

# Tema 0. Introducción a Matlab

Métodos Numéricos y  
Simulación.

Segundo de Grado en Física.

# La pantalla de Matlab

**Current directory:** muestra la carpeta del disco duro en la que estamos trabajando.

The screenshot shows the MATLAB R2012b interface. The top toolbar includes tabs for HOME, PLOTS, and APPS. The HOME tab contains various icons for file operations (New Script, New, Open, Compare, Find Files), workspace management (Import Data, Save Workspace, Clear Workspace), code execution (Analyze Code, Run and Time, Clear Commands), and environment settings (Simulink Library, Layout, Parallel, Preferences, Set Path, Help, Community, Request Support). The current directory path is displayed as D:\docencia\dases\materiales recurrentes\MNS\Tema\_0\_Introduccion\_MATLAB. The interface is divided into several panes: Current Folder (left), Command Window (center), Workspace (right), and Command History (bottom right). Red arrows point from text boxes to these panes: one to the Command Window, one to the Workspace, one to the Current Folder, and one to the Command History.

**Command Window:** Aquí escribimos comandos “en línea” (para ser ejecutados de forma inmediata)

**Workspace:** muestra las variables que están almacenadas en memoria.

**Current Directory:** muestra el contenido de la carpeta en Current Directory.

**Command History:** Aquí se van guardando los comandos que escribimos en Command Window

# Tipos de datos

**En Matlab los datos se guardan dentro de variables.**

Las variables son de distinto tipo dependiendo del tipo de datos que contienen.

Cuando se crea una variable, aparece su nombre en **Workspace** junto con un **icono** que nos indica su tipo

Si picamos dos veces sobre el icono de una variable, se nos muestra su contenido.



numéricos



lógicos



alfanuméricos  
(caracteres y cadenas)



estructuras



arrays de celdas



referencia a función



# Variables numéricas

**Son datos numéricos aquellos que sólo contienen números**

• **Números enteros y reales**

```
>> k=1
```

→  $k = 1$

• **Vectores de números**

```
>> v=[3.5 4.4 1.3]
```

→  $v = (3.5 \ 4.4 \ 1.3)$

```
>> w=[3.5 ; 4.4 ; 1.3]
```

→  $w = \begin{pmatrix} 3.5 \\ 4.4 \\ 1.3 \end{pmatrix}$

• **Matrices de números**

```
>> A=[3.1 9.8 ; 5.4 7.7]
```

$A = \begin{pmatrix} 3.1 & 9.8 \\ 5.4 & 7.7 \end{pmatrix}$

Para acceder a los elementos de un array de números se usa (). Por ejemplo:  $A(1,2)=9.8$

# Operaciones matriciales

**Las operaciones  $+$   $-$   $*$   $^$  se interpretan como operaciones sobre números, vectores o matrices, dependiendo de las dimensiones de los arrays implicados.**

**Ejemplos:**

`k=3`

`w=[1 ;2 ;3]`

`A=magic(3)`

`v=[3 5 7]`

Producto de escalar por vector:

**$k*v$**

**$k*w$**

Suma y resta de vectores

**$v+w$**

**$v-w$**

Producto escalar de dos vectores:

**$v*w$**

Producto diádico

**$w*v$**

Producto de un vector por una matriz

**$v*A$**

**$A*w$**

Potencia de una matriz

**$A^2$**

Nota:  $^$  sólo puede aplicarse sobre matrices cuadradas

# Creación de matrices

**Matlab proporciona comandos para crear matrices especiales**

<b>A = eye(4)</b>	matriz identidad 4*4
<b>B = zeros(3,4)</b>	matriz de ceros de tres filas y 4 columnas
<b>C = zeros(3)</b>	matriz de ceros 3*3
<b>D = ones(4,5)</b>	matriz de unos de 4 filas y 5 columnas
<b>E = rand(2,5)</b>	matriz 2*5 con elementos aleatorios entre 0 y 1
<b>F = rand(2,5)</b>	
<b>x = linspace( 0, 10, 5)</b>	vector FILA con 5 valores equiespaciados entre 0 y 10
<b>mv = []</b>	se puede crear una matriz vacia

<b>whos mv</b>	Proporciona información sobre la matriz mv (nombre, dimensiones, espacio en memoria, tipo de elemento)
----------------	---

# Matriz traspuesta e inversa

**A'** traspuesta de la matriz A

Como la multiplicación de matrices no cumple la propiedad conmutativa hay dos formas de calcular la matriz inversa en Matlab:

**A\** inversa de la matriz A por la **izquierda**

$Y = M * X$

Ejemplo

**Y =**

79  
92  
21

**M =**

3 8 5  
4 9 5  
0 3 2



$X = M \backslash Y$

**x =**

8  
5  
3

**/A** inversa de la matriz A por la **derecha**.

$Y = X * M$

Ejemplo

**Y =**

96 66 55

**M =**

1 5 9  
3 5 4  
9 2 0



$X = Y / M$

**x =**

3 7 8

# Operaciones elemento a elemento

**Anteponiendo un punto . a los operadores + - \* / ^ , las operaciones se hacen elemento a elemento entre arrays del mismo tamaño.**

**Ejemplos:**

`z=[2 4 6]`

`w=[1 ;2 ;3]`

`v=[3 5 7]`

- Producto de los elementos de dos vectores

`v.*z`

`v.*w'`

Pero no..

`v.*w`

- Cuadrado de los elementos de una matriz `A.^2`

**Muchas funciones de Matlab operan sobre arrays elemento a elemento.**

**Ejemplo:**

```
>> sin([0.3 1.15 2.45])
```

```
ans =
```

```
0.2955
```

```
0.9128
```

```
0.6378
```



# Direccionamiento de matrices a partir de vectores

**Podemos extraer submatrices de una matriz usando la sintaxis:**

$$A([y_1, y_2, \dots, y_N], [x_1, x_2, \dots, x_M])$$

Índices de las  
filas a extraer

Índices de las  
columnas a extraer

Ejemplo:

```
>> A=magic(4)
```

A =

16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1


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


ans =


2	3
14	15


# El operador :

**El operador : se usa para referirse a series consecutivas de números (por defecto enteros si no se especifica un paso)**

**A(1:5,2)**  es un vector que contiene elementos A(1,2), A(2,2),..., A(5,2) de la matriz A

**A(:,2)**  es un vector columna que contiene todos los elementos de la segunda columna de la matriz A

**10:15**  es un vector que contiene los números enteros del 10 al 15

**10:0.5:15**  es un vector que contiene los números enteros y semienteros del 10 al 15

el uso del operador : nos evita escribir bucles en Matlab



# Variables alfanuméricas

**Son variables alfanuméricas aquellas que contienen letras y números tratados como letras.**

```
>> nombre1 = 'Miguel'
```

```
>> nombre2 = 'Angel'
```

Matlab trata las cadenas de caracteres igual que los arrays de números. P. ej. podemos concatenarlas tal y como un vector fila.

```
nombre = [nombre1 ' ' nombre2]
```

Algunos caracteres especiales se escriben usando un código

To Insert ...	Use ...	Carriage return	\r
Backspace	\b	Horizontal tab	\t
Form feed	\f	Backslash	\\
New line	\n	Percent character	%%



# Variables lógicas

**Una variable lógica es aquella que sólo puede tomar dos valores: verdadero o falso**

- **verdadero** se representa por **1** o **true**
- **falso** se representa por **0** o **false**

Ejemplos:

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

```
y =
```

```
1      0      0
```

```
>> y = [true false true]
```

```
y =
```

```
1      0      1
```

Aunque lo intentemos, Matlab no permite que tomen otros valores distintos de 0 ó 1. Si tratamos de darles como valor otro número que no sea 1 o 0, Matlab siempre escribe un 1.

```
>> y(2)=3
```

```
y =
```

```
1      1      1
```

# Operadores relacionales y lógicos

Los siguientes **operadores relacionales y lógicos** funcionan **elemento a elemento** entre **arrays del mismo tamaño**, excepto en el caso de que uno de ellos sea un escalar.

< menor que

<= menor o igual que

== igual a

> Mayor que

>= mayor o igual que

~= distinto de

& AND

| OR

~ NOT

Los siguientes **operadores lógicos** sólo pueden usarse entre escalares lógicos. Su evaluación se interrumpe en cuanto en algún punto se determina que la expresión es falsa.

&& AND

|| OR

Ejemplo

```
>> [1==2 2<3 5<3]&[3==2 5<7 7>6]
```

```
ans =
```

```
0      1      0
```

# Direccionamiento de arrays con arrays lógicos

**Un array lógico puede usarse para extraer elementos de otro array. Los elementos extraídos se corresponden con las posiciones donde el array lógico vale true.**

Ejemplo:

```
w=[2 4 6]
```

```
z = [1 2 3]
```

```
>> z(w<5)
```

```
ans =
```

```
1    2
```

La misma respuesta se obtiene con:

```
z([true true false])
```



# Estructuras. Creación

**Una estructura es un tipo especial de variable que alberga varios campos, cada uno con un tipo de datos.**

Cada campo puede contener datos de una naturaleza diferente.

La plantilla para una estructura puede crearse con el comando:

```
strArray = struct('field1',val1,'field2',val2, ...)
```

**field1, field2, ...:** son los nombres de los campos de la estructura.

**val1, val2, ...:** son los valores que se le da a los campos de la estructura.

**Ejemplo:**

```
elemento=struct('nombre',' ','simbolo',' ',  
'peso_atomico',0,'isotopos',zeros(1,10))
```



# Estructuras. Uso

A cada campo se accede escribiendo: Nombre-Estructura ■ Campo

## Ejemplo:

```
elemento.nombre = 'Hidrogeno'
```

```
elemento.simbolo='H'
```

```
elemento.peso_atomico =1.00794
```

```
elemento.isotopos = [1 2 3]
```

Normalmente **siempre se puede leer** un campo de una estructura de esta forma. Sin embargo, algunas estructuras que veremos durante el curso requieren **comandos especiales** para **crearlas** o **modificar** sus campos.





# Arrays de celdas

**Una celda es una variable en el que cada elemento puede ser de una naturaleza diferente.**

```
datos_quimica = {'Hidrogeno' 'H' 1.00794 [1 2 3] ;  
'Helio' 'He' 4.0026 [3 4]}
```

- Un array de celdas se crea listando sus elementos entre llaves {}
- Para acceder a los elementos de un array de celdas se usa ().

datos\_quimica(2,1) da como resultado una celda que contiene la cadena 'Helio'

- Para acceder **al contenido** de un elemento de un array de celdas se usa {}.

datos\_quimica{2,1} da como resultado una variable alfanumérica igual a la cadena 'Helio'

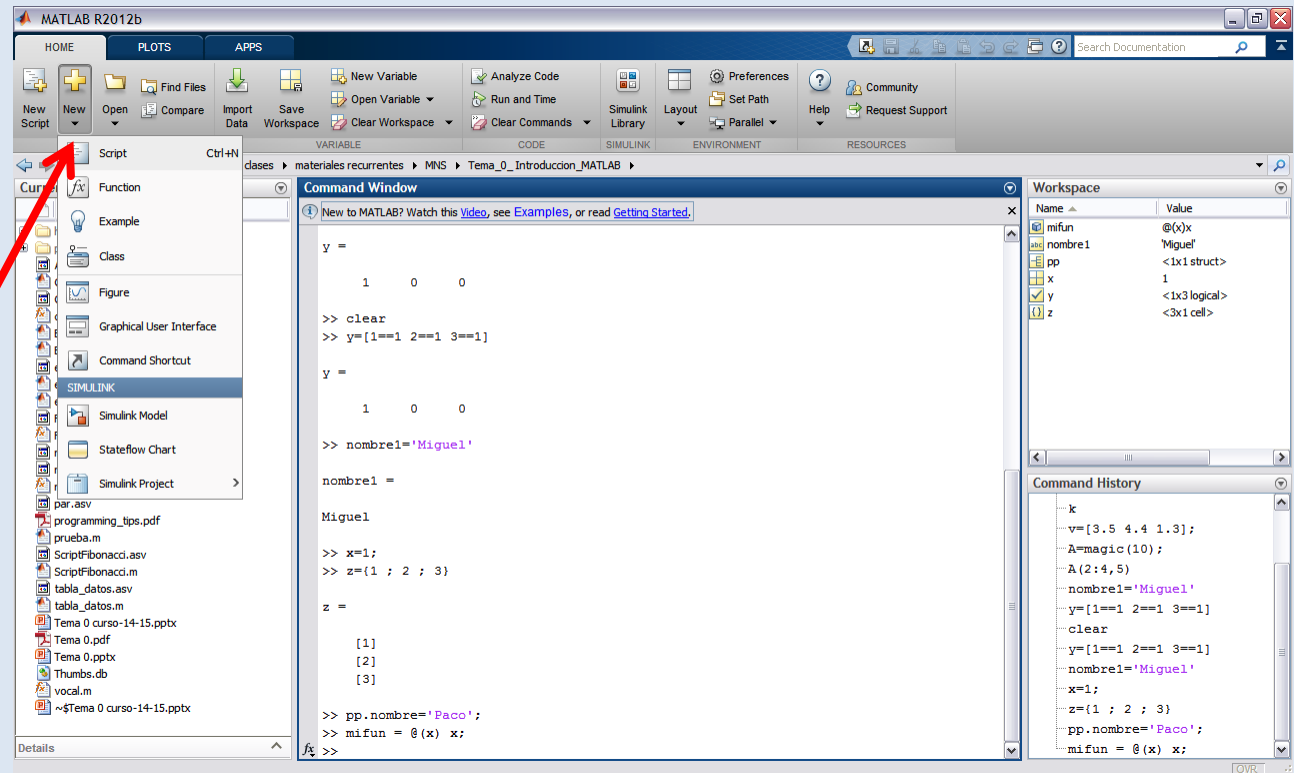
# Archivos .m

En ocasiones, se hace un mismo cálculo o tarea muchas veces.

