     0
     0     1     0
     0     0     1
```

```
% Matriz de números aleatorios (distribución uniforme)
```

```
M = rand(3)
```

```
M = 3×3
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

```
% Matriz de números aleatorios (distribución normal)
M = randn(3)
```

```
M = 3×3
    2.7694    0.7254   -0.2050
   -1.3499   -0.0631   -0.1241
    3.0349    0.7147    1.4897
```

2.2 Definiendo vectores

Partiendo de la base de que la primitiva de trabajo es la matriz, MATLAB también puede trabajar con **escalares** (matrices 1x1) y con **vectores fila** (matrices 1xn) o **columna** (matrices nx1).

Tarea 3: Crea el vector columna v tal que:

$$v = \begin{bmatrix} 1 \\ -2 \\ 3 \\ 4 \end{bmatrix}$$

```
% Tu código aquí
v = [1;-2;3;4]
```

```
v = 4×1
     1
    -2
     3
     4
```

```

```

2.3 Trabajando con matrices

Traspuesta de una matriz

En MATALAB la traspuseta de una matriz se obtiene con el operador tilde `'`. Por ejemplo:

```
A'

ans =

     2     1
     4     3
     5     8

B = [1 2 4]'

B =
```

```
1
2
4
```

Tarea 4: Calcular, a partir de la matriz `A` previamente definida, una nueva matriz `C`, tal que: $C = A^T$

```
% Tu código aquí
C = A'
```

```
C = 4×4
    -1     5     9    13
     2     6   -10   -14
     3    -7    11   -15
    -4     8    12    16
```

```
```

Aritmétrica de matrices

Los siguientes comandos implementan operaciones aritmétricas básicas:

- `+`: Suma de matrices.
- `-`: Resta de matrices.
- `*`: Producto de matrices.
- `/`: División matricial por la derecha (A/B equivale a $A \times B^{-1}$) .
- `\`: División matricial por la izquierda ($A\backslash B$ equivale a $A^{-1} \times B$).
- `^`: Operador potencia.
- `inv`: inversa de una matriz.
- `'`: traspuesta de una matriz.

Tarea 5: Calcular, a partir de la matriz `A` y el vector `v` previamente definidos, el producto: $C = A \times v$. ¿Qué pasaría si intento realizar el producto $C = v \times A$? ¿Por qué?

```
% Tu código aquí
C = A * v
```

```
C = 4×1
   -12
     4
   110
    60
```

```
%No se puede multiplicar v con A dado que v es 4x1 y A 4x4. Al hacer A*v
%obtenemos (4x4)(4x1) coinciden los cuatros luego nos quedara una matríz
%resultante de 4x1 y se puede multiplicar. Sin embargo al hacerlo al revés
```

```
%no nos coincide luego no se pueden multiplicar.
```

Tarea 6: Calcular, a partir de la matriz `A` y el vector `v`, una nuevas matrices `x1` y `x2`, tales que:

$$\begin{cases} A * \mathrm{x1} = v \\ \mathrm{x2} * A = v' \end{cases}$$

```
% Tu código aquí
x1 = A^-1 * v
```

```
x1 = 4×1
    0.7508
   -0.2222
    0.0303
   -0.5261
```

```
x2 = v' * A^-1
```

```
x2 = 1×4
   -0.9091    0.0000    0.2727   -0.1818
```

Tarea 7: Calcular, a partir de la matriz `A`, una nueva matriz `AA`, tal que: $\mathrm{AA} = A^2$.

Pista: Usa el operador potencia

```
% Tu código aquí

AA = A^2
```

```
AA = 4×4
   -14    36    76    -8
    66     4  -224    72
   196  -320    38   208
   -10  -132  -268   -88
```

La versatilidad tiene operador, dos puntos :

El operador dos puntos **:** es uno de los operadores más versátiles e importantes de MATLAB, ya que permite definir vectores, referencias, submatrices, etc.

Por ejemplo, puede definir un vector que posea los valores desde 1 hasta 10 de forma compacta:

```
x = 1:10

x =

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

Nótese que la sintaxis resulta `vector = inicio:fin`. Si se desea que el paso sea disinto de 1, dicho paso se puede introducir entre los valores de inicio y de fin: `vector = inicio:paso:fin`. Por ejemplo:

```
x =
```

```
     1     3     5     7     9
```

Tarea 8. Crea un vector u que contenga toda la serie de valores reales comprendidos en el intervalo $[0, 10]$ que resultan de la consideración de incrementos de una décima entre cada dos elementos consecutivos.

```
% Tu código aquí
u = 0:0.1:10
```

```
u = 1×101
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000 ···
```

Como se ha introducido, el operador dos puntos **:** también puede emplearse para referenciar parte de una matriz. Por ejemplo, el siguiente código extrae la sumatriz compuesta por las dos primeras filas y columnas de `D` y la almacena en una nueva matriz `S1`:

```
D =

     2     4     5
     1     3     8

S1 = D(1:2,1:2)
S1 =

     2     4
     1     3
```

Otra manera de referenciar parte de una matriz es indicar directamente las filas o columnas a usar. En cualquier caso, el uso del operador dos puntos sin emplear inicio o fin indica que se desea referenciar todas las filas o columnas. Ejemplo:

```
S1 = D(:,[1 3])

S1 =

     2     5
     1     8
```

Tarea 9: Obtener una matriz `C` constituida por las filas 1 y 4 de la matriz `A`.

```
% Tu código aquí

C = A([1 4],:)
```

```
C = 2×4
    -1     2     3    -4
    13   -14   -15    16
```

Además de para asignar a una matriz `C` una submatriz de otra matriz `A`, también se puede usar la referncia a fila o columnas para cambiar el valor de dicha submatriz referenciada, incluso para eliminarla. Por ejemplo, el siguiente código cambia el valor de la segunda fila de la matriz `D`:

```
D =

     1     3     5
     2     4     6

D(2,:) = [3 8 3]

D =

     1     3     5
     3     8     3
```

y el siguiente elimina dicha fila:

```
D(2,:) = []

D =

     1     3     5
```

Tarea 10: Obtener la matriz resultante de la eliminación de la columna 2 de la matriz `C`.

```
% Tu código aquí
C(:,2)=[]
```

```
C = 2×3
    -1     3    -4
    13   -15    16
```

```

```

Un comando muy útil: find

El comando **find** permite devolver los índices de los elementos de una matriz que cumplan con una cierta condición. Esta condición puede incluir comparadores como <, <=, >, >= o ==. Los índices siguen un orden resultado de concatenar las columnas de la matriz. Por ejemplo, los elementos de una matriz `D` que sean mayores de 3:

```
D =

     5     3     2
     2     4     6

idx = find(D>3)

idx =

     1
     4
```

```
     6
```

Este comando acepta un segundo argumento que permite escificar el número de elementos máximos que queremos en la respuesta. Por ejemplo si sólo estamos interesados en los dos primeros índices que cumplan la condición:kkk

```
idx = find (D > 3, 2)

idx =

     1
     4
```

Los índices devueltos por **`find`** se pueden usar para referenciar dichos elementos en la matriz `D` y modificarlos. Por ejemplo:

```
D(idx) = 9

D =

     9     3     2
     2     4     9
```

Si en vez de los índices que ocupan los elementos que cumplen con la condición fijada, estuvieramos interesados en la fila y columna que estos ocupan, podríamos obtenerlos empleando la siguiente sintaxis en la llamada a **`find`**:

```
D =

     5     3     2
     2     4     6

[row,col] = find (D > 3, 2)

row =

     1
     2


col =

     1
     2
```

Nótese que `row` and `col` son vectores columna que almacenan la fila y columna donde aparecen los dos primeros elementos en `D` que cumplen con la condición.

Tarea 11: Calcular, a partir de la matriz `A`, una nueva matriz `D` que contenga un cero en el lugar en el que la matriz `A` contenga un número par.

Pista: MATLAB provee el comando **`mod(a,b)`** que devuelve el módulo de dividir `a` entre `b`.

```
% Tu código aquí
idx = find(mod(A,2)==0);
```

```
A(idx)= 0;
D=A
```

```
D = 4×4
    -1     0     3     0
     5     0    -7     0
     9     0    11     0
    13     0   -15     0
```

```

```

Tarea 12: Calcular, a partir de la matriz `A`, una nueva matriz `E` constituida por las filas completas de la matriz `A` que contengan múltiplos de `6`.

