

Unit 2

M-file Programming

1. Introduction.....	2
1.1 M-file editor.....	2
1.2 M-file debugging.....	4
2. Scripts and functions.....	9
2.1 Script programming.....	9
2.2 Function programming.....	11
3. Programming language.....	19
3.1 Input and output commands.....	19
3.2 Indexing.....	21
3.3 Flow control.....	25
3.4 Conditional structures.....	28
3.5 Vectorization.....	28

1. Introduction

The MATLAB powerfulness is due to the fact that it is possible to run a large number of commands from a text file. These files are known as M-files because their names are of the form **filename.m**. M-files can be *functions* that accept input arguments and produce an output, or they can be *scripts* that execute a series of MATLAB statements.



Commercial toolboxes and other free access toolboxes contain M-files. MATLAB users can edit and modify the M-files of the toolboxes and they can create their own M-files as well.

To list the toolboxes installed in your computer, together with their versions and date, type **>>ver** in the command window.

To list the contents of a particular toolbox, you can simply enter **>>help toolbox_name**. For example, **>>help matlab\general** or **>>help control**.

1.1 M-file editor

M-files are text files. Therefore any text editor (such as the Windows Notepad) can be used to edit them. However it is advisable to use the M-files editor available in the MATLAB itself since it has debugging utilities and makes the programming easier due to the color code used with the reserved words.

Click on the  icon to open the M-file editor ( in v8). You can also double-click on any already existing M-file or use the menu bar options: **File** → **New** → **M file**.

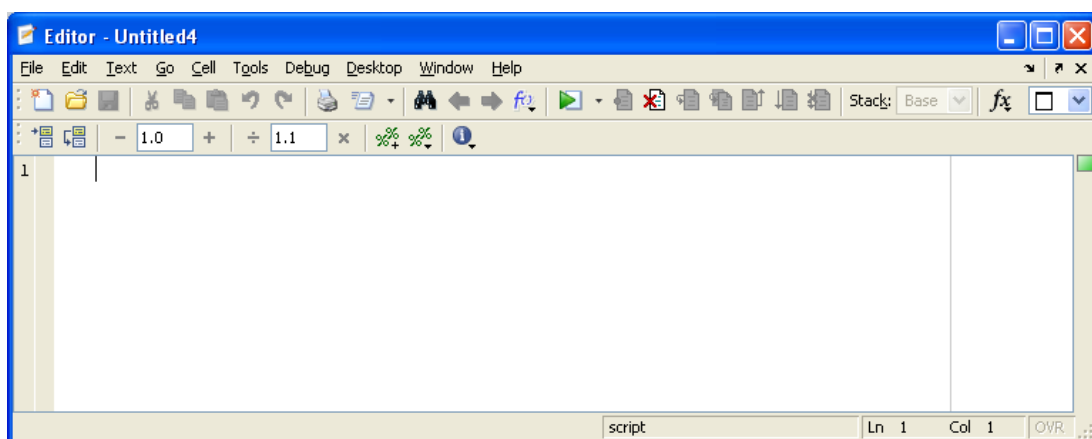


Fig. 1. M-file editor in MATLAB v7.11

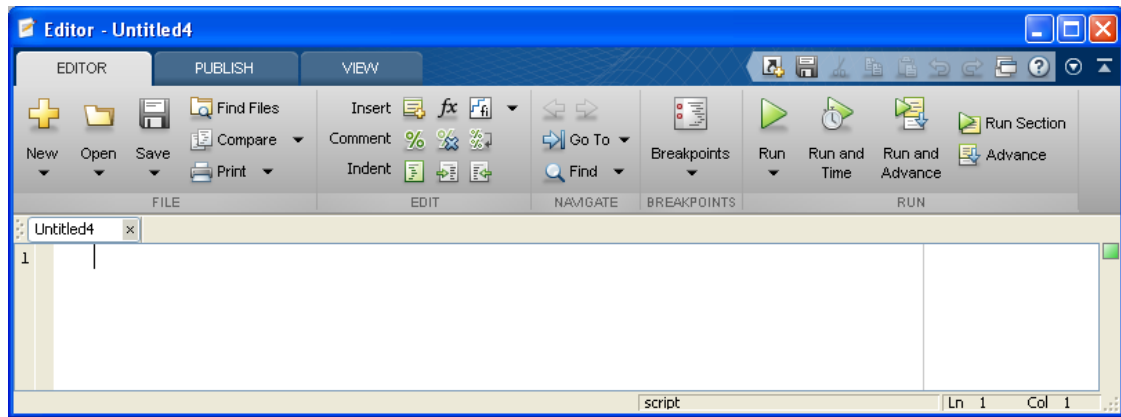



Fig. 2. M-file editor in MATLAB v8

Work folder: By default, M-files created by the user are stored in the <work> directory. Apart from the M-file **filename.m**, the M-file editor also produces an autosave file of name **filename.asv**.