Conviene entonces guardar la secuencia de comandos para que se ejecuten automáticamente sin que tener que volver a escribirlos.

**Matlab guarda secuencias de comandos en archivos de tipo M-file**

Para crear un archivo .m abrimos el editor de Matlab en la pestaña **Home**, icono **New**, opción **Script**



# script y funciones

Un archivo “.m” puede ser de dos tipos: un script o una función

Un **script** es un conjunto de líneas de código escritas tal y como lo haríamos por línea de comando guardadas en un archivo “.m”

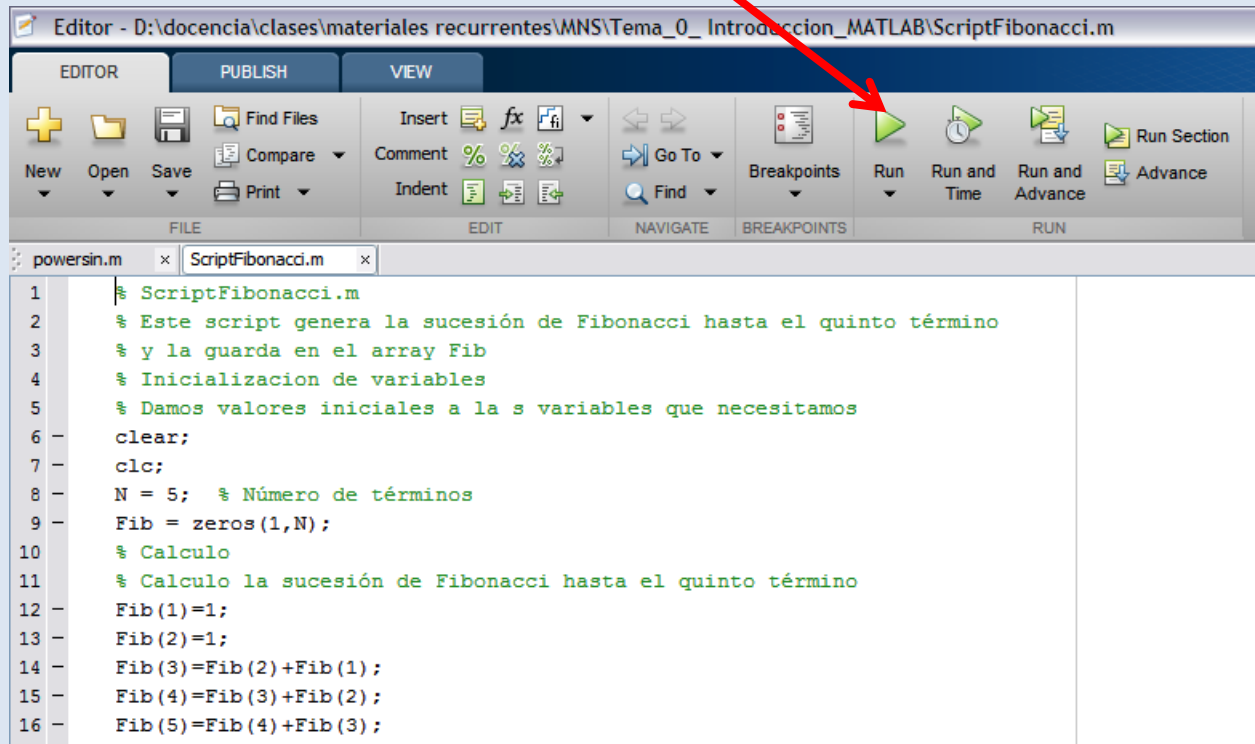
- No aceptan parámetros de entrada ni de salida.
- Operan con variables en el **Workspace**.
- Las variables que crean permanecen en memoria cuando terminan de ejecutarse (excepto si se borran con el comando delete).

Una **función** es un conjunto de líneas de código escritas en un archivo “.m” que comienzan por una línea en la que aparece la palabra clave **function** y terminan en una línea en la que aparece la palabra clave **end**.

- Necesitan parámetros de entrada y/o salida.
- Usan un espacio de memoria propio.
- Todas las variables que se crean en las líneas entre **function** y **end** se borran de memoria cuando se sale de la función.

# Ejemplo de script

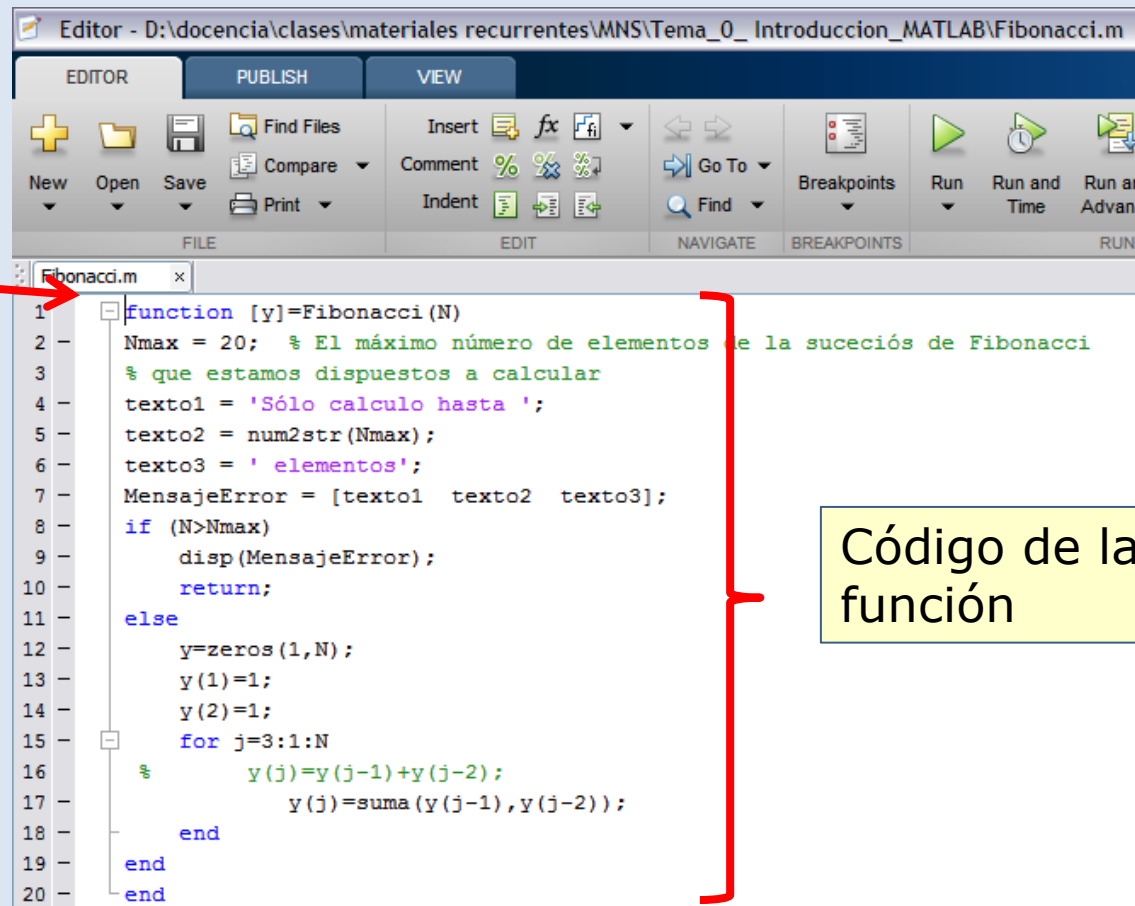
Un script se ejecuta escribiendo su nombre en **Command Window** y pulsando Enter o pulsando en el icono:



# Ejemplo de función

**Una función no puede ejecutarse de forma independiente. Necesita ser llamada por otro código (script, función o línea de comando) que dé valores a sus variables de entrada**

En la línea **function** se indican las variables de entrada y salida



```
Editor - D:\docencia\clases\materiales recurrentes\MNS\Tema_0_Introduccion_MATLAB\Fibonacci.m

EDITOR    PUBLISH    VIEW

+ New    Open    Save    Find Files    Insert    fx    fi
Compare    Comment    %    %    %
Print    Indent    Go To    Breakpoints    Run    Run and Time    Run as Advan

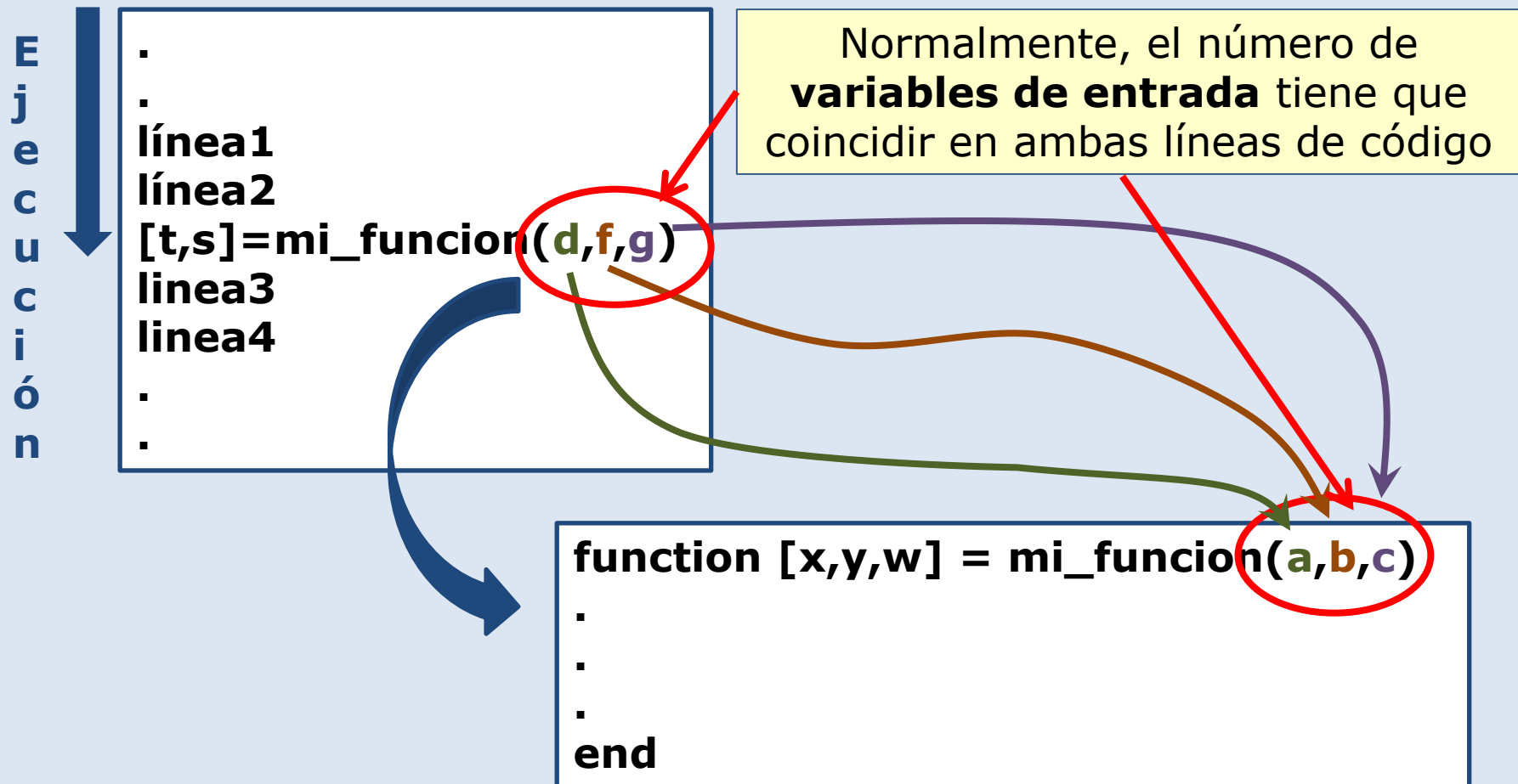
FILE    EDIT    NAVIGATE    BREAKPOINTS    RUN

Fibonacci.m x
1 function [y]=Fibonacci(N)
2 Nmax = 20; % El máximo número de elementos de la sucesión de Fibonacci
3 % que estamos dispuestos a calcular
4 texto1 = 'Sólo calculo hasta ';
5 texto2 = num2str(Nmax);
6 texto3 = ' elementos';
7 MensajeError = [texto1 texto2 texto3];
8 if (N>Nmax)
9     disp(MensajeError);
10    return;
11 else
12     y=zeros(1,N);
13     y(1)=1;
14     y(2)=1;
15     for j=3:1:N
16         % y(j)=y(j-1)+y(j-2);
17         y(j)=suma(y(j-1),y(j-2));
18     end
19 end
20 end
```