```
% Tu código aquí
A
```

```
A = 4×4
    -1     2     3    -4
     5     6    -7     8
     9   -10    11    12
    13   -14   -15    16
```

```
[row,col]=find(mod(A,6)==0)
```

```
row = 2×1
     2
     3
col = 2×1
     2
     4
```

```
E = A(row,:)
```

```
E = 2×4
     5     6    -7     8
     9   -10    11    12
```

```

```

Introducción a MATLAB

Manejo de polinomios

Table of Contents

A lo largo de nuestra vida académica hemos ido acumulando experiencia con **polinomios**: expresiones algebraicas en las que intervienen números (coeficientes) y letras (variables) que están relacionadas mediante operaciones como sumas, multiplicaciones y/o potencias. Los polinomios se definen en MATLAB mediante vectores fila con los coeficientes del polinomio en cuestión en orden de potencias decrecientes. Por ejemplo, si queremos definir el polinomio $p(x) = x^3 - 2x - 5$ se procedería como sigue:

```
% p = x^3 - 2*x - 5
% Polinomio de tercer grado, con lo que:
% Coeficiente que acompaña a x^3: 1
% Coeficiente que acompaña a x^2: 0
% Coeficiente que acompaña a x^1: -2
% Término independiente: -5
% Resultando en la definición:
p = [1 0 -2 -5]
```

```
p =

     1     0    -2    -5
```

De manera general, el polinomio $p(x) = p_2x^2 + p_1x + p_0$ se representa en Matlab como:

```
p = [p2 p1 p0]
```

Una vez presentados nuestros protagonistas, ¡trabajemos un poco con ellos!

1. Trabajando con polinomios

1.1 Evaluando polinomios

Para evaluar un polinomio en un cierto punto MATLAB nos ofrece el comando **`polyval`**. Por ejemplo, $p(4)$ se puede calular como:

```
res = polyval(p,4)
```

```
res =
```

```
51
```

Alternativamente, también se puede evaluar un polinomio desde el punto de vista matricial usando **`polyvalm`**.

Por ejemplo, el polinomio $p(x) = x^3 - 2x - 5$ se convertiría en la expresión matricial $p(X) = X^3 - 2X - 5I$, dónde X es una matriz cuadrada y I es la matriz identidad. Por ejemplo:

```
p = [1 0 -2 -6];
X = [3 -2; 0 3];
Y = polyvalm(p,X)

Y =

    15   -50
     0    15
```

Tarea 1: Definir el polinomio $q(x) = -2x^5 + 3x^3 - 2x + 5$ y evaluarlo en $x = 2$.

```
% Tu código aquí
q = [-2 0 3 0 -2 5]
```

```
q = 1×6
    -2     0     3     0    -2     5
```

```
x = 2
```

```
x = 2
```

```
polyval(q,x)
```

```
ans = -39
```

```

```

Resultado esperado: `q_x = -39`

1.2 Obteniendo las raíces de un polinomio

Las raíces de un polinomio pueden calcularse empleando el comando **`roots`**. Por ejemplo:

```
r = roots(p)

r =

   2.0946 + 0.0000i
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

Tarea 2: Calcula las raíces del polinomio $q(x)$ previamente definido.

```
% Tu código aquí
roots(q)
```

```
ans = 5×1 complex
  -1.1927 + 0.6181i
  -1.1927 - 0.6181i
   1.3854 + 0.0000i
   0.5000 + 0.8660i
   0.5000 - 0.8660i
```

1.3 Multiplicando polinomios

Si estamos trabajando con dos polinomios, estos pueden multiplicarse empleando el comando **conv**. Por ejemplo, si se desea multiplicar el ya conocido polinomio $p(x) = x^3 - 2x - 5$ por $p_2(x) = 4x^3 + 5x + 6$

```
p2 = [4 5 6]

p2 =

     4     5     6

p3 = conv(p,p2)

p3 =

     4     5    -2   -30   -37   -30
```

Lo que da como resultado el polinomio $p_3(x) = 4x^5 + 5x^4 - 2x^3 - 30x^2 - 37x - 30.$

Este comando también es útil para construir un polinomio a partir de su factorización. Por ejemplo si $q(x) = (x-2)(x+3)(x+5i)(x-5i)$ podemos obtener el polinomio del que proviene dicha factorización como sigue:

```
q = conv(conv(conv([1 -2],[1 3]),[1 5i]),[1 -5i])

q =

     1     1    19    25  -150
```

Lo que da como resutlado $q(x) = x^4 + x^3 + 19x^2 + 25x - 150.$

Nota: Nótese que la convolución de dos vectores es equivalente a la multiplicación si estos representan coeficientes polinomiales.

Tarea 3: Obtén el polinomio $q(x)$ resultante de la factorización $q(x) = (x+3)(x+2+3i)(x+2-3i)$. Comprueba que el polinomio obtenido es el correcto calulando sus raíces y comparándolas con la factorización inicial.

```
% Tu código aquí
 q = conv(conv([1 3],[1 2+3*i]),[1 2-3*i])
```

```
q = 1×4
     1     7    25    39
```

```
roots(q)
```

```
ans = 3×1 complex
  -2.0000 + 3.0000i
  -2.0000 - 3.0000i
  -3.0000 + 0.0000i
```

```
roots([1 2+3*i])
```

```
ans = -2.0000 - 3.0000i
```

```
roots([1 2-3*i])
```

```
ans = -2.0000 + 3.0000i
```

1.4 Obteniéndo el polinomio característico de una matriz

El polinomio característico de una matriz se calcula como $p_A(\lambda) = \mathrm{def}(A - \lambda I)$. En álgebra, este polinomio característico es una herramienta ampliamente utilizada ya que provee gran cantidad de información sobre la matriz, como por ejemplo los valores propios, el determinante y su traza.

En MATLAB el polinomio característico de una matriz puede obtenerse con el comando **`poly`**. Por ejemplo:

```
A =

     2     3     4
     4     5     2
     1     8     6

pc = poly(A)

pc =

    1.0000  -13.0000   20.0000  -70.0000
```

Tarea 4: Calcular, a partir de la matriz `A=[2 4 3; -1 3 2; 8 3 -9]`, un vector p constituido por los coeficientes del polinomio característico de la matriz `A`.

```
% Tu código aquí
 A=[2 4 3; -1 3 2; 8 3 -9];
 p = poly(A)
```

```
p = 1×4
    1.0000    4.0000  -65.0000  119.0000
```

```

```

Resultado esperado:

```
pc = 1×4
     1.0000    4.0000  -65.0000  119.0000
```

Tarea 5: El comando **`poly`** tiene un segundo uso,interesante. Para ilustrarlo, calcular, a partir del vector `p = [1 0 -2 -5]`, un nuevo vector `r` constituido por las raíces de dicho polinomio.

```
% Tu código aquí
p = [1 0 -2 -5];
r= roots(p)
```

```
r = 3×1 complex
   2.0946 + 0.0000i
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

poly permite obtener el vector de coeficientes de un polinomio a partir de un vector que contiene sus raíces. Por ejemplo, empleando las raíces de nuestro polinomio de ejemplo p:

```
p4 = poly(r)
```

```
p4 =

    1.0000   -0.0000   -2.0000   -5.0000
```

De este modo, los comandos **poly** y **roots** pueden verse como inversos: con **roots** calculo raíces que puedo convertir en polinomio con **poly**, y viceversa.

Tarea 6: Calcular, a partir del vector r, un nuevo vector p4 constituido por los coeficientes de un polinomio cuyas raíces son los elementos del vector r. Comprobar que el vector p4 obtenido coincide con el vector p calculado anteriormente.

```
% Tu código aquí
p4 = poly(r)
```

```
p4 = 1×4
    1.0000   -0.0000   -2.0000   -5.0000
```

[BOMBA] El comando salvador: help

Si tenemos dudas sobre el uso de algún comando o función, podemos emplear el comando **help** para obtener ayuda sobre el mismo. Por ejemplo, ejecuta la siguiente celda de código para revisar la documentación relativa al comando **poly**:

```
help poly
```

Si seguimos teniendo dudas, vistar la documentación del comando puede ser útil.

1.5 División, cociente y resto

El cociente y el resto de la división de dos polinomios puede obtenerse mediante el comando **deconv**. Por ejemplo:

```
[q,r] = deconv(p,p2)

q =

    0.2500   -0.3125


r =

         0         0   -1.9375   -3.1250
```

Tarea 7: Calcular, a partir de los vectores `p = [1 0 -2 -5]` y `p2 = [1 4 5 6]`, unos nuevos vectores `coc` y `res` constituidos, respectivamente, por los coeficientes de los polinomios cociente y resto de la división del polinomio cuyos coeficientes son los términos del vector `p2` entre el polinomio cuyos coeficientes son los términos del vector `p`.

¿Un poco de ayuda sobre **deconv**?

```
help deconv
```

```
deconv - Least-squares deconvolution and polynomial division
   This MATLAB function deconvolves a vector h out of a vector y using
   polynomial long division, and returns the quotient x and remainder r
   such that y = conv(x,h) + r.

