Programación en MATLAB

Esta guía pretende servir para empezar a dar los primeros pasos en la programación. Para aprender, el lenguaje que se emplee es - hasta cierto punto- irrelevante: lo más importante son los conceptos. Una vez conocido un lenguaje, pasar a otro no suele resultar difícil. Como MATLAB es un lenguaje simple, resulta idóneo para los comienzos.
Un programa consiste en una serie de instrucciones que se ejecutan secuencialmente (una detrás de otra). Por lo tanto, es requisito previo conocer esas instrucciones, y cómo escribirlas correctamente (*sintaxis*).

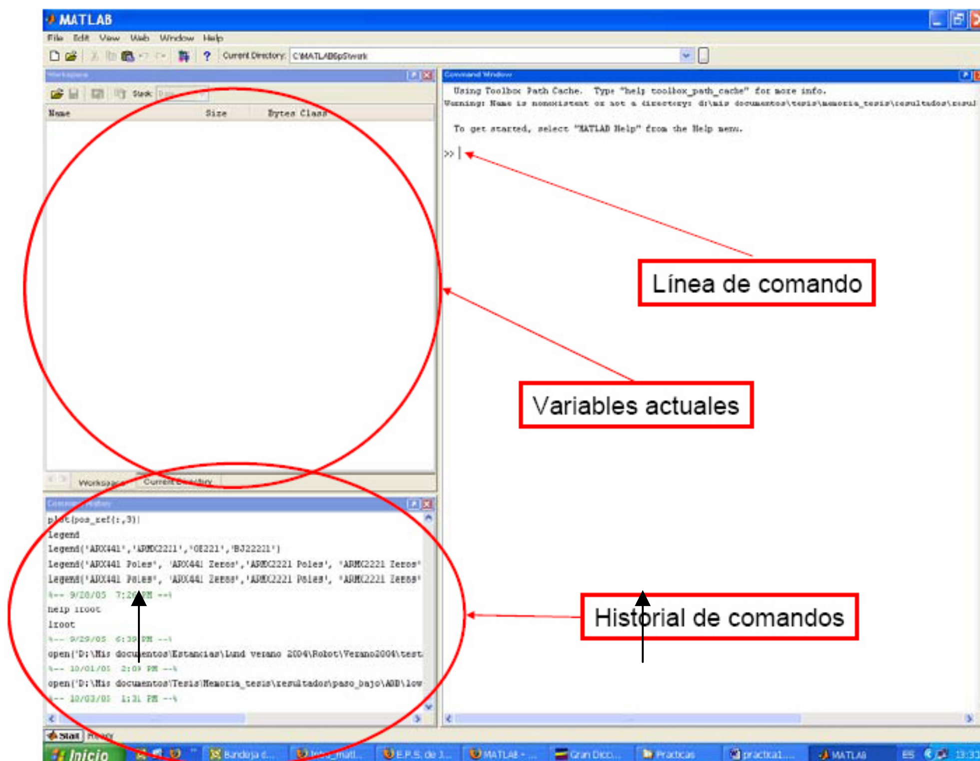
1. ARCHIVOS SCRIPTS Y FUNCIONES EN MATLAB

Los archivos con extensión (.m) son archivos de texto sin formato (archivos ASCII) que constituye el centro de la programación en MATLAB. Estos archivos se crean y modifican con un editor de texto cualquiera. En el caso de MATLAB ejecutado en una PC bajo Windows, lo más sencillo es utilizar su propio editor de MATLAB.

Para Ingresar al editor de textos de MATLAB:

File (Barra de Menú)
•**New** ⇨ **M-File**

Tenemos la siguiente ventana del editor de textos: (Se recomienda tener las dos ventanas activas tanto la **ventana de comandos** como la **ventana editor de textos**).



ventana editor de texto ***ventana de comandos***

Es importante observar en el directorio que estamos trabajando[1] pues es donde se guardará el archivo.

Una vez editado el archivo, para ejecutarlo basta teclear el nombre del archivo:
>> prueba

Tipos de Archivos *.m

- Archivos Scripts
- Archivos Funciones.

ARCHIVOS SCRIPT

Esencialmente, un archivo script es equivalente a escribir secuencialmente las instrucciones del archivo. El contenido de un archivo script se ejecuta tecleando simplemente su nombre en la ventana de comando ó tecleando F5.