Código de la función

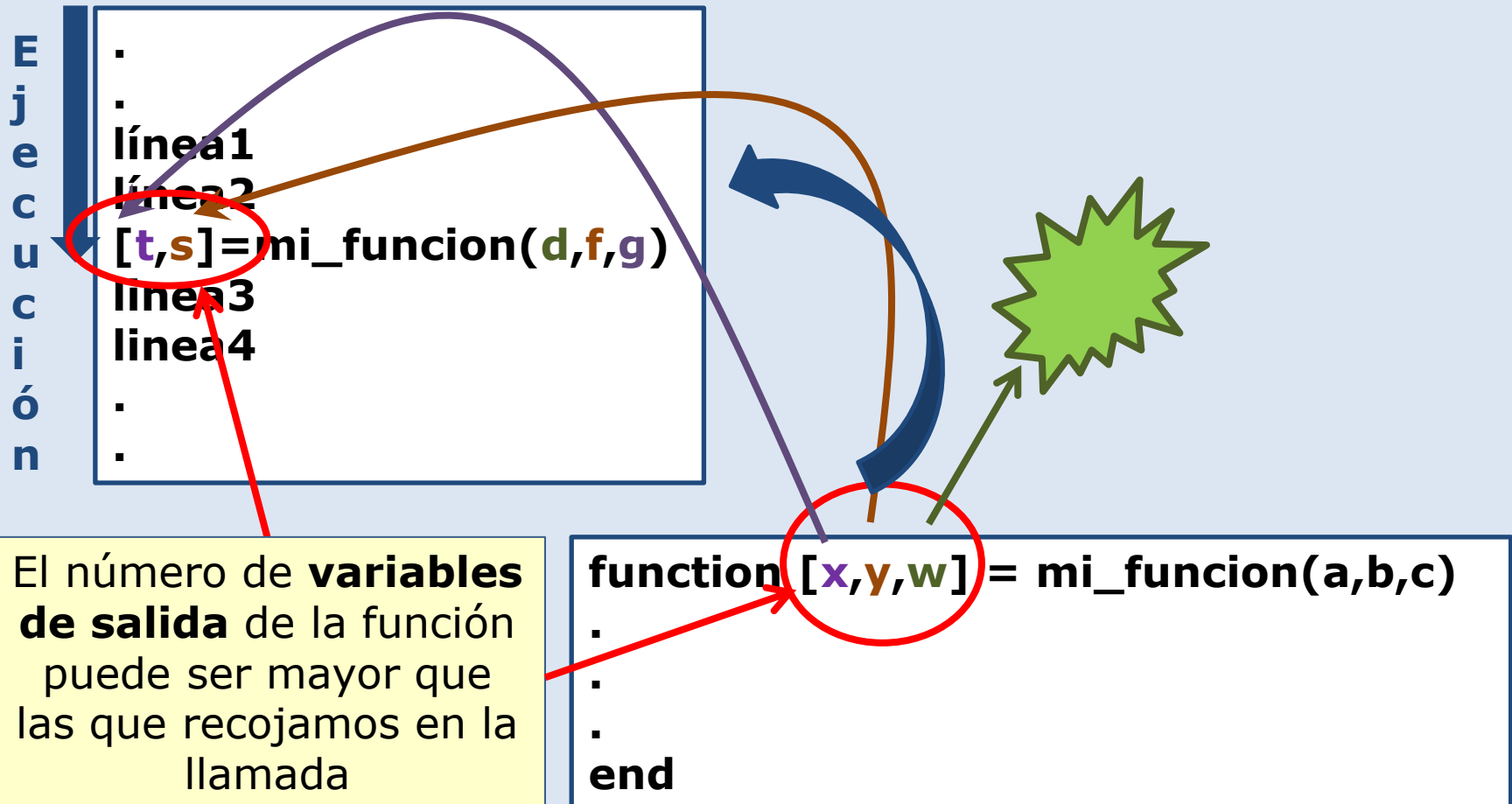
# Concepto de asignación: entrada

Cuando una parte del código llama a una función, el valor de las **variables de entrada** en el código se copia en las variables de entrada que hayamos escrito en la función **siguiendo el mismo orden**.



# Concepto de asignación: salida

Cuando la función acaba de ejecutarse, el valor de las **variables de salida** de la función se copia en las variables que hayamos escrito en el código **siguiendo el mismo orden** y se destruyen las variables que creó la función.



# Debugging (Búsqueda de errores en el código)

Es muy difícil escribir un programa sin cometer errores. Hay dos tipos genéricos de errores:

- **Errores de escritura.**

- Son aquellos que se cometen al escribir el programa: palabras clave mal escritas, variables que no se usan...
- Se pueden corregir **releyendo** el programa

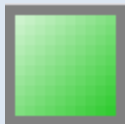
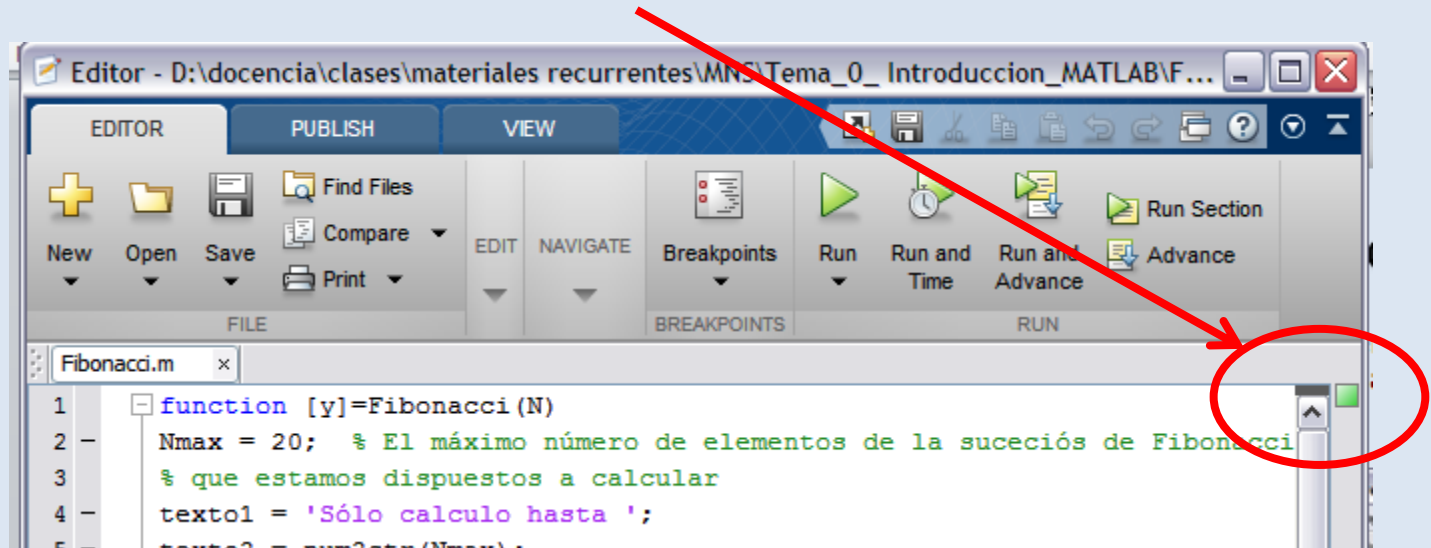
- **Errores de depuración**

- Son aquellos que se encuentran dentro de un programa sin erratas pero que tiene problemas de diseño
- Para corregirlos hay que prestar atención a qué es lo que ocurre **mientras se ejecuta el programa.**



# Errores de escritura

Matlab muestra los posibles errores de escritura en un cuadrado en la parte superior derecha de la ventana del editor y su posible localización con marcas en la columna derecha



sin errores de escritura



posible error de escritura



error de escritura: Matlab no dejará que se ejecute el código hasta que se corrija.

# Errores de depuración

Un **error de depuración** consiste en que algunas líneas de código que no hacen la función que esperábamos de ellas.

Para solucionar errores en tiempo de ejecución hemos de encontrar las líneas de código “problemáticas” y seguir la ejecución del programa **paso a paso** por ellas.

Se detectan de una de estas dos formas:

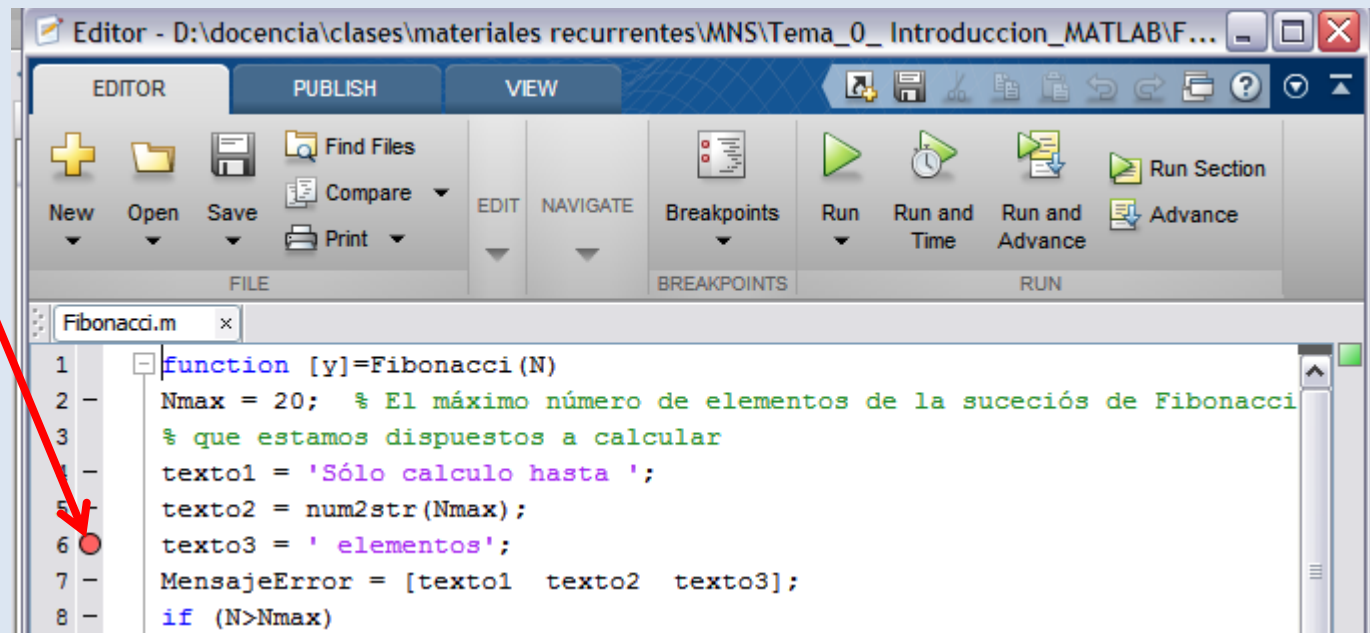
- 1)** El programa termina de forma prematura y se muestra un mensaje de error en **Command window**, que nos da una pista de por dónde buscar.
- 2)** El programa termina cuando debe, pero no hace el cometido que nosotros esperamos de él. Para encontrar el error, dependemos de nuestra experiencia e ingenio.

# *breakpoints*

Para correr el programa paso a paso primero hay que decir a Matlab que detenga la ejecución del programa en un punto.