   Polynomial Long Division
     [x,r] = deconv(y,h)

   Least-Squares Deconvolution
     [x,r] = deconv(y,h,shape)
     [x,r] = deconv(___,Name=Value)

   Input Arguments
     y - Input signal to be deconvolved
       row or column vector
     h - Impulse response or filter used for deconvolution
       row or column vector
     shape - Subsection of convolved signal
       "full" (default) | "same" | "valid"

   Name-Value Arguments
     Method - Deconvolution method
       "long-division" (default) | "least-squares"
     RegularizationFactor - Tikhonov regularization factor
       0 (default) | real number

   Output Arguments
     x - Deconvolved signal or quotient from division
       row or column vector
     r - Residual signal or remainder from division
       row or column vector

   Examples
     Polynomial Division
     Least-Squares Deconvolution of Fully Convolved Signal
     Least-Squares Deconvolution of Central Part of Convolved Signal
     Least-Squares Deconvolution Problem with Infinite Solutions
     Specify Regularization Factor for Noisy Signal
```

```
    See also conv, residue

    Introduced in MATLAB before R2006a
    Documentation for deconv
```

```
% Tu código aquí
p = [1 0 -2 -5];
p2 = [1 4 5 6];
[coc,res] = deconv(p2,p)
```

```
coc = 1
res = 1×4
     0     4     7    11
```

Comprueba que el resultado es correcto calculando el producto del cociente por p y sumandole el resto.

```
% Tu código aquí
p2 = (coc * p) + res
```

```
p2 = 1×4
     1     4     5     6
```

Resultado esperado:

```
p2 = 1×4
     1     4     5     6
```

Introducción a MATLAB

Funciones matemáticas top

Table of Contents

1.Biblioteca de funciones predefinidas

Para facilitarnos la vida MATLAB incorpora una serie de funciones matemáticas. Vamos a revisar algunas de las más usadas. Algunas os sonarán de scripts anteriores, como **`find`**, **`conv`** o **`deconv`**:

Operaciones con matrices/números:

- **`abs:`** Si se aplica a un número real calcula su valor absoluto y si se aplica a un número complejo calcula su módulo.
- **`all:`** Devuelve verdad (uno) si todos los elementos del vector al que se aplica esta función son verdad (distintos de cero).
- **`any:`** Devuelve verdad (uno) si algún elemento del vector al que se aplica esta función es verdad (distinto de cero).
- **`exp:`** Aplica la función exponencial.
- **`find:`** Devuelve un vector con los índices de todos los elementos no nulos del vector al que se aplica. find(a>100) devuelve los índices de todos **`los`** elementos del vector a que son mayores que 100.
- **`log:`** Calcula el logaritmo neperiano.
- **`log10:`** Calcula el logaritmo decimal.
- **`max:`** Calcula el valor máximo de los elementos de un vector. Si se desea, permite conocer en qué posición del vector se encuentra el elemento de valor máximo.
- **`min:`** Calcula el valor mínimo de los elementos de un vector. Si se desea, permite conocer en qué posición del vector se encuentra el elemento de valor mínimo.
- **`mod / rem:`** Calcula el resto de una división.
- **`sqrt:`** Calcula la raíz cuadrada.

Operaciones con polinomios:

- **`conv:`** Calcula el producto de dos polinomios.
- **`deconv:`** Calcula el cociente y el resto de la división de dos polinomios.
- **`poly:`** Genera el polinomio característico de una matriz. Genera el vector de coeficientes de un polinomio a partir de un vector que contiene sus raíces.
- **`roots:`** Calcula las raíces de un polinomio a partir de su vector de coeficientes.

Trabajando con funciones:

- **`fminsearch:`** Calcula el valor de la variable independiente para el cual una función matemática alcanza el valor mínimo. Hay que especificarle como parámetro de entrada el punto en el que empieza a buscar el mínimo.
- **`quad:`** Obtiene la integral definida de una función matemática por aproximación cuadrática (método de Simpson). La función considerada debe indicarse entre comillas simples.
- **`quadl:`** Obtiene la integral definida de una función matemática por aproximación cuadrática (método de Lobatto). La función considerada debe indicarse entre comillas simples.

Trigonometría:

- **`cos:`** Calcula el coseno de una cantidad expresada en radianes.
- **`sin:`** Calcula el seno de una cantidad expresada en radianes.

Tarea 1: Calcular el seno de 60º.

Pista: MATLAB, en su eterna sabiduría, nos provee de los comandos **rad2deg** y **deg2rad**. Puedes emplear **help** para comprobar como se comportan.

```
% Tu código aquí
p = deg2rad(60)
```

```
p = 1.0472
```

```
q = sin(p)
```

```
q = 0.8660
```

```
help rad2deg
```

```
 rad2deg - Convert angle from radians to degrees
    This MATLAB function converts angle units from radians to degrees for
    each element of R.

    Syntax
      D = rad2deg(R)

    Input Arguments
      R - Angle in radians
        scalar | vector | matrix | multidimensional array

    Output Arguments
      D - Angle in degrees
        scalar | vector | matrix | multidimensional array

    Examples
      pi in Degrees
      Spherical Distance

    See also deg2rad

    Introduced in MATLAB in R2015b
    Documentation for rad2deg
    Other uses of rad2deg
```

Resultado esperado:

```
s_rad = 0.8660
```

Tarea 2: El siguiente código define una función `fun` con dos variables independientes `x(1)` y `x(2)`, y un punto de partida `x0`. Calcula cual es el valor de dichas variables para el cual la función alcanza el valor mínimo.

```
fun = @(x)100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;
x0 = [-1.2,1];
% Tu código aquí
help fminsearch
```

```
 fminsearch - Find minimum of unconstrained multivariable function using derivative-free method
    Nonlinear programming solver.

    Syntax
      x = fminsearch(fun,x0)
      x = fminsearch(fun,x0,options)
      x = fminsearch(problem)
      [x,fval] = fminsearch(___)
      [x,fval,exitflag] = fminsearch(___)
      [x,fval,exitflag,output] = fminsearch(___)

    Input Arguments
      fun - Function to minimize
        function handle | function name
      x0 - Initial point
        real vector | real array
      options - Optimization options
        structure such as optimset returns
      problem - Problem structure
        structure

    Output Arguments
      x - Solution
        real vector | real array
      fval - Objective function value at solution
        real number
      exitflag - Reason fminsearch stopped
        integer
      output - Information about the optimization process
        structure

    Examples
      Minimize Rosenbrock's Function
      Monitor Optimization Process
      Minimize a Function Specified by a File
      Minimize with Extra Parameters
      Find Minimum Location and Value
      Inspect Optimization Process

    See also fminbnd, optimset, Optimize

    Introduced in MATLAB before R2006a
    Documentation for fminsearch
```

```
x = fminsearch(fun,x0)
```

```
x = 1×2
```

```
     1.0000    1.0000
```

```

```

Resultado esperado:

```
x = 1×2
     1.0000    1.0000
```

2. Variables predefinidas/Constantes

MATLAB también incorpora una serie de variables predefinidas que se podrían interpretar como valores constantes, y que pueden ser directamente añadidos a expresiones. Juega un poco con ellas ejecutando estas celdas de código:

```
% La famosa constante pi
pi
```

```
ans = 3.1416
```

```
% Valor infinito
Inf
```

```
ans = Inf
```

```
% Valor no un número (not a number)
NaN
```

```
ans = NaN
```

```
% Parte imaginaria de un número complejo
1i
```

```
ans = 0.0000 + 1.0000i
```

```
% Parte imaginaria de un número complejo
1j
```

```
ans = 0.0000 + 1.0000i
```

Tarea 3: Calcula la circunferencia de un círculo que tiene como diámetro 4 metros, empleando la constante `pi`.