Ejercicio 1: Edita un archivo script con el nombre *operacionesvarias.m* en tu directorio de trabajo y escribe[2]:

```
A=[1 2 3; 4 5 6; 7 8 9];
disp('dimensiones de la matriz')
disp(size(A))
disp('det=')
disp(det(A))
```

Ejecuta el archivo (basta teclear el nombre, es decir, operacionesvarias). Comprueba como ahora uno puede ejecutar todas esas instrucciones cambiando las veces que desee la matriz A.

Ejercicio 2: Edita un archivo script con el nombre *tanplot.m* en tu directorio de trabajo y escribe[3]:

```
theta = linspace(1.6,4.6);
tandata = tan(theta);
plot(theta,tandata);
xlabel('\theta     (radians)');
ylabel('tan(\theta)');
grid on;
axis([min(theta) max(theta) -5 5]);
```

FUNCIONES

MATLAB permite crear funciones nuevas en forma de archivos con extensión ***.m**

Características de funciones

---

[1] por defecto es C:\MATLAB6p5\work

[2] disp es una abreviatura de display. Despliega un texto o un valor numérico por pantalla.

[3] disp es una abreviatura de display. Despliega un texto o un valor numérico por pantalla.

- El nombre de la función y del archivo debe ser el mismo
- Esta se ejecuta desde la ventana de comandos
- La primera línea debe ser

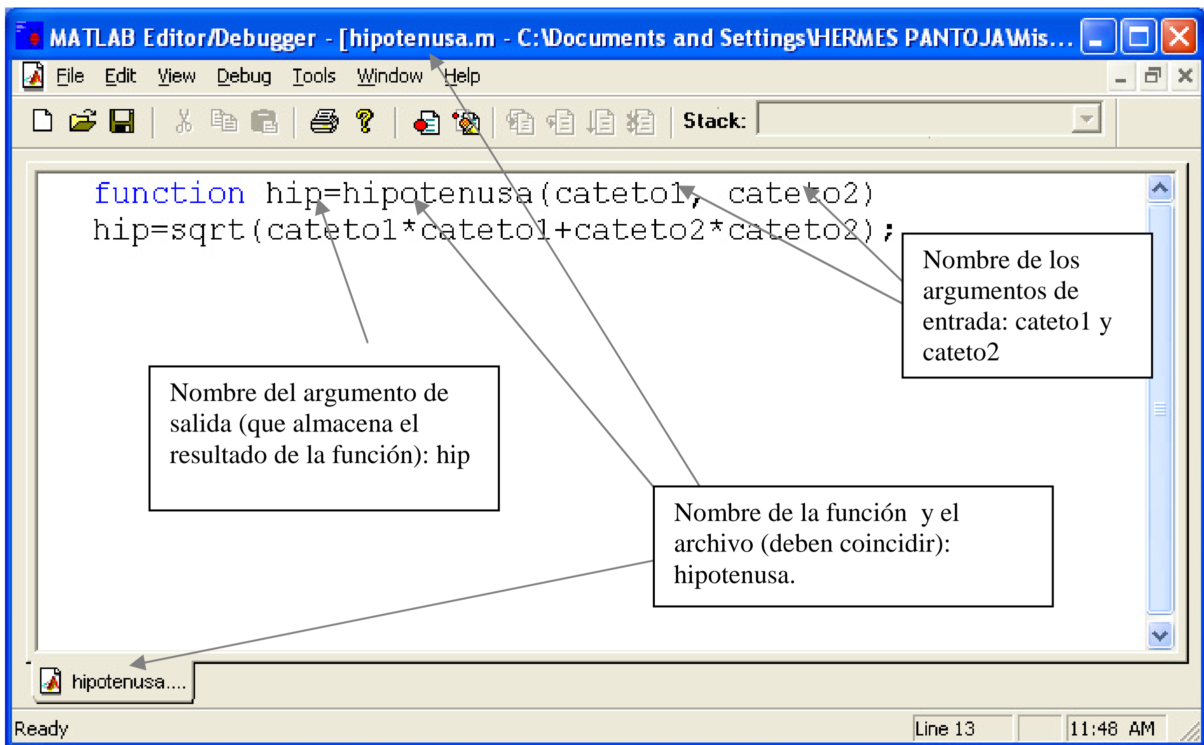
function [argumentos de salida] = nombre_función(argumentos de entrada)

seguida de las instrucciones necesarias. Cuando hay más de un argumento de salida, éstos deben ir entre corchetes y separados por comas. Es conveniente utilizar las primeras líneas del archivo como comentario (iniciando con '%'). Así, dicha definición será visible mediante la instrucción **help *nombre_función***

Ejemplo: Crear una función que calcule el valor de la hipotenusa de un triángulo rectángulo dado sus dos catetos.