Para indicar dónde se debe detener el programa se usan ***breakpoints***.

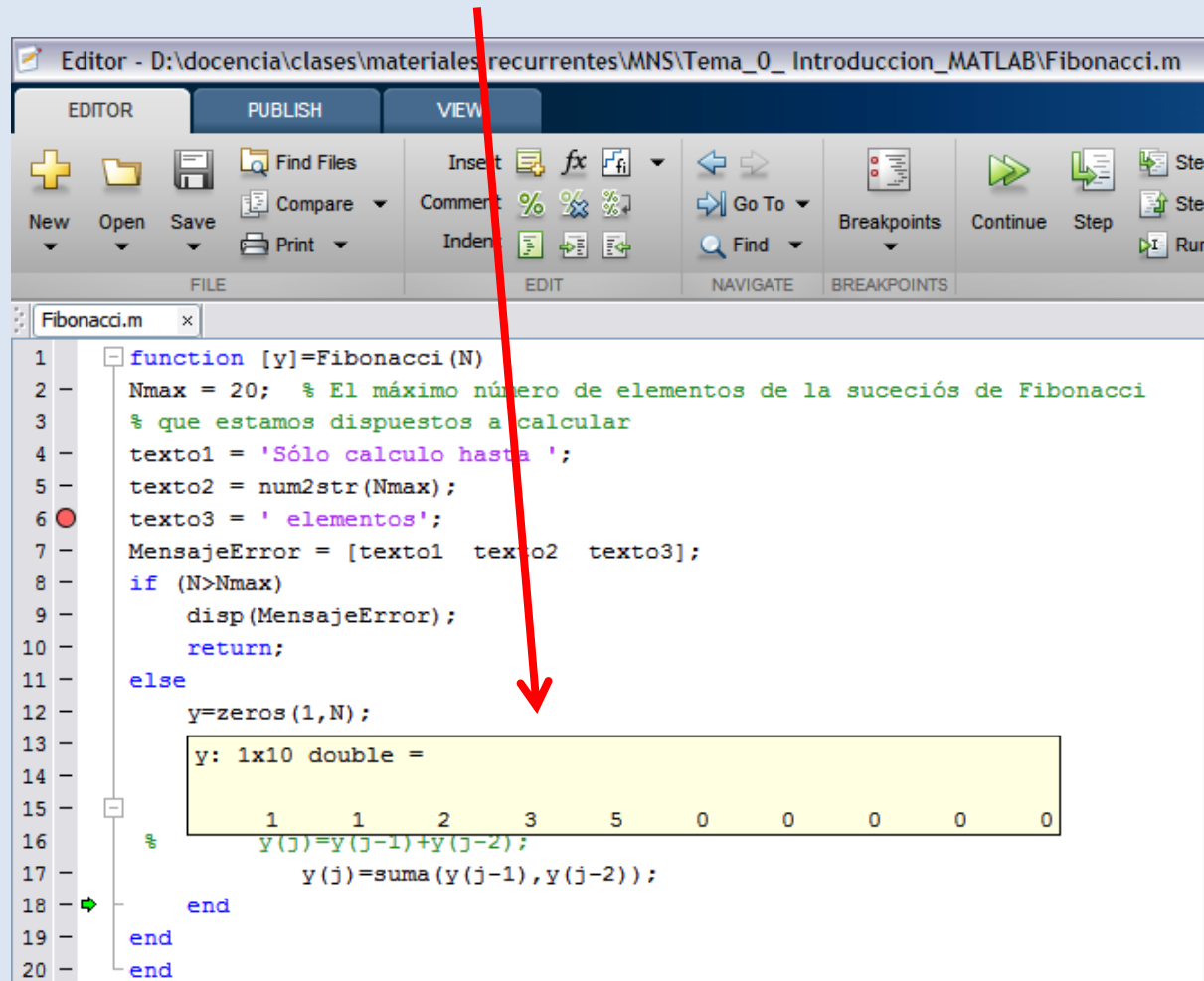
Para insertar un "breakpoint", se pica en la línea de código donde queremos detener el programa.



Es posible (y a veces conveniente) poner más de un ***breakpoint*** en un programa.

# Examinar variables (I)

Cuando un programa está detenido, podemos inspeccionar el contenido de las variables en memoria poniendo el ratón sobre ellas....



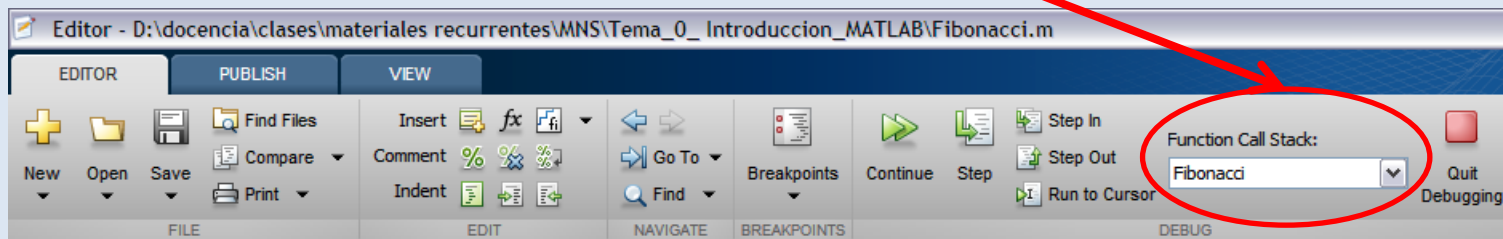
The image shows the MATLAB Editor window with the file `Fibonacci.m` open. The code is as follows:

```
1 function [y]=Fibonacci(N)
2     Nmax = 20; % El máximo número de elementos de la sucesión de Fibonacci
3     % que estamos dispuestos a calcular
4     texto1 = 'Sólo calculo hasta ';
5     texto2 = num2str(Nmax);
6     texto3 = ' elementos';
7     MensajeError = [texto1 texto2 texto3];
8     if (N>Nmax)
9         disp(MensajeError);
10        return;
11    else
12        y=zeros(1,N);
13        y: 1x10 double =
14            1     1     2     3     5     0     0     0     0     0
15        %
16        y(j)=y(j-1)+y(j-2);
17        y(j)=suma(y(j-1),y(j-2));
18    end
19 end
20
```

A red arrow points from the top of the window to a yellow tooltip that appears over the variable `y` on line 13. The tooltip displays the current value of `y` as a 1x10 double array: `1 1 2 3 5 0 0 0 0 0`.

# Examinar variables (II)

...o seleccionando el espacio de memoria (**Stack**) donde están las variables y luego yendo al **Workspace**



```
1 function [y]=Fibonacci(N)
2 Nmax = 20; % El máximo número de elementos de la sucesión de Fibonacci
3 % que estamos dispuestos a calcular
4 texto1 = 'Sólo calculo hasta ';
5 texto2 = num2str(Nmax);
6 texto3 = ' elementos';
7 MensajeError = [texto1 texto2 texto3];
8 if (N>Nmax)
9     disp(MensajeError);
```

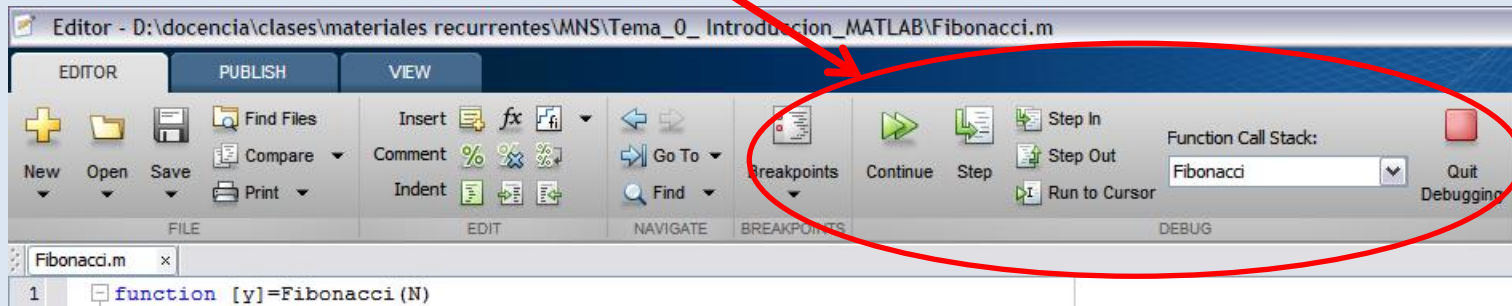
The image shows the MATLAB Workspace window. It contains a table with the following data:

Name	Value	Min	Max
N	10	10	10
Nmax	20	20	20
texto1	'Sólo calculo hasta '		
texto2	'20'		

Below the table is the 'Command History' section.

# Uso del *debugger*

**Un programa detenido puede hacerse avanzar de forma controlada usando los comandos del *debugger*.**



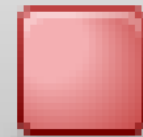
Continue

Correr hasta el final o el próximo *breakpoint*.



Step

Avanzar un paso en el programa.



Quit

Interrumpir la ejecución.



Step In

Entrar en una función.



Step Out

Salir de una función.



Run to Cursor

Correr el programa hasta la posición del cursor de texto.

# Contexto de una variable

Hemos visto que las variables creadas dentro de una función:

- 1) Sólo están en memoria mientras la función está ejecutándose.
- 2) Están en una parte de la memoria separada de las de cualquier otro código que se esté ejecutando

Se llama **contexto** (stack) de una variable al espacio de memoria que ocupa junto con otras variables. Sólo es posible relacionar entre sí directamente variables del mismo contexto.

Esto implica que:

- Las variables de una función pueden tener el mismo nombre que otras variables que aparezcan en otras partes de un programa y no por ello tener el mismo valor.
- El único intercambio de información con una función tiene lugar a través de sus parámetros de entrada y salida.

# Compilación separada

- Una función bien diseñada se debe poder utilizar desde cualquier otra función o script que escribamos en un futuro.

Para ello, Matlab debe saber dónde está la función.

Para hacer una función accesible a cualquier otra función o script, hay que escribirla en su propio archivo .m que debe tener el mismo nombre que la función.

Hay que tener en cuenta que Matlab sólo busca funciones en el directorio de trabajo y en la lista de directorios del **Path**.

- Si escribimos varias funciones en un único archivo “.m”, todas las funciones de las segunda en adelante sólo son accesibles para las funciones de ese mismo archivo.

Matlab considera todas las funciones siguientes a la primera como sub-funciones de ella y no deja que otros programas accedan a ellas.



# Sentencias de control

El flujo normal de ejecución de un programa es secuencial. Las sentencias de control se usan para alterar el flujo de ejecución.

**Ejecución condicional:** if, switch

Permiten hacer bifurcaciones en el flujo de ejecución haciendo que ciertas líneas se ejecuten o no.

**Bucles:** for, while.

Permiten repetir la ejecución de determinadas líneas.

**Finalizadores de ejecución:** continue, break, return

Se usan para salir de un bucle o de una función antes de que acabe su secuencia normal de ejecución.