```
% Tu código aquí
c = 2 * pi * 2
```

```
c = 12.5664
```

```

```

Resultado esperado: `c = 12.5664`

Introducción a MATLAB

Gráficos

Table of Contents

Uno de los puntos fuertes de MATLAB es el amplio abanico de posibilidades que ofrece a la hora de mostrar visualmente, en forma de gráficas, los datos con los que se está trabajando. Vamos a ver alguna de las opciones más comunes, ya que el estudio de la totalidad de funcionalidades ofrecidas requeriría de una asignatura completa para ello. A lo largo de la asignatura iremos descubriendo más posibilidades.

1. Uno para gobernarlos a todos: plot

El comando **plot** se postula como uno de los más usados a la hora de generar gráficos bidimensionales. Veamos su documentación:

```
help plot
```

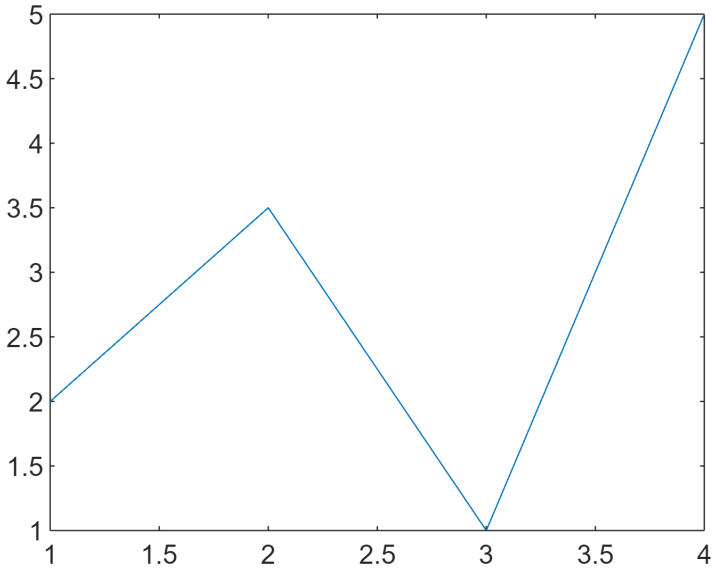
```
  plot - 2-D line plot
     This MATLAB function creates a 2-D line plot of the data in Y versus the
     corresponding values in X.

     Vector and Matrix Data
       plot(X,Y)
       plot(X,Y,LineSpec)
       plot(X1,Y1,...,Xn,Yn)
       plot(X1,Y1,LineSpec1,...,Xn,Yn,LineSpecn)
       plot(Y)
       plot(Y,LineSpec)

     Table Data
       plot(tbl,xvar,yvar)
       plot(tbl,yvar)

     Additional Options
       plot(ax,___)
       plot(___,Name,Value)
       p = plot(___)

     Input Arguments
       X - x-coordinates
         scalar | vector | matrix
       Y - y-coordinates
         scalar | vector | matrix
       LineSpec - Line style, marker, and color
         string scalar | character vector
       tbl - Source table
         table | timetable
       xvar - Table variables containing x-coordinates
```

```
    string array | character vector | cell array | pattern |
    numeric scalar or vector | logical vector | vartype()
  yvar - Table variables containing y-coordinates
    string array | character vector | cell array | pattern |
    numeric scalar or vector | logical vector | vartype()
  ax - Target axes
    Axes object | PolarAxes object | GeographicAxes object

Name-Value Arguments
  Color - Line color
    [0 0.4470 0.7410] (default) | RGB triplet | hexadecimal color code |
    "r" | "g" | "b"
  LineStyle - Line style
    "-" (default) | "--" | ":" | "-." | "none"
  LineWidth - Line width
    0.5 (default) | positive value
  Marker - Marker symbol
    "none" (default) | "o" | "+" | "*" | "."
  MarkerIndices - Indices of data points at which to display markers
    1:length(YData) (default) | vector of positive integers |
    scalar positive integer
  MarkerEdgeColor - Marker outline color
    "auto" (default) | RGB triplet | hexadecimal color code | "r" |
    "g" | "b"
  MarkerFaceColor - Marker fill color
    "none" (default) | "auto" | RGB triplet | hexadecimal color code |
    "r" | "g" | "b"
  MarkerSize - Marker size
    6 (default) | positive value
  DatetimeTickFormat - Format for datetime tick labels
    character vector | string
  DurationTickFormat - Format for duration tick labels
    character vector | string

Examples
  Create Line Plot
  Plot Multiple Lines
  Create Line Plot From Matrix
  Specify Line Style
  Specify Line Style, Color, and Marker
  Display Markers at Specific Data Points
  Specify Line Width, Marker Size, and Marker Color
  Add Title and Axis Labels
  Plot Durations and Specify Tick Format
  Plot Coordinates from a Table
  Plot Multiple Table Variables on One Axis
  Specify Axes for Line Plot
  Modify Lines After Creation
  Plot Circle

See also title, xlabel, ylabel, xlim, ylim, legend, hold, gca, yyaxis,
  plot3, loglog, Line

Introduced in MATLAB before R2006a
Documentation for plot
Other uses of plot
```

Como podemos comprobar permite, entre otras cosas:

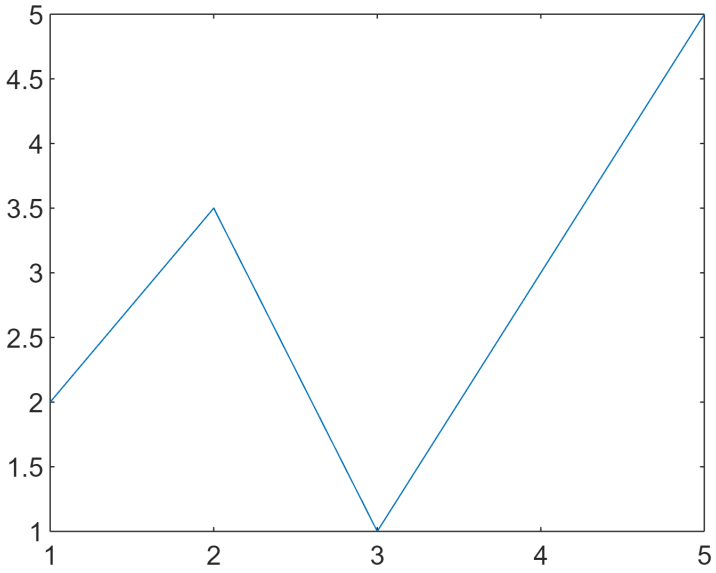
- dibujar las columnas de un vector Y frente a sus índices (orden en el que aparecen en el vector, comenzando por el índice 1):

```
Y = [2 3.5 1 5];
plot(Y)
```



- dibujar un vector X frente a otro Y. Ejecuta el siguiente ejemplo para comprobarlo:

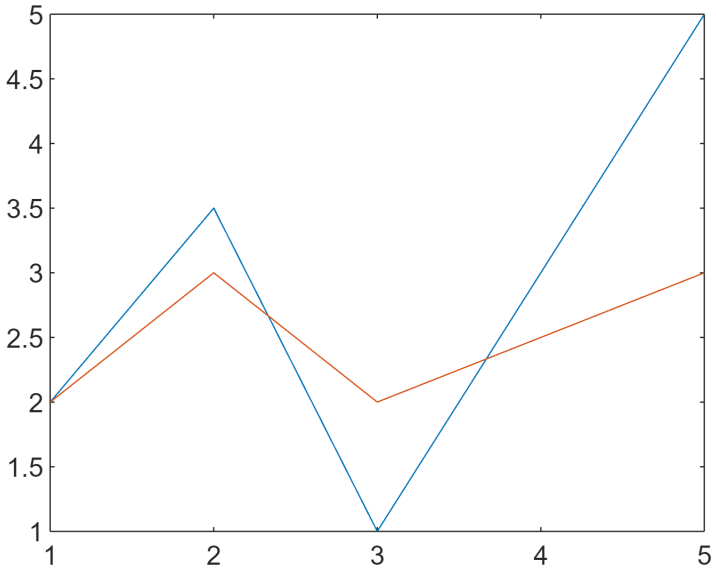
```
X = [1 2 3 5];
Y = [2 3.5 1 5];
plot(X,Y)
```



- mostrar varias gráficas distintas en una misma figura:

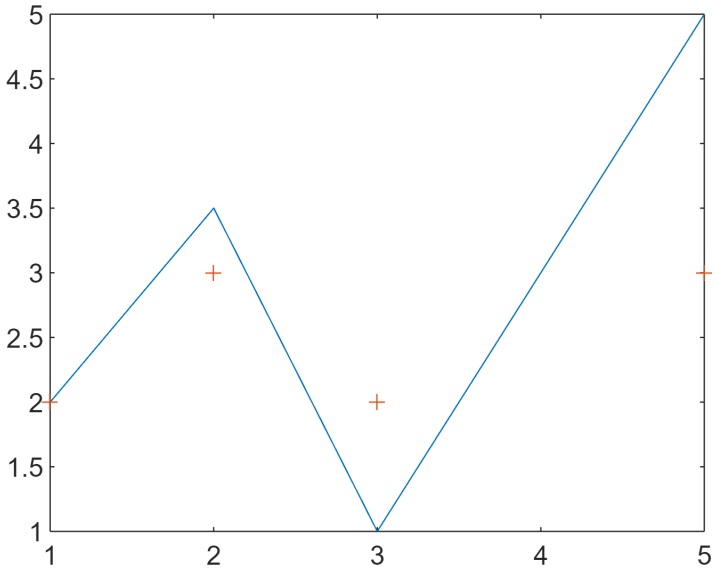
```
X2 = [1 2 3 5];
Y2 = [2 3 2 3];
```