Solución:
Las instrucciones a programar:

```
function hip = hipotenusa(cateto1, cateto2)
    hip=sqrt(cateto1*cateto1+cateto2*cateto2);
```



Tras guardar el archivo, damos valores a dos variables y ejecutamos la nueva función creada desde la **ventana de comando**:

```
>> x = 3, y = 4
>> hipotenusa(x,y)
```

Mostrando el resultado:

```
ans=
     5
```

Observamos que la solución de la función viene asignada a la variable *ans*.

Ahora para

```
>> x=5;  y = 12;  z = hipotenusa(x,y)
```

Mostrando el resultado:

```
z =
    13
```

En este caso, las variables *x* e *y* toman los valores indicados, pero no muestran por pantalla el resultado de la operación. Como resumen, se pueden separar varias instrucciones en una misma línea por una coma o un punto y coma, diferenciándose en que el segundo caso no muestra los resultados de las operaciones. De ahí que todas las instrucciones de una función acaben con punto y coma. Cuando una instrucción es demasiada larga para que entre en una sola línea, se puede acabar con tres puntos (". . .") indicando que continúa en la línea siguiente.

2. BUCLES Y ESTRUCTURAS DE DECISION

2.1 Bucles:

El comando for

Repetición de sentencias un número conocido de veces. La orden básica para implementar un bucle es la instrucción **for**. Su sintaxis es

```
for i=mi:ms:mf
.....
end
```

La instrucción anterior ejecuta el conjunto de instrucciones comprendidas entre el **for** y el **end** con **i** tomando los valores del vector **mi:ms:mf**

mi: Valor Inicial
ms: Incremento
mf: Valor Final

Ejemplo:

```
>> for j=1:3;  disp(j);  end
     1

     2

     3
```

En general, si **v** es un vector, la orden

```
for i=v
.....
end
```

Procede a ejecutarse el bucle con valores consecutivos v(1),v(2),…,v(n)

Ejemplo:

```
>> v=[2 5 3 1]; for j=v; disp(j); end
```

Asociados a **for** encontraremos las órdenes **break** y **continue.** La primera fuerza la salida inmediata del bucle mientras que la segunda reinicializa la iteración con el nuevo valor del índice **i.**

Ejercicio 1: La matriz de Hilbert de orden ***n*** viene dado

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \cdots & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \cdots & \cdots & \frac{1}{n+1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \frac{1}{n} & & & \frac{1}{2n-2} & \frac{1}{2n-1} \end{bmatrix}$$

Construye la matriz anterior mediante un archivo **script** y el uso de dos **for** anidados

El comando while

Repetición indefinida de sentencias. La orden básica para implementar un bucle es la instrucción **while**. Su sintaxis es

```
while expresión
sentencias;
end
```

Las "sentencias" se repiten mientras "expresión" sea no nulo. La "expresión" generalmente es una comparación tipo expresión operador_relacional expresión dónde el operador relacional puede ser alguno de los siguientes (ver help relop):

<	Menor
<=	Menor o igual
>	Mayor
>=	Mayor o igual
==	Igual
~=	Diferente

Ejemplo:

```
x=0;
n=0;
disp('Ingrese números menores que 100')
disp('Para finalizar ingrese un número mayor que 100')
while x < 100,
x=input('Ingrese un número: ');
n=n+1;
```

```
y(n,1)=x;
end
disp(['Se ingresaron ', int2str(n), ' números'])
y
```

2.2 Estructuras de decisión

Veamos un caso simple. Construyamos el diagrama de flujo de un programa que escoja un número al azar del 0 al 9 y nos pida que lo adivinemos. Una vez escogido el número, el programa debe informar si hemos acertado o no. El diagrama de flujo correspondiente es