# Ejecución condicional: if

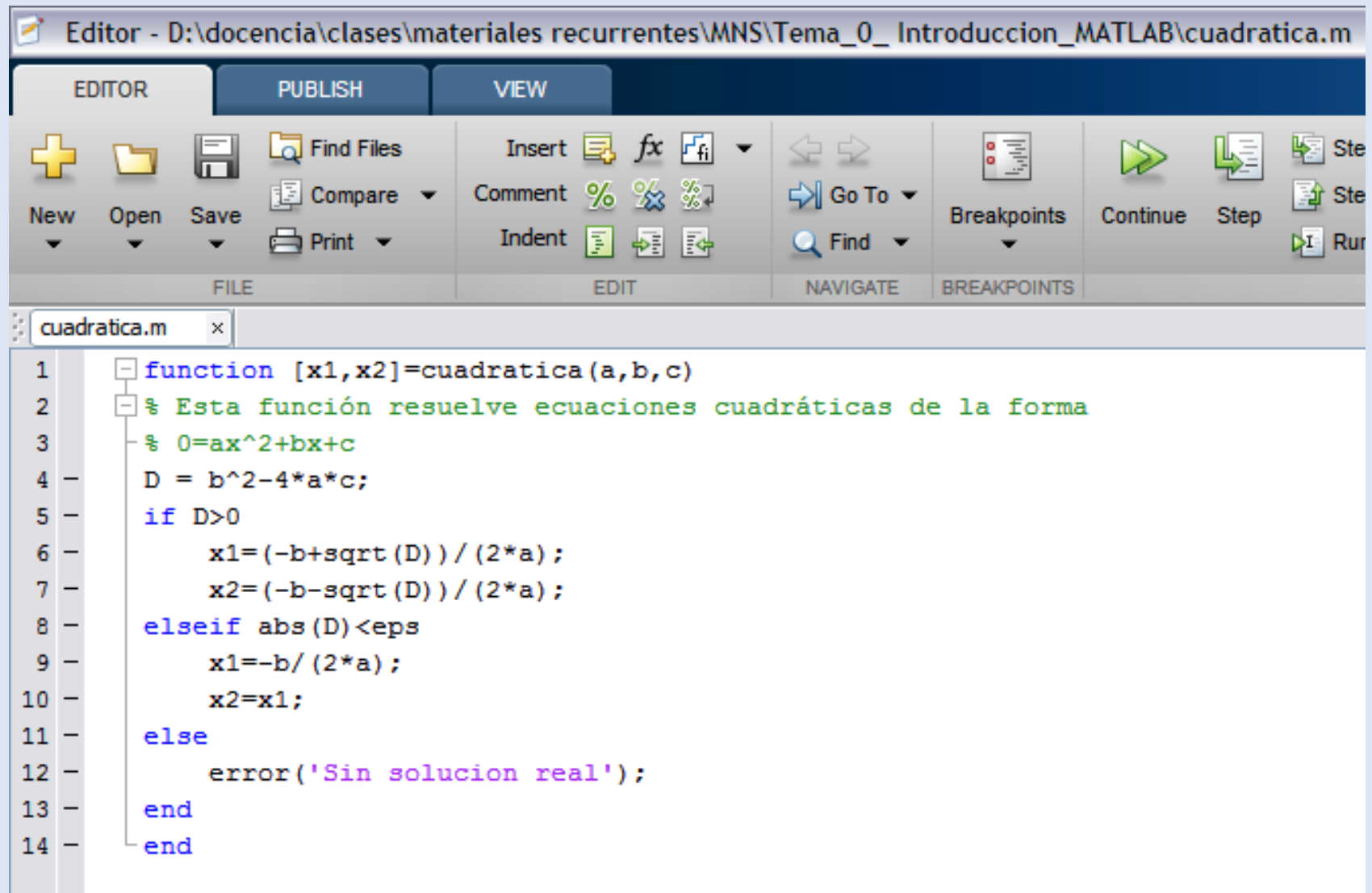
Una sentencia **if** determina si cierto número de líneas de código se ejecutan dependiendo de una condición lógica.

La forma más general de una sentencia **if** es la siguiente:

```
if condición1
    líneas de código que se ejecutan si la condición1 es verdadera
elseif condición2
    líneas de código que se ejecutan si la condición2 es verdadera
elseif condición3
    ...
else
    líneas de código que se ejecutan si condición1, condición2, ...,
    condiciónN son falsas
end
```

condición1,.. han de ser expresiones **lógicas** o bien una variable **numérica** que sólo pueda tomar los valores 1 (=verdadero) o 0 (=falso)

# Sentencia if: ejemplo



```
Editor - D:\docencia\clases\materiales recurrentes\MNS\Tema_0_Introduccion_MATLAB\cuadratica.m

EDITOR PUBLISH VIEW

New Open Save Find Files Compare Print Insert Comment Indent Go To Find Breakpoints Continue Step Run

cuadratica.m x

1 function [x1,x2]=cuadratica(a,b,c)
2 % Esta función resuelve ecuaciones cuadráticas de la forma
3 % 0=ax^2+bx+c
4 D = b^2-4*a*c;
5 if D>0
6     x1=(-b+sqrt(D))/(2*a);
7     x2=(-b-sqrt(D))/(2*a);
8 elseif abs(D)<eps
9     x1=-b/(2*a);
10    x2=x1;
11 else
12     error('Sin solución real');
13 end
14 end
```

# Ejecución condicional: switch

Una sentencia **switch** hace que se ejecuten ciertas líneas de código en función del valor de una determinada expresión que debe ser numérica o una cadena de caracteres.

**switch** expresion

**case** opcion1

código que se ejecuta si expresión=opcion1

**case** opcion2

código que se ejecuta si expresión=opción2

.

.

**otherwise**

código que se ejecuta cuando no se cumple ninguno de los case

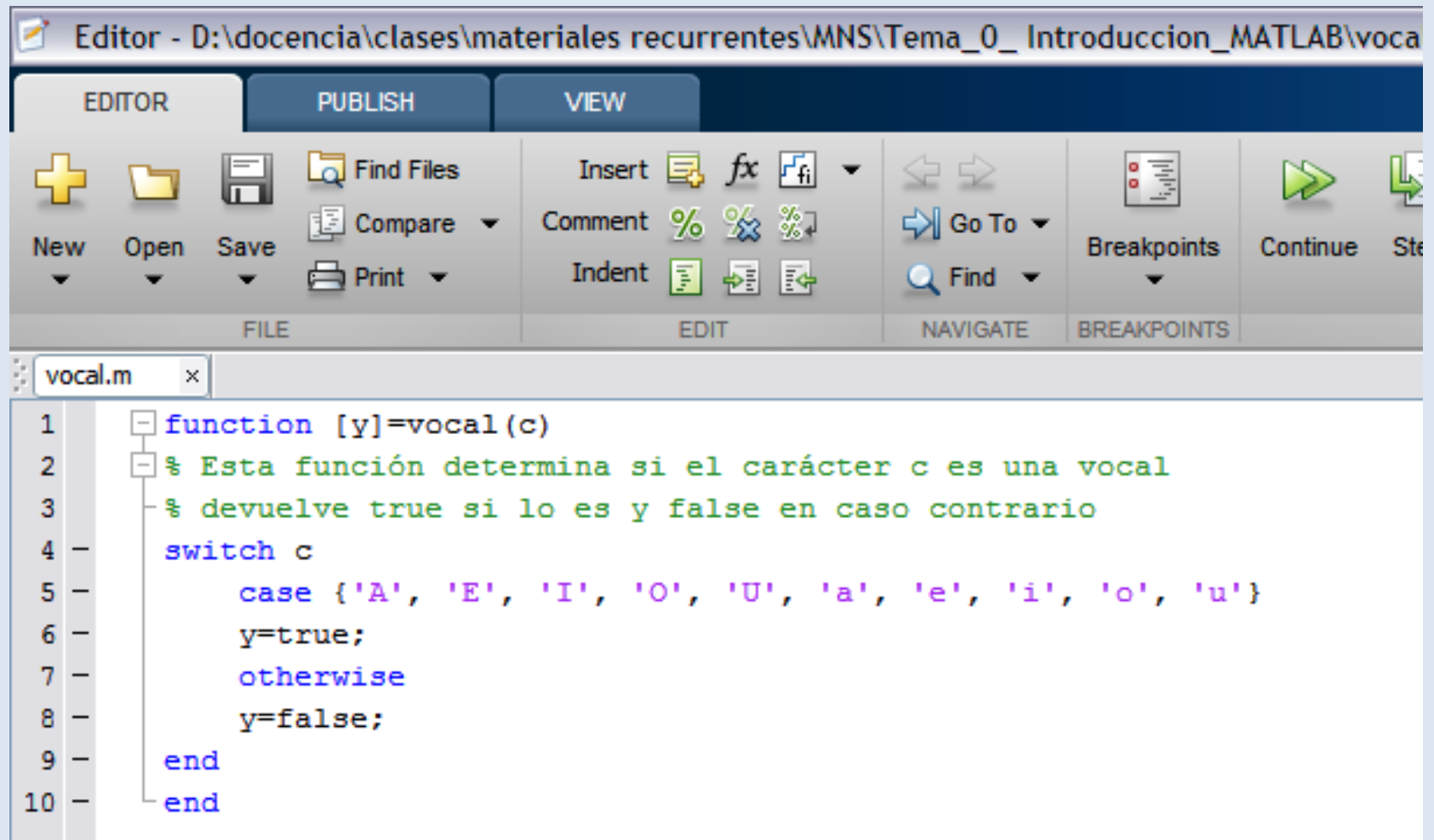
**end**

opción debe ser una variable numérica o una cadena de caracteres.

opción1 no tiene por qué consistir en un único valor: se pueden poner varios valores entre llaves separados por comas

la sentencia **otherwise** no tiene por qué incluirse si no hace falta

# Sentencia switch: ejemplo



The screenshot shows the MATLAB Editor interface. The title bar indicates the file path: `D:\docencia\clases\materiales recurrentes\MNS\Tema_0_Introduccion_MATLAB\vocal.m`. The editor has three tabs: EDITOR, PUBLISH, and VIEW. The EDITOR tab is active, showing a toolbar with icons for New, Open, Save, Find Files, Compare, Print, Insert, Comment, Indent, Go To, Find, Breakpoints, Continue, and Step. The code in the editor is as follows:

```
1 function [y]=vocal(c)
2 % Esta función determina si el carácter c es una vocal
3 % devuelve true si lo es y false en caso contrario
4 switch c
5     case {'A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u'}
6         y=true;
7     otherwise
8         y=false;
9 end
10 end
```

# Bucles

Un **bucle for** repite las líneas de código entre **for** y **end** todas las veces que sea necesario para que la variable **index** tome los valores de **inicio** a **fin** en incrementos de **salto**

```
for index = inicio:salto:fin  
    líneas de código  
end
```

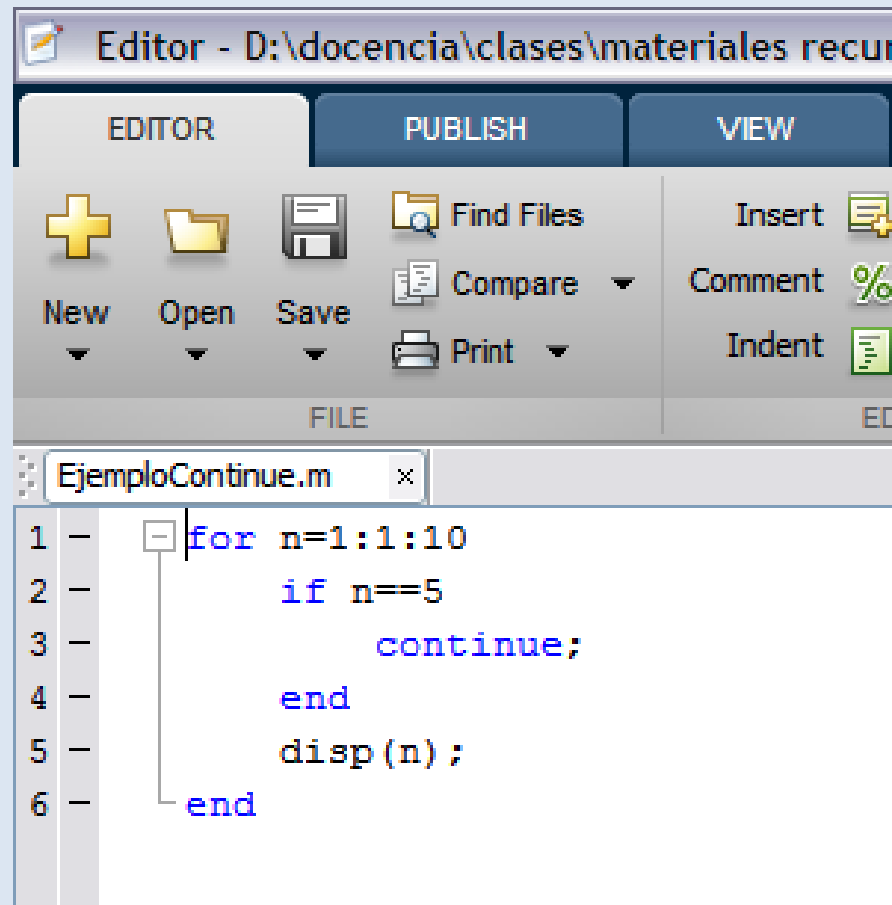
Un **bucle while** repite las líneas de código entre **while** y **end** siempre que la condición lógica **expresion** sea cierta.

```
while expresion  
    líneas de código  
end
```

Para que el bucle no se itere de forma infinita, el valor de la condición lógica **expresion** tiene que cambiar dentro del código

# Sentencia *continue*

***continue*** se usa para que un bucle ***for*** o ***while*** salte a la siguiente iteración sin ejecutar las líneas que le siguen.