```
plot(X,Y,X2,Y2)
```



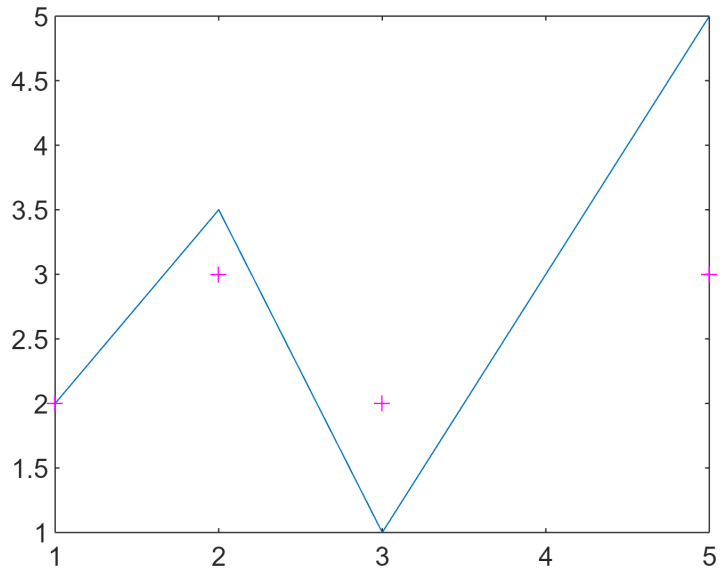
Por defecto MATLAB conecta cada uno de los puntos en los vectores anteriores con una línea, este comportamiento se puede modificar incluyendo un **símbolo** que represente dichos puntos:

```
plot(X,Y,X2,Y2,'+')
```



Otra posibilidad de personalización de las gráficas es elegir el **color** de estas. Por ejemplo para pintar en magenta:

```
plot(X,Y,X2,Y2,'+m')
```



Puedes consultar la documentación de MATLAB para comprobar los símbolos y colores que hay disponibles.

Algunas funcionalidades interesantes

De manera genérica, MATLAB ofrece una serie de comandos para trabajar con gráficos:

- **`clf`**: Borra la ventana gráfica activa.
- **`figure`**: Abre una ventana gráfica y le asigna el papel de ventana gráfica activa.
- **`ginput`**: Permite extraer, utilizando el ratón, las coordenadas de una serie de puntos situados sobre un gráfico. La activación de la tecla return finaliza el proceso
- **`grid`**: Muestra una rejilla sobre la gráfica situada en la ventana gráfica activa.
- **`hold`**: hold on hace que todos los gráficos sucesivos cuyo trazado sea ordenado se incluyan en la ventana gráfica activa. hold off deshace este efecto.

Tarea 1: Obtener una representación gráfica en el color magenta de la función $y = \sin(\theta)$para los valores de θ comprendidos en el intervalo $[-2\Pi, 8\Pi]$ con una resolución (paso) de un grado. Además, marcar en la misma con un '*' negro el punto en el que ésta alcanza un mínimo en el intervalo de θ comprendido entre 0 y 10. Asimismo, obtener la integral definida de dicha función en el intervalo de θ comprendido entre 1 y 2.

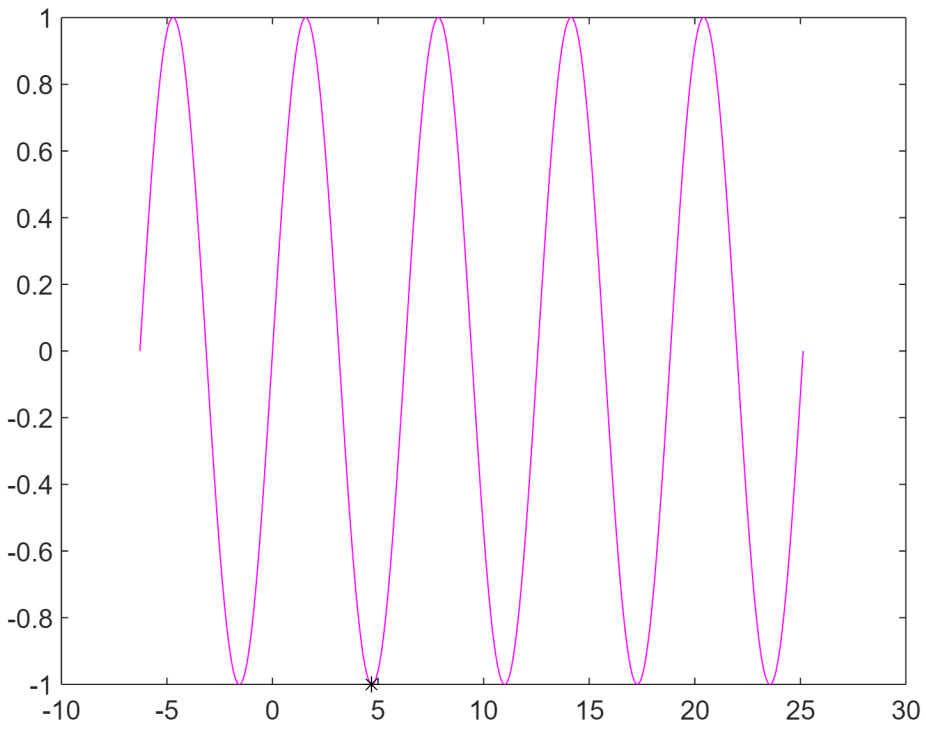
Pista: necesitarás usar los comandos **`sin`**, **`fminsearch`**, **`hold`**, y **`quad`**.

```
% Siempre que se trabaja con figuras es buena idea abrir una
% ventana gráfica para evitar continuar "pintando" sobre una
% creada anteriormente
figure
% Tu código aquí
d = deg2rad(1);
x = (-2*pi:d:8*pi);
% y es la funcion seno
y = sin(x);
```

```
plot(x,y,'m');
hold on
fun = @(x) sin(x);
x0 = 3;
f = fminsearch(fun,x0)
```

```
f = 4.7124
```

```
g = sin(f);
plot(f,g,'*k')
hold off
```




```
help quad
```

```
quad - (Not recommended) Numerically evaluate integral, adaptive Simpson quadrature
   This MATLAB function approximates the integral of function fun from a to
   b using recursive adaptive Simpson quadrature:

   Syntax
     q = quad(fun,a,b)
     q = quad(fun,a,b,tol)
     q = quad(fun,a,b,tol,trace)
     [q,fcnEvals] = quad(___)

   Input Arguments
     fun - Integrand
       function handle
     a - Integration limits (as separate arguments)
       scalars
     b - Integration limits (as separate arguments)
       scalars
     tol - Absolute error tolerance
```

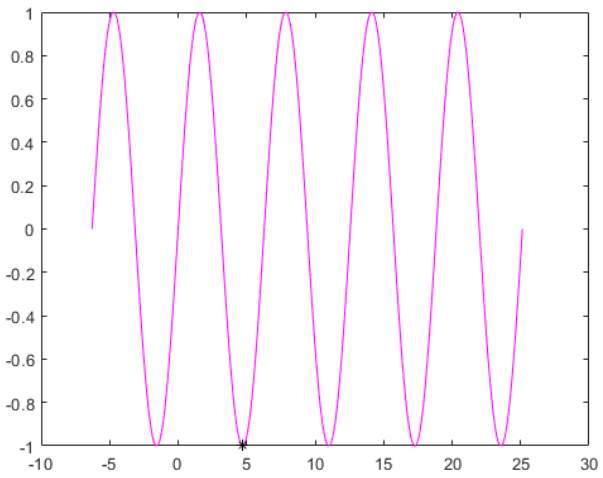
```
      [] or 1e-6 (default) | scalar
    trace - Toggle for diagnostic information
      nonzero scalar

  Output Arguments
    q - Value of integral
      scalar
    fcnEvals - Number of function evaluations
      scalar

  Examples
    Compute Definite Integral

  See also quad2d, quadgk, trapz, integral, integral2, integral3

  Introduced in MATLAB before R2006a
  Documentation for quad
```

```
integral = quad(fun,1,2)
```

```
integral = 0.9564
```

```

```

Resultado esperado:




```
integral = 0.9564
```

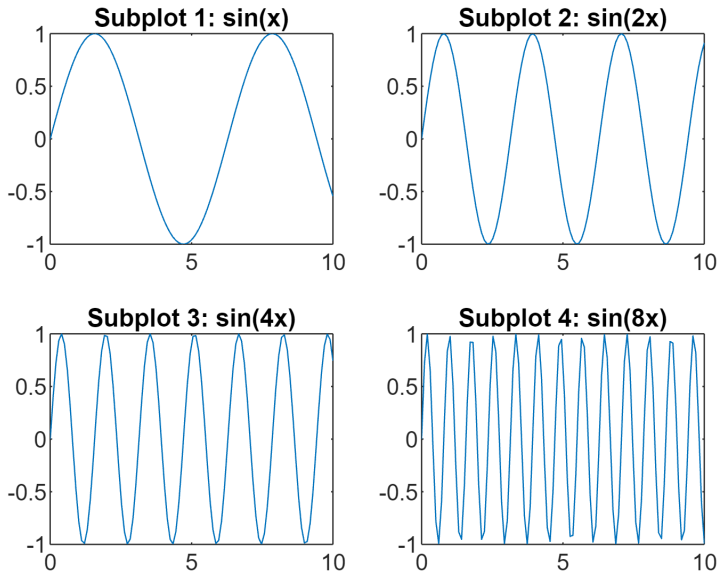
2. Subgráficas

Pero, ¿qué pasa si necesito mostrar gráficas distintas simultáneamente? No hay problema, MATLAB ha pensado en todo y mediante el comando **`subplot`** podemos dividir la ventana gráfica activa en una serie de particiones horizontales y verticales, activando una de ellas. Si no hay ventana gráfica activa, la crea.

Veamos el siguiente código de ejemplo que divide la ventana gráfica en 2 filas con 2 gráficos cada una (el tercer argumento de **`subplot`** indica el índice de la subgráfica que se va a activar para *pintar* en ella):

```
subplot(2,2,1)
```

```
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)
title('Subplot 1: sin(x)')

subplot(2,2,2)
y2 = sin(2*x);
plot(x,y2)
title('Subplot 2: sin(2x)')

subplot(2,2,3)
y3 = sin(4*x);
plot(x,y3)
title('Subplot 3: sin(4x)')

subplot(2,2,4)
y4 = sin(8*x);
plot(x,y4)
title('Subplot 4: sin(8x)')
```

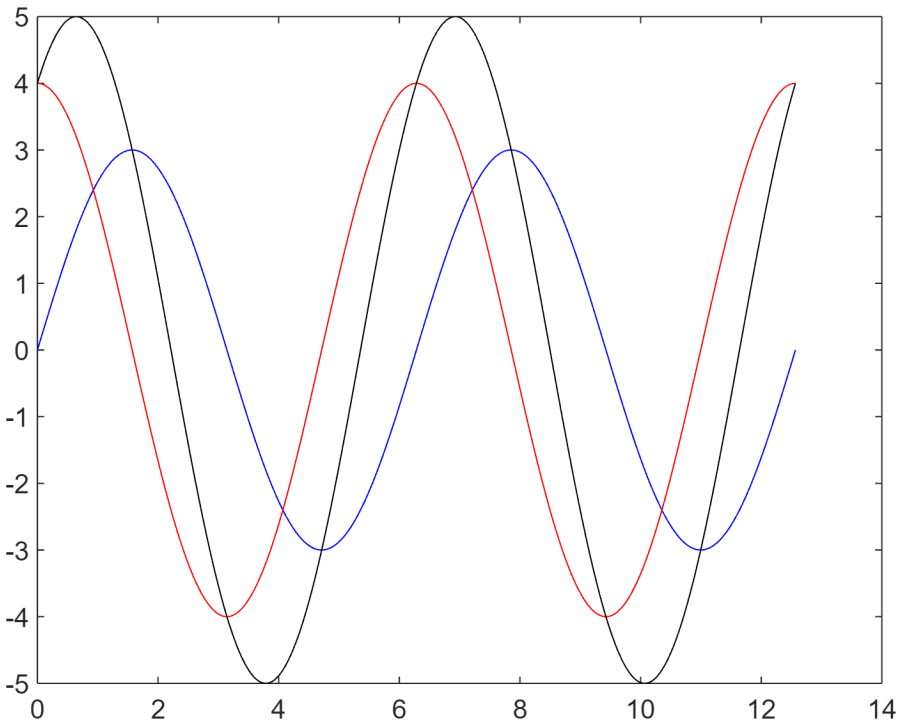


Tarea 2: Obtener una representación gráfica de las funciones $y_1 = 3\sin(\theta)$, $y_2 = 4\cos(\theta)$ y la suma de ambas $y_3 = y_1 + y_2$ para los valores de θ comprendidos en el intervalo $[0, 4\Pi]$, con una resolución de un grado, en cada una de las formas siguientes:

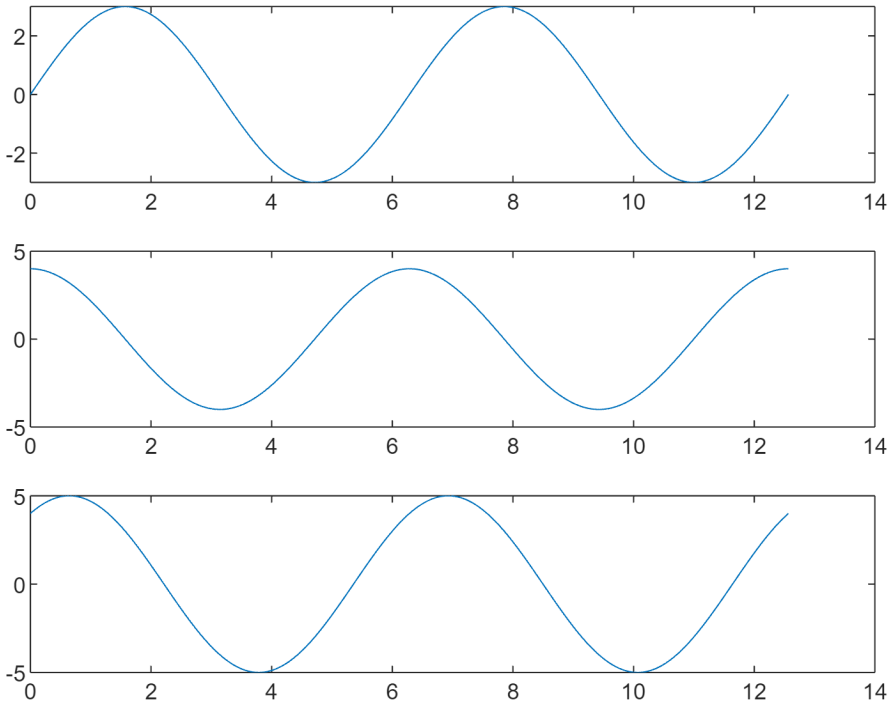
- a) En una única ventana gráfica utilizando los mismos ejes y colores diferentes.
- b) En una única ventana gráfica (figura) donde podrán visualizarse las tres gráficas alineadas verticalmente.

Pista: Entre los comandos a utilizar necesitarás recurrir a **`figure`**.para ir creando las nuevas figuras.

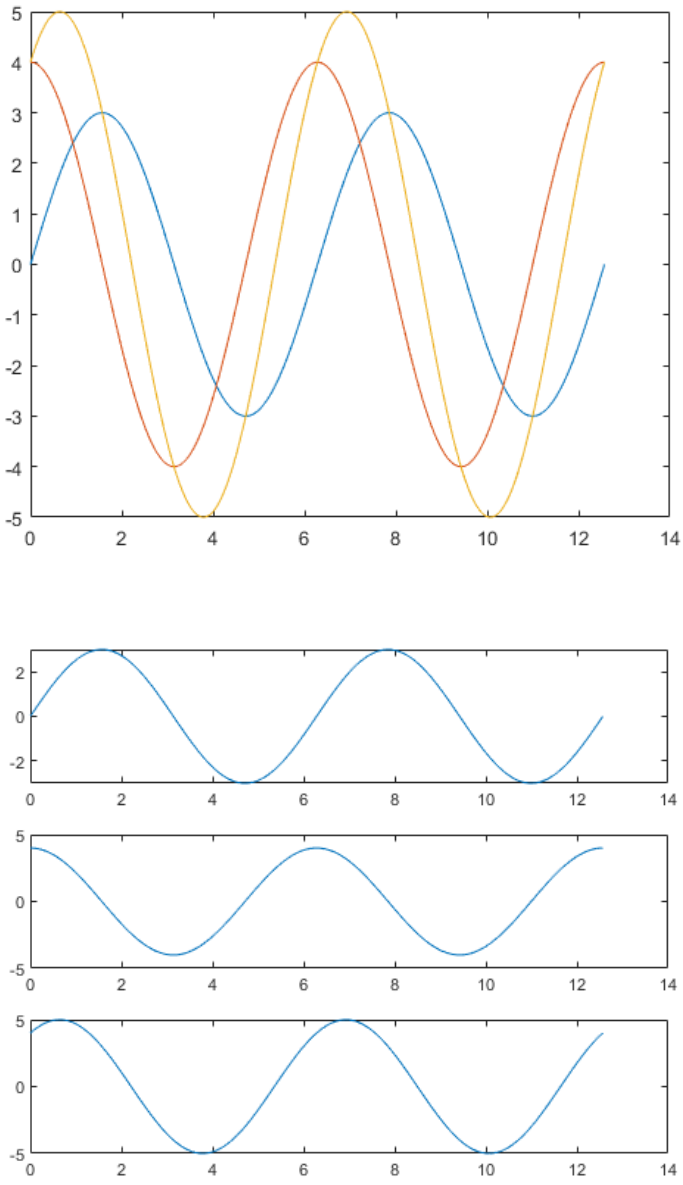
```
% Tu código aquí