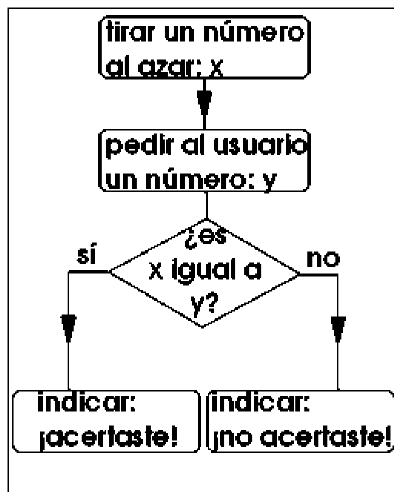
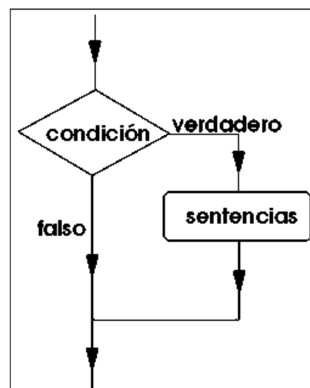


Diagrama de flujo

2.2.1 El comando if

Veamos a continuación la sintaxis de la función **if**:




```
if expresión
  sentencias;
end
```

- `if` simple:

```
if (x<-1 | x>1)
    x=1;
    disp('valor absoluto de x mayor que 1')
end
```

Si “expresión” es no nula, se ejecutan “sentencias”.

```
if expresión_1
 sentencias_A;
 else
sentencias_B;
end
```

- `if` compuesto

```
if (x<0 & x>-10)
   f=x^2;
else
   f=sin(x^2)
end
```

Si “expresión_1” es no nulo, se ejecutan las “sentencias_A”, si no, se ejecutan las “sentencias_B”.

```
if expresión_1
   sentencias_A;
elseif expresión_2
  sentencias_B;
else
  sentencias_C;
end
```

Si “expresión_1” es no nulo, se ejecutan las “sentencias_A”, si no, si “expresión_2” es no nulo, se ejecutan las “sentencias_B”, si no (ninguno de los anteriores), se ejecutan las “sentencias_C”.

- `if` anidados

```
if (x<0)
   f=x.^2;
   disp('x menor que cero')
elseif (x<sqrt(pi))
   f=sin(x^2);
   disp('x en [0,sqrt(pi))')
elseif (x<2*sqrt(pi))
   f=(x-sqrt(pi))^2;
   disp('x en [sqrt(pi),2*sqrt(pi) )')
else
   f=pi*cos(x^2);
   disp('x en [2*pi,infty)')
end
```

Nota. Con la instrucción **switch** se puede implementar una estructura de decisión que es esencialmente equivalente a un **if** anidado.

2.2.2 Sentencia switch

Otra posibilidad de bifurcación múltiple la ofrece la construcción **switch**. La sintaxis es:

```
switch variable
case valor1,
   sentencias A
case valor2,
   sentencias B
case ...
   ...
otherwise,
   sentencia C
end
```

La bifurcación **switch** opera de la siguiente manera. Al llegar a la expresión switch *variable*, si *variable* tiene el valor *valor1* se ejecutan las *sentencias A*; si *variable* toma el valor *valor2*, las *sentencias B*; y así sucesivamente. Es importante notar que la variable sólo debe tomar unos pocos valores: *valor1*, *valor2*, etc. Si ninguno es igual a *variable* se ejecutan las sentencias correspondientes a **switch.** Para que el programa se ramifique en unas pocas ramas. No tiene sentido intentar una ramificación **switch** con una variable que pueda tomar un número infinito de valores.

Ejemplo:

```
n=4;
switch(n)
case 1,
   resp='el valor de n=1'
case 2,
   resp='el valor de n=2'
case 3,
   resp='el valor de n=3'
case 4,
   resp='el valor de n=4'
otherwise,
   resp='Ninguna de las anteriores'
end
```

EJEMPLOS

Ejemplo 1: Calcular la suma de los n primeros términos de la sucesión 1, 2x, $3x^2$, $4x^3$, ..

```
n=input('¿Cuántos términos quieres sumar?  ');
x=input('Dame el valor del numero x  ');
suma=1;
for i=2:n
   suma=suma+i*x^(i-1);
end
disp('El valor pedido es')
disp(suma)
```