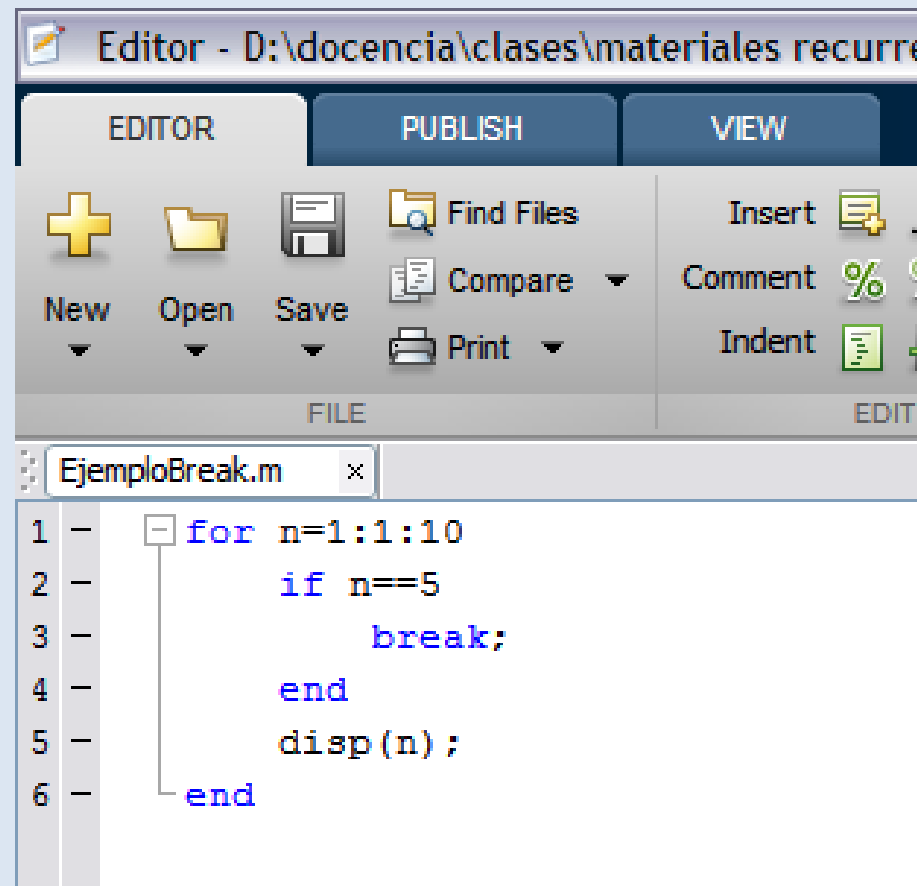


The screenshot shows a MATLAB editor window titled "Editor - D:\docencia\clases\materiales recur". The window has three tabs: "EDITOR", "PUBLISH", and "VIEW". Below the tabs is a toolbar with icons for "New", "Open", "Save", "Find Files", "Compare", "Print", "Insert", "Comment", and "Indent". The "FILE" and "ED" labels are visible below the toolbar. The editor displays a script named "EjemploContinue.m" with the following code:

```
1 - for n=1:1:10
2 -     if n==5
3 -         continue;
4 -     end
5 -     disp(n);
6 - end
```

# Sentencia *break*

***break*** se usa para salir de un bucle ***for*** o ***while*** antes de que se cumpla su condición natural de salida



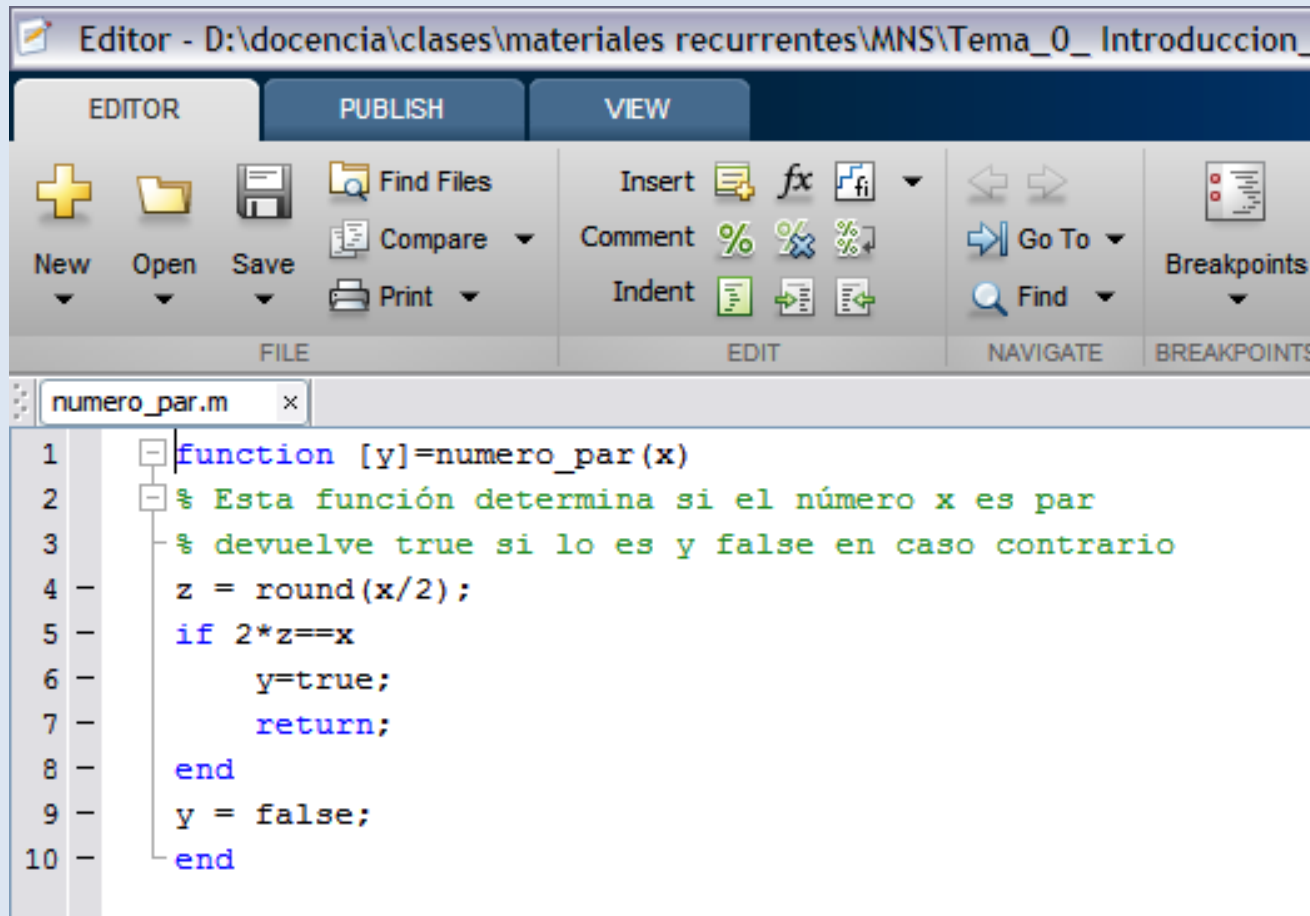
The screenshot shows a MATLAB editor window titled "Editor - D:\docencia\clases\materiales recurrentes". The window has three tabs: "EDITOR", "PUBLISH", and "VIEW". Below the tabs is a toolbar with icons for "New", "Open", "Save", "Find Files", "Compare", "Print", "Insert", "Comment", and "Indent". The main editing area shows a script named "EjemploBreak.m" with the following code:

```
1 - for n=1:1:10
2 -     if n==5
3 -         break;
4 -     end
5 -     disp(n);
6 - end
```



# Sentencia *return*

***return*** se usa para terminar la ejecución de una función y devolver el control al código que la llamó



The screenshot shows a MATLAB editor window titled "Editor - D:\docencia\clases\materiales recurrentes\MNS\Tema\_0\_Introduccion\_". The window has three tabs: "EDITOR", "PUBLISH", and "VIEW". Below the tabs is a toolbar with icons for "New", "Open", "Save", "Find Files", "Compare", "Print", "Insert", "Comment", "Indent", "Go To", "Find", and "Breakpoints". The main area displays the code for a function named "numero\_par.m". The code is as follows:

```
1 function [y]=numero_par(x)
2 % Esta función determina si el número x es par
3 % devuelve true si lo es y false en caso contrario
4 z = round(x/2);
5 if 2*z==x
6     y=true;
7     return;
8 end
9 y = false;
10 end
```



# Referencia a función

A veces, es necesario pasar una función como argumento de otra función. Para hacer esto definimos una referencia a función:

```
fref = @(variable1,variable2,...)  
foriginal(variable1,parámetro1,parámetro2,variable2,...)
```

**variable1, variable2,...**, son las variables de entrada de la referencia a función **fref**.

**parámetro1, parámetro2,...**, son variables de entrada de **foriginal** que no son variables de entrada de la función anónima. Tienen que tener valores asignados antes de la línea en la que se define la función anónima.

**fref** puede usarse como:

- 1) Una variable de entrada en otra función.
- 2) Una función cuyos argumentos de salida son los mismos que los de la función **foriginal**

# Funciones anónimas: ejemplo

## Creación:

```
>> raiz = @(x) cuadratica(1,0,-1*x);  
>> raiz(2)  
  
ans =  
  
    1.4142
```

Define la función anónima *raiz* a partir de la función *cuadratica* que teníamos definida en un M-file

## Uso como variable de entrada:

```
>> fplot(raiz,[0 2])
```

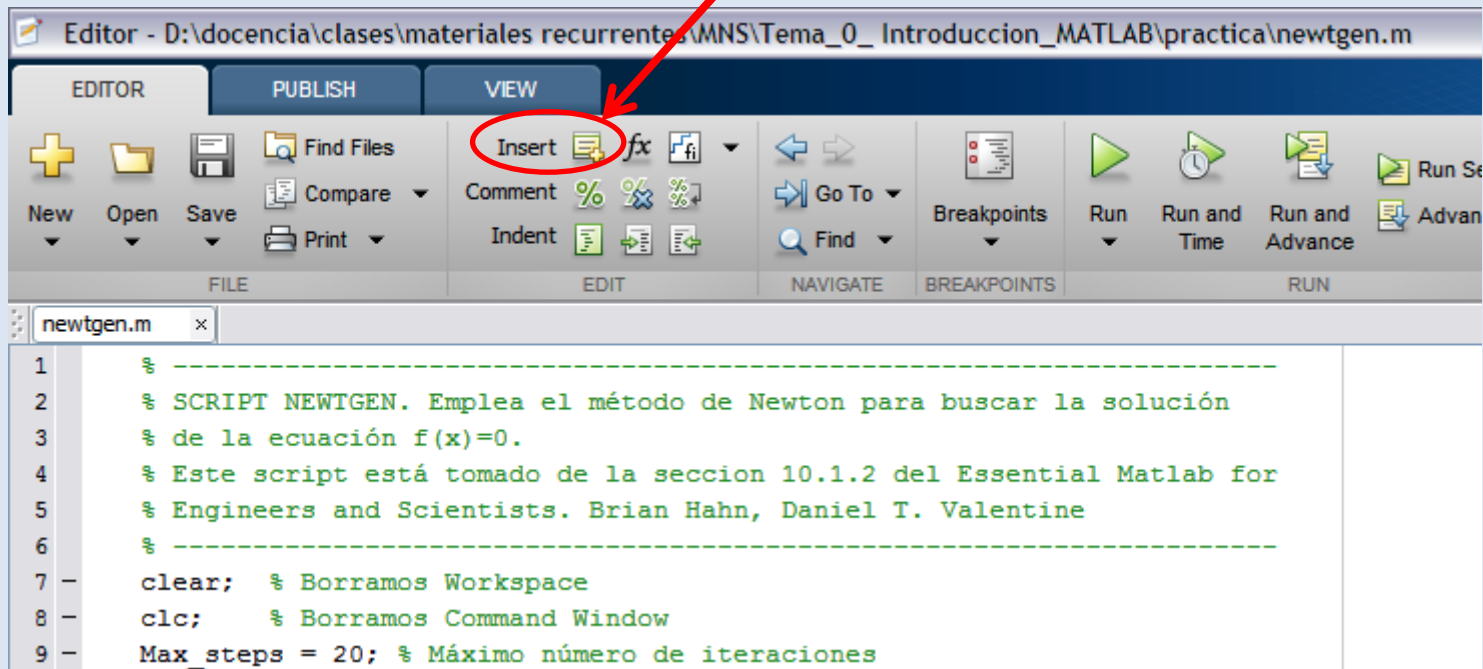
Usa la función anónima *raiz* como un variable de entrada de la función *fplot*, que hace gráficas de funciones de una única variable.

# Bloques de código

En ocasiones, un programa cuenta con distintas secciones encargadas cada una de una tarea específica.

A cada una de estas secciones la llamaremos un **bloque de código**.