% a)
figure
d = deg2rad(1);
x = (0:d:4*pi);
y1 = 3*sin(x);
y2 = 4*cos(x);
y = y1 + y2;
plot(x,y1,"b",x,y2,"r",x,y,"k")
```




```
%b)
figure
subplot(3,1,1)
d = deg2rad(1);
x = (0:d:4*pi);
y1 = 3*sin(x);
plot(x,y1)
subplot(3,1,2)
y2 = 4*cos(x);
plot(x,y2)
subplot(3,1,3)
y = y1 + y2;
plot(x,y)
```



Resultado esperado:



3. Control sobre los ejes

MATLAB provee de una serie de funciones, equivalentes a **`plot`**, que permiten especificar la escala de los ejes. Por ejmplo:

- **`loglog`**: tanto el eje de abcisas como el de ordenadas se representan en escala logarítmica decimal.
- **`semilogx`**: el eje de abcisas se representa en escala logarítmica decimal.
- **`semilogy`**: el eje de ordenadas se representa en escala logarítmica decimal.

Además, también es posible poner un título a la gráfica con el comando **`title`** (habrá que usar **`sgtitle`** si se le quiere poner título a una figura que contiene **`subfigure`**), así como un texto junto al eje de abcisas (**`xlabel`**) o de ordenadas (**`ylabel`**).

Para finalizar con los ejes, también está disponible la función **`axis`**. Vamos a ver en qué consiste:

```
help axis
```

```
  axis - Set axis limits and aspect ratios
     This MATLAB function specifies the limits for the current axes.

     Syntax
       axis(limits)

       axis style
       axis mode
       axis ydirection
       axis visibility

       lim = axis
       [m,v,d] = axis('state')

       ___ = axis(ax,___)

     Input Arguments
       limits - Axis limits
         four-element vector | six-element vector | eight-element vector
       mode - Manual, automatic, or semiautomatic selection of axis limits
         manual | auto | 'auto x' | 'auto y' | 'auto z' | 'auto xy' |
         'auto xz' | 'auto yz'
       style - Axis limits and scaling
         tight | padded | fill | equal | image | square | vis3d | normal
       ydirection - y-axis direction
         xy (default) | ij
       visibility - Axes lines and background visibility
         on (default) | off
       ax - Target axes
         one or more axes

     Output Arguments
       lim - Current limit values
         four-element vector | six-element vector

     Examples
       Set Axis Limits
       Add Padding Around Stairstep Plot
       Use Semiautomatic Axis Limits
       Set Axis Limits for Multiple Axes
       Display Plot Without Axes Background
       Use Tight Axis Limits and Return Values
       Change Direction of Coordinate System
       Retain Current Axis Limits When Adding New Plots

     See also xlim, ylim, zlim, tiledlayout, nexttile, title, grid, Axes,
       PolarAxes

     Introduced in MATLAB before R2006a
     Documentation for axis
     Other uses of axis
```

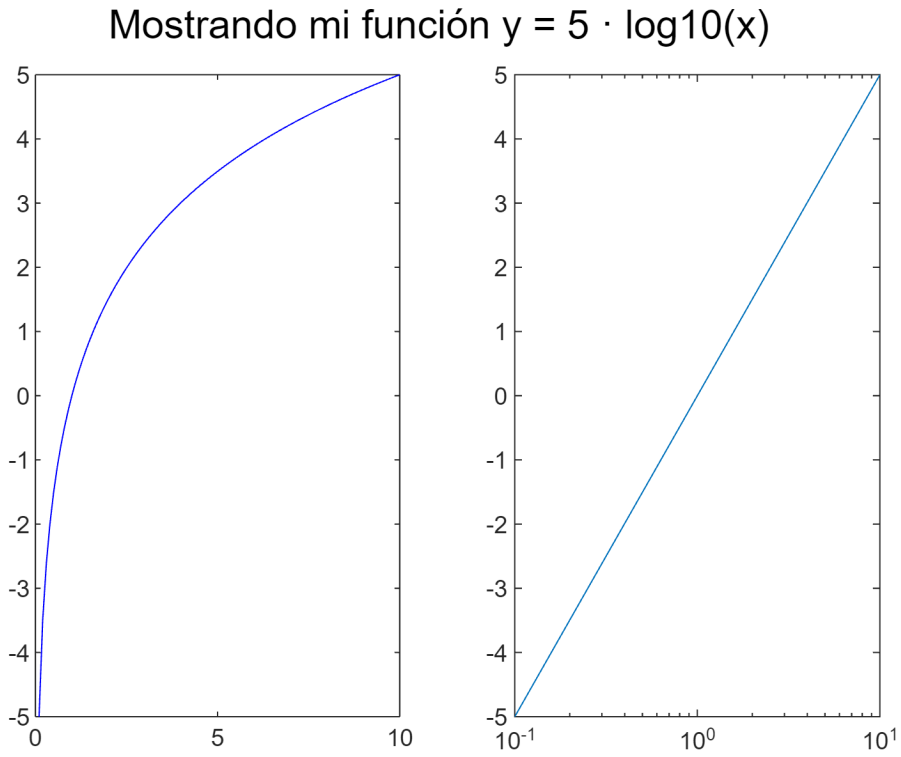
Tarea 3: Se desea representar gráficamente la función $y = 5\log_{10}x$ para los valores de x comprendidos en el intervalo $[0.1, 10]$, con una resolución de una décima. Obtener en una única ventana gráfica, utilizando ejes diferentes alineados verticalmente, una representación de dicha función en cada una de las formas siguientes:

- a) Utilizando escalas lineales en ambos ejes.

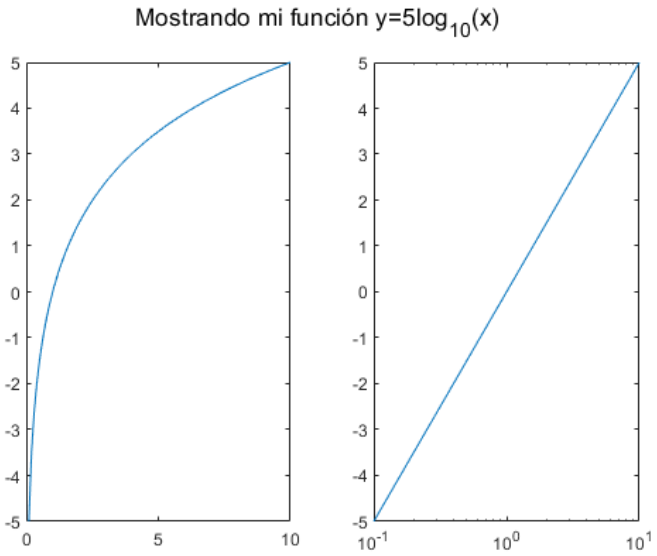
- b) Utilizando escala logarítmica decimal para el eje de abcisas y escala lineal para el eje de ordenadas.

Además, ponle el título que desees.