Ejemplo 2: Decidir si un número natural es primo.

```
n=input('Número natural que deseas saber si es primo ');
i=2;
primo=1;
while i<=sqrt(n)
 if rem(n,i)==0   % Resto de dividir n entre i
      primo=0;
   break
   end
   i=i+1;
end
if primo
   disp('El número dado es primo.')
else
   disp('El número dado no es primo.')
   disp('De hecho, es divisible por:')
   disp(i)
end
```

Ejemplo 3: Escribir un número natural en una base dada (menor que diez).

```
n=input('Dame el número que quieres cambiar de base ');
base=input('¿En qué base quieres expresarlo? ');
i=1;
while n>0
   c(i)=rem(n,base);
   n=fix(n/base);   % Parte entera de n/base
   i=i+1;
end
disp('La expresión en la base dada es:')
i=i-1;
disp(c(i:-1:1))
```

Ejemplo 4: Implementar una función que calcule, mediante el algoritmo de Euclides, el máximo común divisor de dos números naturales

```
function m=euclides(a,b)
if a<b
  c=b; b=a; a=c;
end
while b>0
  c=rem(a,b);
 a=b; b=c;
end
m=a;
```

Funciones inline

Existe una forma alternativa de definir una función en Matlab en la línea de comandos.
Al teclear:

```
»f=inline('exp(-x)*sin(x)','x');
```

Definimos la función $f(x) = e^{-x} \sin(x)$. La evaluación es ahora tan sencilla como:

```
»f(2)

ans =
0.1231
```

A efectos prácticos, la forma anterior es equivalente a editar una función con ese nombre y a escribir un simple código. Obviamente, la definición queda en memoria, y una vez cerrada la sesión de MATLAB (esto es, al salir del programa) la función se pierde. Esta es pues una importante desventaja frente la funciones definidas a través de M-files
Sin embargo puede resultar útil para tareas sencillas.
Nada nos impide definir funciones con más argumentos,

```
»g=inline('x*cos(y)-y*cos(x)','x','y');
```

Los últimos argumentos del comando inline definen las variables de la función. Si no se especifican, MATLAB trata de reconocer cuales pueden ser las variables:

```
»g=inline('x^2*cos(y)-y*cos(x)*z');
»g
g =
Inline function:
g(x,y,z) = x^2*cos(y)-y*cos(x)*z
```

Suele ser suficiente, aunque a veces expresar todas las variables explícitamente puede aclarar y facilitar la lectura, además de especificar el orden de las variables en la definición de la función.

Nótese que las funciones anteriores no están vectorizadas, es decir, al aplicarla sobre un vector o matriz no actúa elemento a elemento. Para ello, debería escribirse

```
»g = inline('x.^2.*cos(y)-y.*cos(x).*z');
»g([0 pi 0],[pi 0 0], [0 0 pi]); % la función esta ahora vectorizada

ans =
0 9.8696 0
```

Otra forma, más sencilla, es introducir la función de la forma habitual y pedir luego a MATLAB que la prepare para su aplicación a vectores. El comando apropiado para esta tarea es ***vectorize***:

```
»g = inline('x^2*cos(y)-y*cos(x)*z') % función normal
g =
Inline function:
```

g(x,y,z) = x^2*cos(y)-y*cos(x)*z
»g= vectorize(g)
g =
Inline function:
g(x,y,z) = x.^2.*cos(y)-y.*cos(x).*z

Observa como tras la aplicación de este comando, MATLAB redefine la función añadiendo "." en los sitios adecuados.

Para dibujar graficas de funciones definidas por trozos, necesitamos utilizar lo que vamos a denominar índices o variables lógicas.
Veamos un ejemplo.
Creamos un vector con los números del 1 al 7

```
>> x=1:7
x =
   1    2    3    4    5    6    7
```

Y ahora escribimos

```
>> x>4
ans =
   0    0    0    0    1    1    1
```

Observamos que donde no se cumple la condición, aparece 0 y donde se cumple, aparece 1

Así, por ejemplo, sobre el mismo x de antes, si escribimos

```
>> (2<x)&(x<=6)
ans =

   0    0    1    1    1    1    0
```

Ahora supongamos que queremos representar la función:

$$f(x)=\begin{cases} x^2 & si \quad x<0 \\ 1 & si \quad 0\le x<1 \\ -x+2 & si \quad 1\le x \end{cases}$$

Generamos una tabla de valores en el dominio en el que queramos dibujar la función

```
>>x=linspace(-2,3,3000);
```