Si este es el caso, suele ser conveniente marcar de alguna forma los límites de cada bloque. Para ello, pulsamos en el icono ***Insert Section Break*** en el editor



# Título de bloque de código

Para marcar un bloque de código también se puede escribir **%%** al principio del bloque.

El texto que sigue a %% es el título bloque. Servirá de índice cuando publiquemos el código. El texto bajo el título aparecerá como texto plano.

El bloque  
que  
tenemos  
seleccionado  
aparece  
sobre fondo  
amarillo

Editor - D:\docencia\clases\matrices recurrentes\MNS\Tema\_5\_ODE\_1\_variable\practica1\soluciones\_curso\_13\_14\pe

EDITOR PUBLISH VIEW

New Open Save Find Files Compare Print Insert Comment Indent Go To Find Breakpoints Run Run and Time Run and Advance

FILE EDIT NAVIGATE BREAKPOINTS RUN

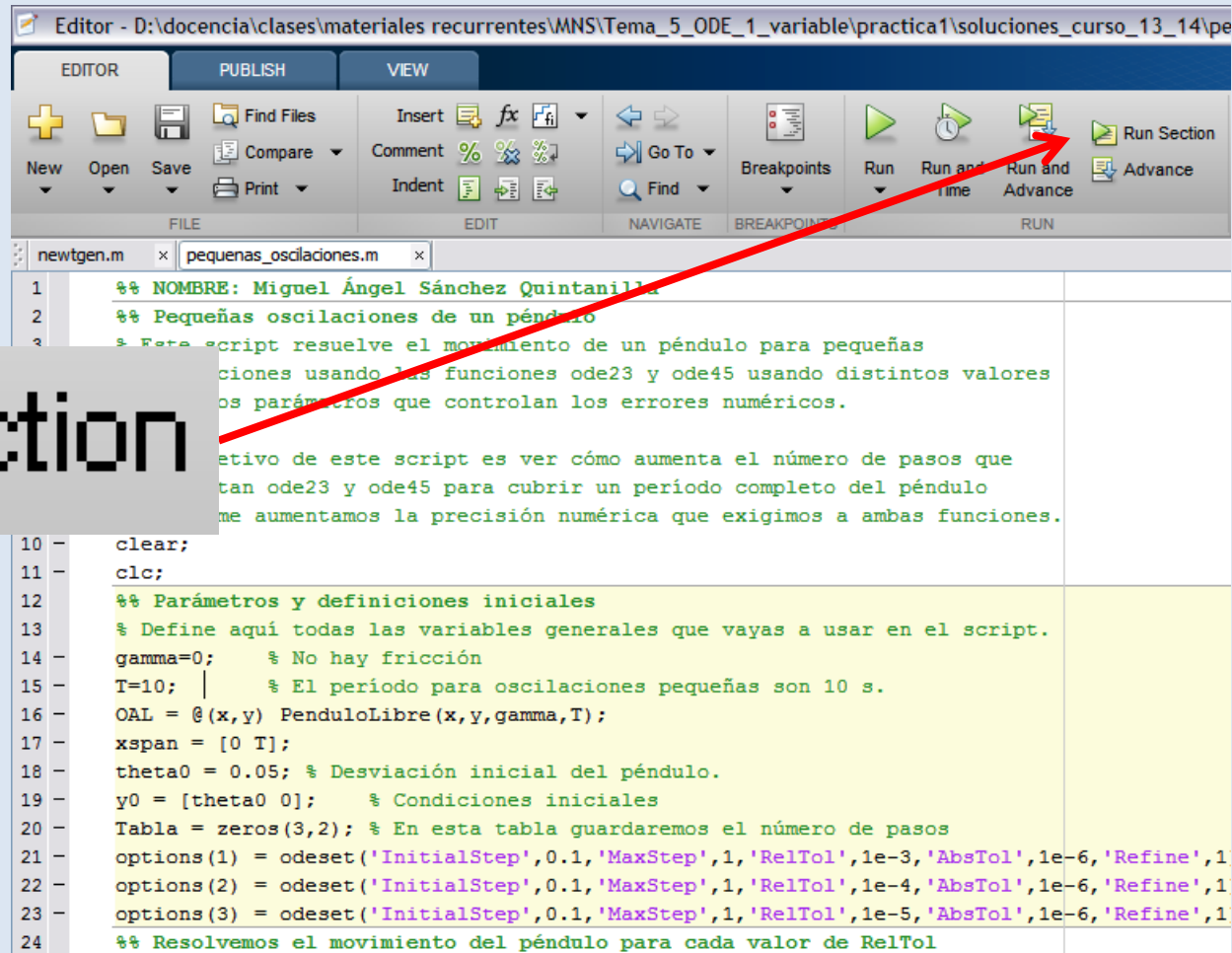
```

newtgen.m x pequenas_oscilaciones.m x
1 %% NOMBRE: Miguel Ángel Sánchez Quintanilla
2 %% Pequeñas oscilaciones de un péndulo
3 % Este script resuelve el movimiento de un péndulo para pequeñas
4 % oscilaciones usando las funciones ode23 y ode45 usando distintos valores
5 % para los parámetros que controlan los errores numéricos.
6 %
7 % El objetivo de este script es ver cómo aumenta el número de pasos que
8 % necesitan ode23 y ode45 para cubrir un período completo del péndulo
9 % conforme aumentamos la precisión numérica que exigimos a ambas funciones.
10 clear;
11 clc;
12 %% Parámetros y definiciones iniciales
13 % Define aquí todas las variables generales que vayas a usar en el script.
14 gamma=0; % No hay fricción
15 T=10; % El periodo para oscilaciones pequeñas son 10 s.
16 OAL = 0.4; y PenduloLibre(x,y,gamma,T);
17 xspan = [0 T];
18 theta0 = 0.05; % Desviación inicial del péndulo.
19 y0 = [theta0 0]; % Condiciones iniciales
20 Tabla = zeros(3,2); % En esta tabla guardaremos el número de pasos
21 options(1) = odeset('InitialStep',0.1,'MaxStep',1,'RelTol',1e-3,'AbsTol',1e-6,'Refine',1
22 options(2) = odeset('InitialStep',0.1,'MaxStep',1,'RelTol',1e-4,'AbsTol',1e-6,'Refine',1
23 options(3) = odeset('InitialStep',0.1,'MaxStep',1,'RelTol',1e-5,'AbsTol',1e-6,'Refine',1
24 %% Resolvemos el movimiento del péndulo para cada valor de RelTol

```

# Ejecución por bloques

Si un bloque es completamente autónomo (no necesita datos de otros bloques), es posible evaluar únicamente lo que hacen sus líneas seleccionándolo y pulsando en el icono:



Editor - D:\docencia\clases\materiales recurrentes\MNS\Tema\_5\_ODE\_1\_variable\practica1\soluciones\_curso\_13\_14\pe

EDITOR PUBLISH VIEW

New Open Save Find Files Compare Print Insert Comment Indent Go To Find Breakpoints Run Run and Advance Run Section Advance

newtgen.m x pequeñas\_oscilaciones.m x

```
1 %% NOMBRE: Miguel Ángel Sánchez Quintanilla
2 %% Pequeñas oscilaciones de un péndulo
3 %% Este script resuelve el movimiento de un péndulo para pequeñas
4   oscilaciones usando las funciones ode23 y ode45 usando distintos valores
5   de los parámetros que controlan los errores numéricos.
6   El objetivo de este script es ver cómo aumenta el número de pasos que
7   usan ode23 y ode45 para cubrir un periodo completo del péndulo
8   cuando aumentamos la precisión numérica que exigimos a ambas funciones.
9
10 clear;
11 clc;
12 %% Parámetros y definiciones iniciales
13 %% Define aquí todas las variables generales que vayas a usar en el script.
14 gamma=0; % No hay fricción
15 T=10; % El periodo para oscilaciones pequeñas son 10 s.
16 OAL = @(x,y) PenduloLibre(x,y,gamma,T);
17 xspan = [0 T];
18 theta0 = 0.05; % Desviación inicial del péndulo.
19 y0 = [theta0 0]; % Condiciones iniciales
20 Tabla = zeros(3,2); % En esta tabla guardaremos el número de pasos
21 options(1) = odeset('InitialStep',0.1,'MaxStep',1,'RelTol',1e-3,'AbsTol',1e-6,'Refine',1);
22 options(2) = odeset('InitialStep',0.1,'MaxStep',1,'RelTol',1e-4,'AbsTol',1e-6,'Refine',1);
23 options(3) = odeset('InitialStep',0.1,'MaxStep',1,'RelTol',1e-5,'AbsTol',1e-6,'Refine',1);
24 %% Resolvemos el movimiento del péndulo para cada valor de RelTol
```

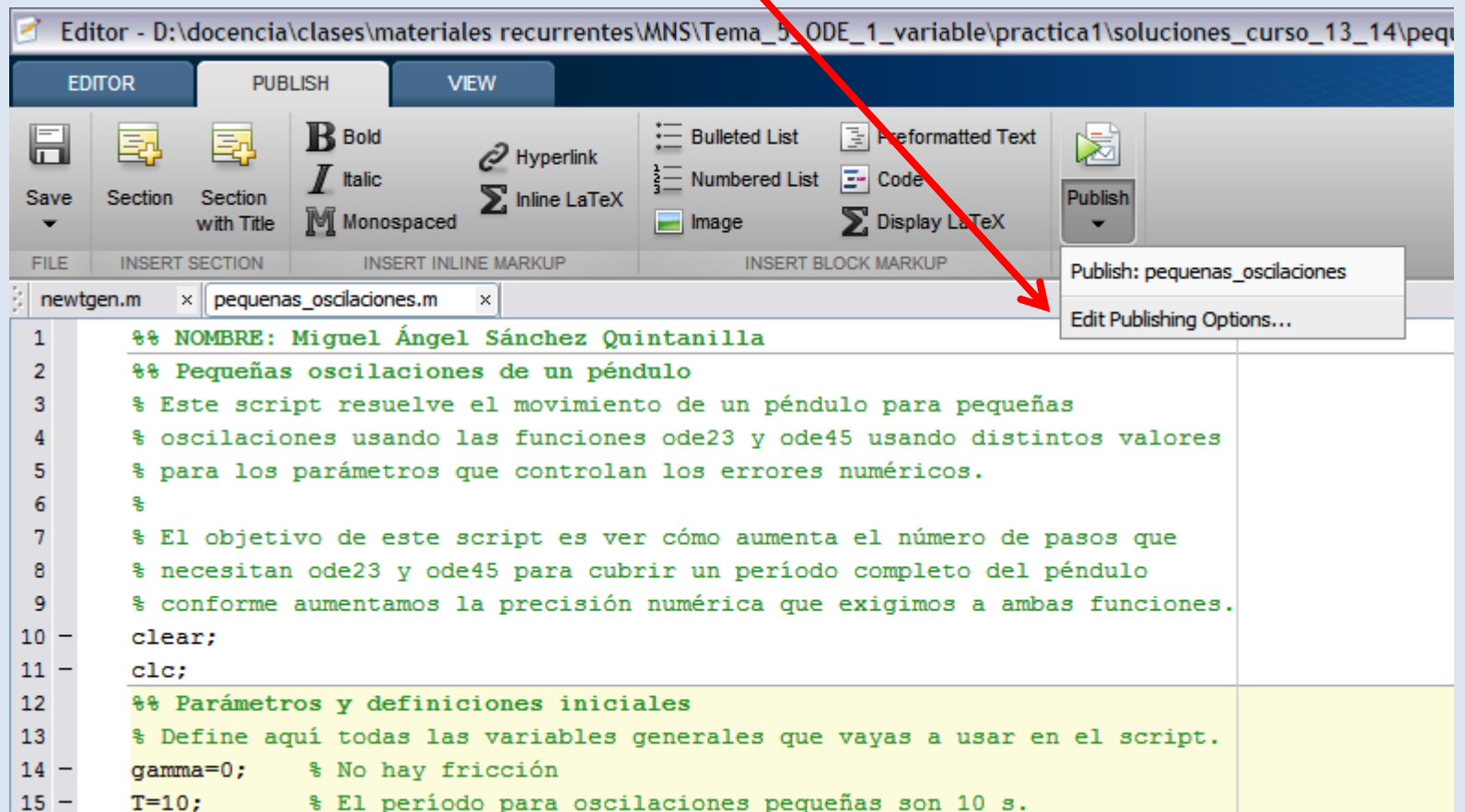


Run Section

# Publicando nuestro trabajo

Hay ocasiones en que hay que contar a otras personas el trabajo que hemos hecho. Para ello usaremos los comandos del menú **Publish** del Editor.