```
% Tu código aquí
figure
sgtitle('Mostrando mi función y = 5 · log10(x)')
subplot(1,2,1)
x = 0.1:0.1:10;
fun = @(x) 5*log10(x);
y = 5*log10(x);
plot(x,y,'b')
subplot(1,2,2)
semilogx(x,y)
```



Resultado esperado:


Mostrando mi función y=5log10(x)
5
4
3
2
1
0
-1
-2
-3
-4
-5
0
5
10
10-1
100
101

Introducción a MATLAB

Scripts, bucles e instrucciones condicionales

Table of Contents

Hasta el momento hemos venido trabajando con **scripts vivos** que según MATLAB pueden definirse como:

"Archivos de programa que contienen código, salidas y texto con formato, y conviven en un solo entorno interactivo conocido como Live Editor. En los scripts en vivo, puede escribir código y ver las salidas y las gráficas generadas junto con el código que las produjo. Añada texto con formato, imágenes, hipervínculos y ecuaciones para crear una narrativa interactiva que puede compartir con otros."

Esta manera de trabajar con MATLAB es bastante reciente, sin embargo, existe un enfoque más tradicional, que aún sigue siendo el adecuado cuando se necesita definir funciones complejas que alargarían en exceso los scripts vivos: los **archivos de comando** o **scripts** a secas.

En este cuaderno también vamos a ver unos commandos que nos permiten controlar el flujo de nuestros scripts: los **bucles** y las **instrucciones condicionales**.

1. Archivos de Comandos

Los archivos de comandos (archivos *script*) almacenan secuencias de sentencias. Se utilizan para llevar a cabo operaciones que involucran a múltiples sentencias, con objeto de evitar que cuando exista un error en alguna de ellas haya que reescribir y reejecutar todas las que le siguen. Asimismo, este tipo de archivos se puede utilizar simplemente para almacenar una secuencia de sentencias para la que se prevé una futura reutilización. Otro importante uso de los *script* es el de la definición de funciones. Por ejemplo, los comandos que hemos visto están definidos en este tipo de ficheros.

La siguiente imagen muestra un ejemplo de un script que para un cierto vector v calcula su media y su desviación típica:

```
EDITOR  PUBLISH  VIEW
1 -    v = [1 3 4 5 6];
2 -    n = length( x );
3 -    media = sum( x ) / n;
4 -    desvtip= sqrt( sum( (x - media).^2 ) / n );
5      % salida formateada
6 -    fprintf( 'Media : %f \n', media );
7 -    fprintf( 'Desviación Típica : %f \n', desvtip );
```

UTF-8 | script | Ln 7 | Col

Mientras que la siguiente ilustra una función (comando) denominada **`desv`** que permite ejecutar el mismo código con distintos vectores de entrada:

```
EDITOR  PUBLISH  VIEW
1   function [ media, desvtip ] = desv( x )
2   %desv Calcula la media y desviación típica de un vector
3   %   [media, desvtip] = desv(v) calcula la media y la desviación típica
4   %   vector v.
5   %
6   %   Ejemplos:
7   %   v = [1 3 4 5 6];
8   %   [media, desvtip] = desv(v);
9   %   Media : 3.800000
10  %   Desviación Típica : 1.720465
11
12 -    n = length( x );
13 -    media = sum( x ) / n;
14 -    desvtip= sqrt( sum( (x - media).^2 ) / n );
15      % salida formateada
16 -    fprintf( 'Media : %f \n', media );
17 -    fprintf( 'Desviación Típica : %f \n', desvtip );
18 -  end
19
```

UTF-8 | desv | Ln 19 | Col

Los archivos de comandos tienen una extensión '.m' y pueden crearse en el editor que incorpora el entorno de MATLAB ('Home/New script' o 'Ctrl+N'). Las líneas de comentarios ubicadas por delante de la primera línea de programa de un archivo de comandos se muestran en la ventana de comandos de Matlab cuando en ésta se ejecuta el comando **`help`** seguido del nombre del archivo de comandos (excluyendo su extensión).

Un archivo de comandos puede ejecutarse de distintas formas:

- o bien desde la ventana de comandos (escribiendo una sentencia con el nombre del fichero sin extensión),
- directamente desde el editor del entorno de Matlab,
- o en una celda de código de un script vivo.

Para que dicha ejecución sea viable, el directorio en el que el que está almacenado el archivo de comandos considerado debe estar incluido en la lista de caminos de búsqueda de MATLAB ('Home/Set path'), el cual incluye el directorio de trabajo actual. Básicamente con esto nos aseguramos de que MATLAB conoce su existencia.

2. Bucles

2.1 For

El bucle por excelencia. Una instrucción **`for`** se repite un número especifico de veces, realizando un seguimiento de cada iteración con una variable en aumento. Su sintaxis es la siguiente:

```
for index = values

statements

end
```

El siguiente código a ejecutar muesta un ejemplo en el que un bucle **`for`** recorre ciertas posiciones de un vector y suma sus elementos:

```
x = [1 2 1 2 1 2];
suma = 0;
for i=3:5
    suma = suma + x(i);
    disp(suma)
end
```

```
    1

    3

    4
```

Un bucle puede terminar bien sea por haber alcanzado el último valor de la variable en aumento a comprobar, o bien por la ejecución de la instrucción **`break`**. Por ejemplo:

```
suma = 0;
for i=3:5
    suma = suma + x(i);
    disp(suma)
    if i == 4
        break
    end
```

```
end
```

```
     1

     3
```

Por último, también podemos saltar a la siguiente iteración del bucle con el comando **`continue`**. Ejemplo:

```
suma = 0;
for i=3:5
    if i == 4
        continue
    end
    suma = suma + x(i);
    disp(suma)
end
```

```
     1

     2
```

Tarea 1: Define un vector fila de números aleatorios (generados a partir de una distribución de probablidad normal gaussiana) v con 10 elementos e implementa un bucle **`for`** que sume los que se encuentren en el rango de posiciones $[2, 7]$, mostrando el valor de la suma una vez concluido el bucle.

```
% Tu código aquí
v = randn(1,10)
```

```
v = 1×10
   -0.7342   -0.0308    0.2323    0.4264   -0.3728   -0.2365    2.0237   -2.2584 ···
```

```
sum = 0;
for i=1:10
    if i >= 2
        if  i <= 7
        sum = sum + v(1,i)
        end
    end
    disp(sum)
    disp(i)
end
```

```
     0
     1
sum = -0.0308
   -0.0308
     2
sum = 0.2015
    0.2015
     3
sum = 0.6279
    0.6279
     4
sum = 0.2551
    0.2551
```

```
     5
sum = 0.0187
    0.0187
     6
sum = 2.0423
    2.0423
     7
    2.0423
     8
    2.0423
     9
    2.0423
    10
```

2.2 While

La otra joya de la corona a nivel de bucle de control es el bucle **`while`**. Este bucle continúa realizando una serie de instrucciones mientras que la expressión que evalúa sea cierta. Su sintaxis es:

`while` *`expression`*

`statements`

`end`

Un ejemplo de su uso:

```
i = 3;
suma = 0;
while i < 6
    suma = suma + x(i);
    disp(suma)
    i = i+1;
end
```

De igual modo que con el bucle **`for`**, usando **`while`** también se puede salir de dicho bucle con el comando **`break`**, o saltar a la siguiente iteración con **`continue`**.

3. Instrucciones condicionales

3.1 Si yo tuviera... (if)

Volvemos a encontrarnos con una instrucción popular: **`if`**. Esta instrucción condicional nos permite ejecutar una serie de instrucciones cuando una cierta condición es verdadera. En MATLAB una expresión es verdadera cuando su resultado no está vacío y contiene solo elementos no nulos (numéricos reales o lógicos). De lo contrario, la expresión es falsa.

Su sintaxis es:

`if` *`expression`*

`statements`

`elseif` *`expression`*

statements

else

statements

end

Los bloques **elseif** y **else** son opcionales, y permiten indicar que ocurre si no se cumple la primera condición condicional pero sí una segunda (o un número de n de condiciones subsecuentes) y que ejecutar en cualquier otro caso, respectivamente.

Veamos un ejemplo del uso de **if** dentro de un bucle **for** que sólo suma los elementos en una posición par de un vector:

```
x = [1,2,1,2,1,2,1,2];
suma = 0;
n = length(x);
for i = 1:n
    if mod(i,2)
        suma = suma + x(i);
        disp(suma)
    end
end
```

Tarea 2: Usar el bucle **for** y la instrucción **if** para, una vez definido un vector v que contenga valores tanto negativos como positivos (definidor por ti), sume sólo los elementos positivos del vector. Muestra el resultado de la suma al salir del bucle.

```
% Tu código aquí
v = [-3 -2 -1 0 1  5 10 20]
```

```
v = 1×8
    -3    -2    -1     0     1     5    10    20
```

```
tam = 8;
sum = 0;

for i= 1:tam
    if v(i)>0
        sum = sum + v(i)
    end
end
```

```
sum = 1
sum = 6
sum = 16
sum = 36
```

```
disp(sum)
```

```
    36
```

```
disp(i)
```

```
     8
```

3.2 El caso (switch)

Y concluímos nuestro repaso a las instrucciones condicionales de MATLAB con la sentencia switch, la cual permite evaluar una condición y ejecutar uno de varios grupos de instrucciones. Su sintaxis es:

```
switch switch_expression
case case_expression
statements
case case_expression
statements
...
otherwise
statements
end
```

El siguiente código a ejecutar muestra un ejemplo de su uso, donde se nos va a pedir que introduzcamos un número y se va a ejecutar un caso distinto en función de este:

```
n = input('Enter a number: ');

switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case 1
        disp('positive one')
    otherwise
        disp('other value')
end
```