Primero usaremos la programación no vectorizada (clásica)

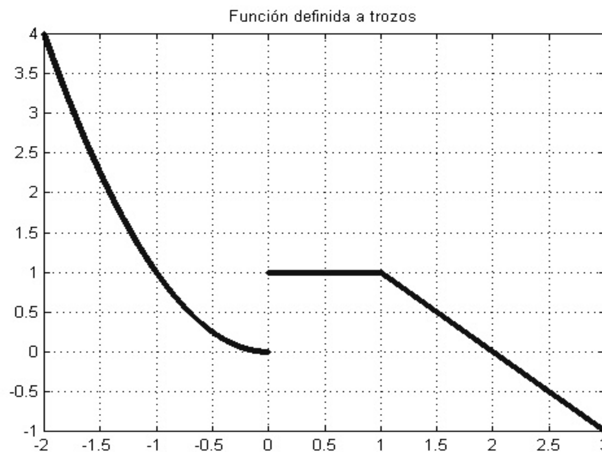
```
function y=ej_ftrozo(x)
y=[];
long=length(x);
for i=1:long
    if (x(i)<0);z=x(i)^2;
    elseif x(i)<1;z=1;
    else;z=-x(i)+2;end
    y=[y z];
end
```

Y ahora usaremos la ***programación vectorizada*** .Definimos la función, multiplicando cada trozo por el índice lógico que describa el lugar en el que queremos dibujarlo,

```
>>y=(x.^2).*(x<0)+1.*((0<=x)&(x<1))+(-x+2).*(1<=x);
```

Y luego la dibujamos. Resulta conveniente hacerlo con puntos, asteriscos o cruces porque, de otra forma, no aparecerán las discontinuidades.

```
>>plot(x,y,'.'),grid on,title('Función definida a trozos')
```



Otra versión se obtiene reemplazando el for por la función find.

```
function y=ej_ftrozo1(x)
y=zeros(size(x));
i=find(x<0);y(i)=x(i).^2;
i=find((x>=0) & (x<1));y(i)=1;
i=find(x>=1);y(i)=2-x(i);
```

Ejercicios

1. Crear function x = complejo(z) que obtiene el módulo y argumento de un complejo.
   Entrada: valor complejo z=a+i b
   Salida: vector x de dos componentes con el módulo y argumento de z
   Paso 1: definir el vector x con 2 componentes nulas
   Paso 2: calcular x1 = |z|
   Paso 3: calcular x2 = argumento(z)
   Comprobar los resultados obtenidos con las funciones de MATLAB que realizan lo mismo.

2. Programar la función

$$f(x)\begin{cases} -2x-1 & x \le -1 \\ x^2 & -1 < x < 1 \\ 2x-1 & x \ge 1 \end{cases}$$

   Comprobar su funcionamiento para diferentes valores y representarla.

3. Programar la función que calcula las raíces de una ecuación de segundo grado según el algoritmo siguiente:
   Entrada: coeficientes a, b y c de $ax^2 + bx + c = 0$

   Paso 1: definir el vector x con 2 componentes de valor -b/(2a)
   Paso 2: calcular d = b^2-4ac
   Paso 3: SI d=0, finalizar el programa
   Paso 4: SINO SI d>0, hacer d=sqrt(d)/ (2a)
   Paso 5: hacer $x_1 = x_1 + d$; $x_2 = x_2 - d$; finalizar el programa
   Paso 6: SINO; hacer d=sqrt(-d)/ (2a)
   Paso 7: hacer $x_1 = x_1 + i*d$; $x_2 = x_2 - i*d$; finalizar el programa

   Con estos datos, la primera fila del archivo debe ser: function x = ecua2gr(a,b,c)
   Ejecutarla con diferentes valores de a, b y c, comprobando todos los casos posibles.

4. Crear una función que obtenga el valor de e mediante el desarrollo:
   $e \approx \sum_{k=1}^{N} \frac{1}{k!}$ Como criterio de parada se tomará $\frac{1}{k!} < tol$, donde *tol* es el parámetro de entrada. Como resultados se debe obtener s (aproximación de *e*) y n (número de términos utilizados).

5. Utilizando *inline* dibujar la función

   $$f(x) = \frac{x-2}{x^2+1}$$

   en [-5,5], poniendo etiquetas a los ejes.