Aunque hay varios formatos disponibles, en este curso publicaremos en **HTML**



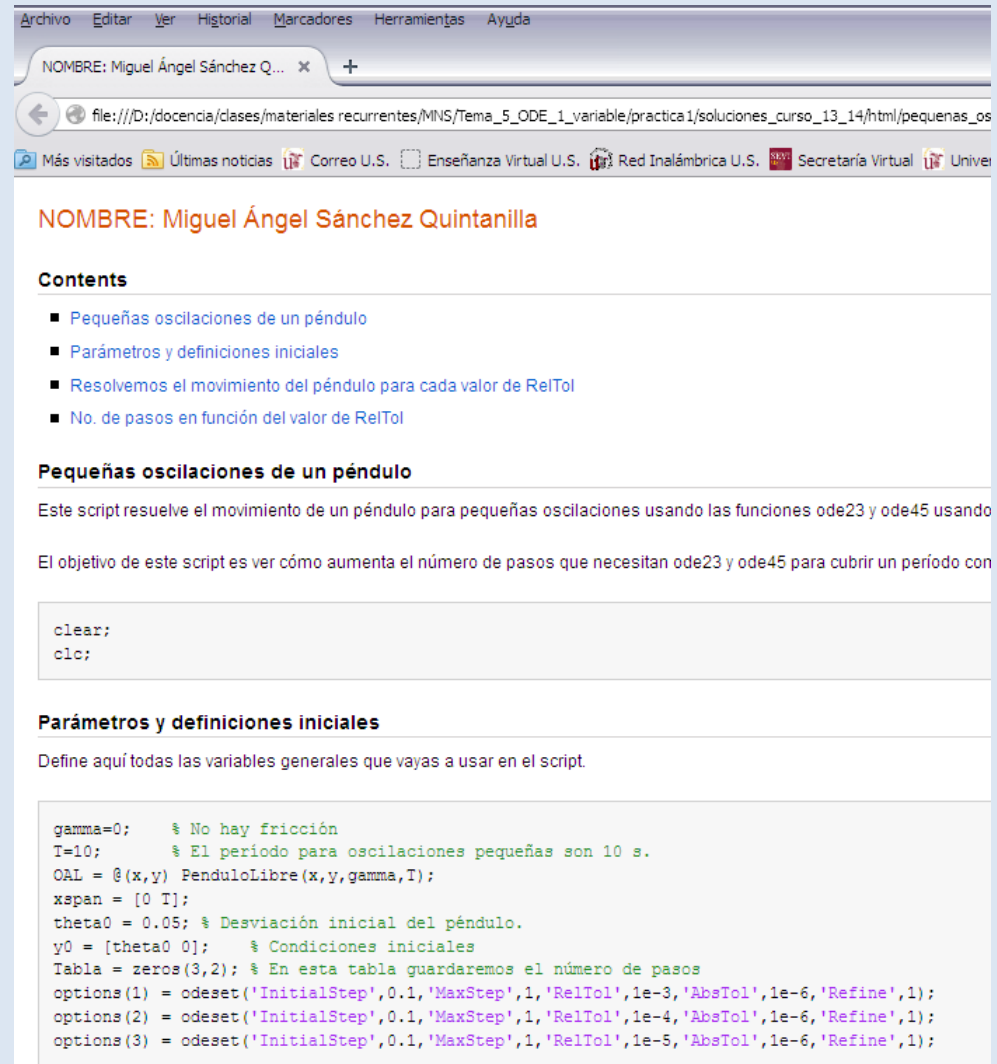
# Publicando un script

Cuando publicamos un script con Publish to HTML, Matlab:

**1)** Crea un archivo HTML con el texto del programa, poniendo un link a cada bloque de código:

**2)** Corre el programa y genera todas las figuras y otras salidas y las añade a la página web.

**3)** Todos los ficheros generados se guardan en una carpeta de nombre html en el directorio de trabajo

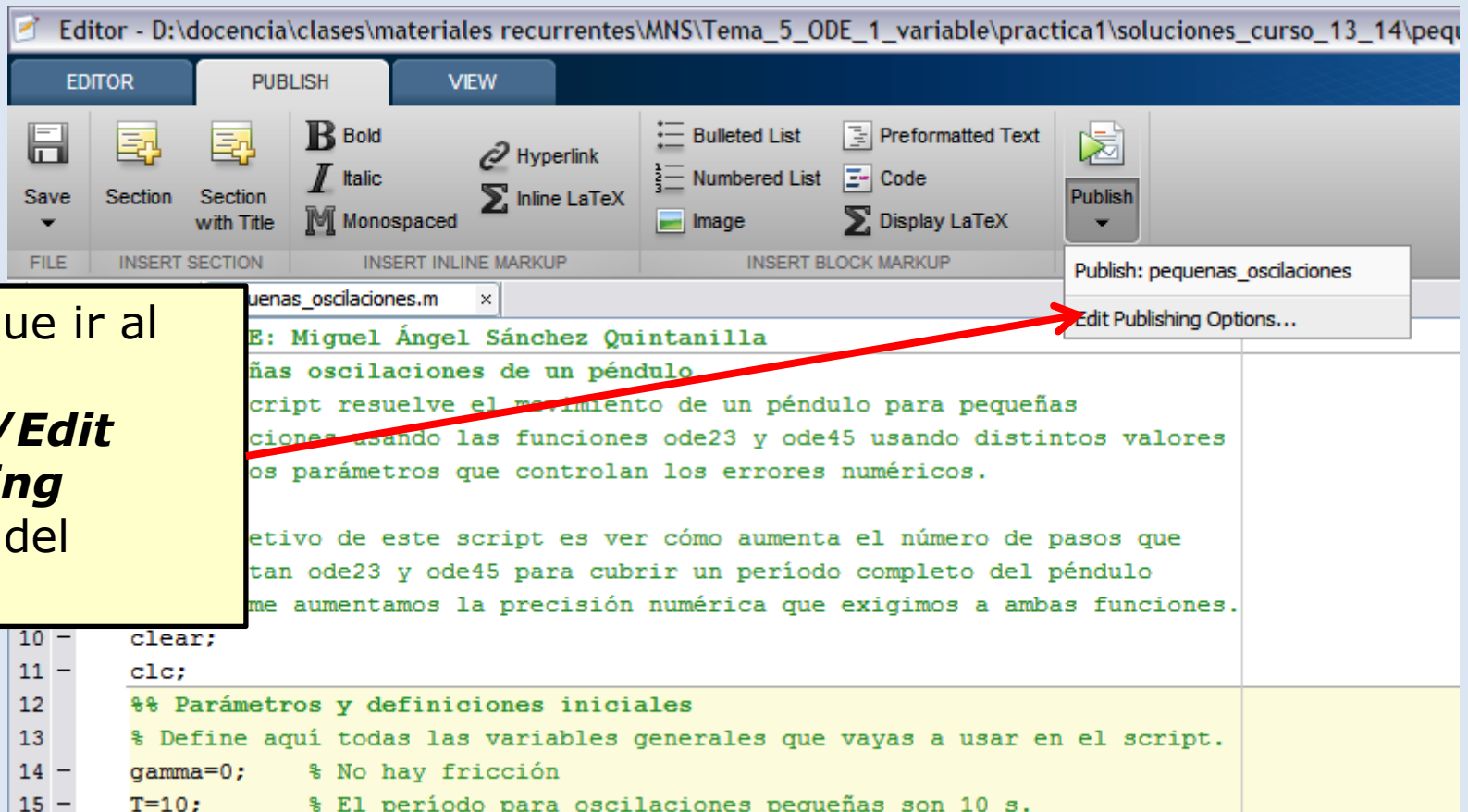




# Publicando una función (I)

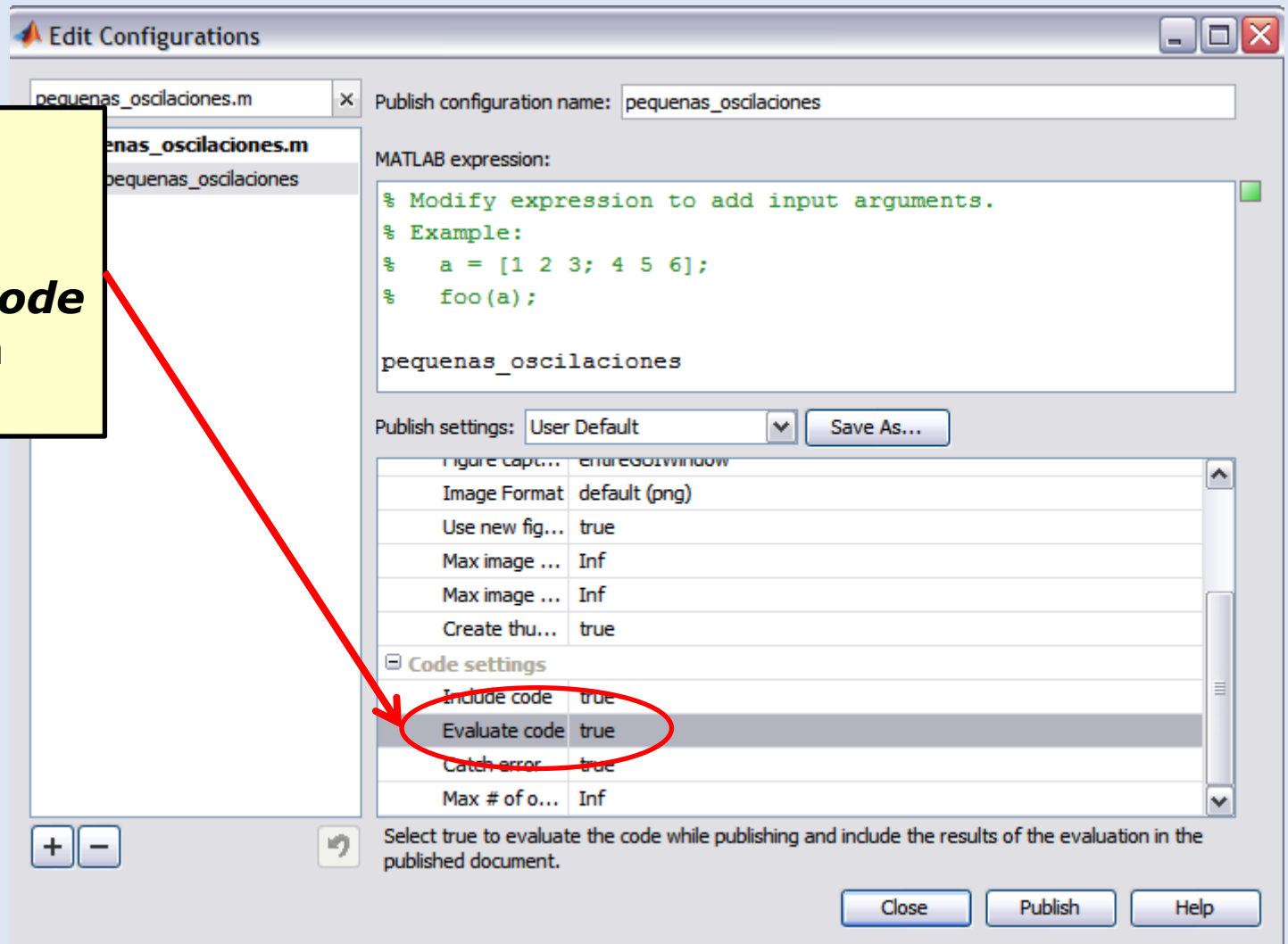
Cuando queremos publicar una función, hay que decirle a Matlab que no la ejecute o dará error. Para ello:

**1)** Hay que ir al menú **Publish/Edit Publishing options** del editor....




# Publicando una función (II)

2) Y marcar **False** en la entrada **Evaluate Code** en la opción **Publishing**



# Publicando un función (III)

Si ahora usamos Publish to html la función se imprime en un html **pero no se ejecuta.**

 cuadratica

```
function [x1,x2]=cuadratica(a,b,c)
% Esta función resuelve ecuaciones cuadráticas de la forma
% 0=ax^2+bx+c
D = b^2-4*a*c;
if D>0
    x1=(-b+sqrt(D))/(2*a);
    x2=(-b-sqrt(D))/(2*a);
elseif abs(D)<eps
    x1=-b/(2*a);
    x2=x1;
else
    error('Sin solucion real');
end
end
```

# Bibliografía del tema

Si quieres saber más detalles sobre los contenidos de este tema, lee en:

- Capítulos 2, 6, 8 y 10 del *Essential Matlab for Engineers and Scientists*. Brian Hahn, Daniel T. Valentine.
- Capítulos 2, 3, 5 y 6 del *Aprenda Matlab 7.0 como si estuviera en primero*. Javier García de Jalón, José Ignacio Rodríguez, Jesús Vidal