To see the code of **filename.m** without having to open the M-file editor simply type **>>type filename** in the command window. For instance, **>>type roots**. Note: built-in functions are not M-files but there exist M-files with the same name containing the help comments. For instance, try **>>type who** and **>>type who.m**.

Menu bar options: The menu bar and the toolbar present many utilities. Look to the available options and try to deduce what they do.

Some interesting options are:

- **File → Preferences...** (to change the general parameters regarding colour, text fonts, etc.)
- **Cell** (if you want to execute only a part of the code)
- **Tools → Check Code with M-Lint** (to get suggestions about correcting and improving your code)
- **Debug → Run** (to save and run the M-file. This option is equivalent to click on the  icon)

In the v8 the Preferences options can be found in **Home>Environment** (see Fig. 3).

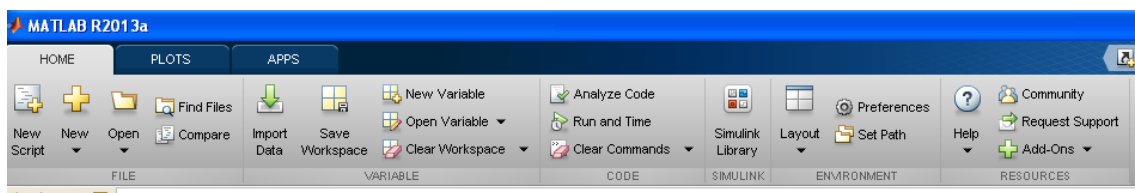


Fig. 3. Environment in v8

1.2 M-file debugging

Color hints: In versions 7 and 8, colors in the frame of the M-file editor give hints about your code. If there are no errors a green square appears on the frame. If there are errors a red square appears instead. If there are warnings or suggestions (such as the semicolon in line 1 in the example below), the square that appears in the frame is orange.

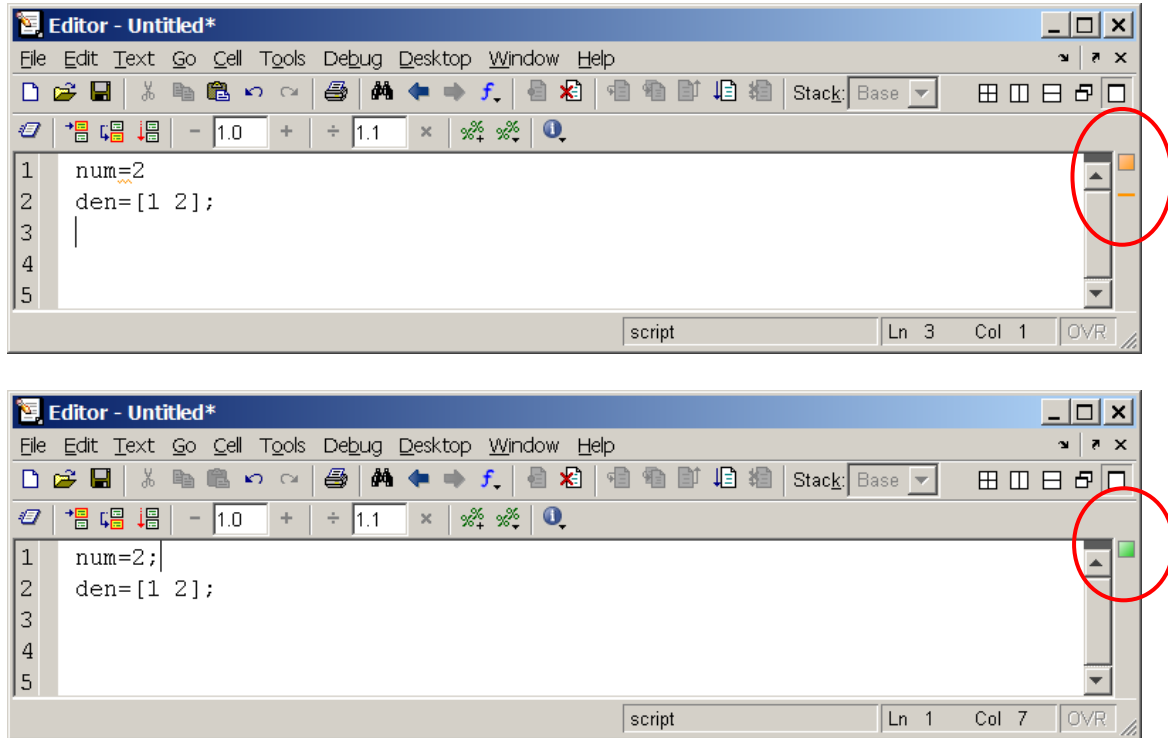










Fig. 4. M-file editor in MATLAB v7

Breakpoints: It is possible to insert breakpoints. Click on the  icons to insert or to clear breakpoints in the selected lines or use the **Debug** options in the menu bar.

To run the file click on  or . Use the remaining icons     to run the file step-to-step, to clear all breakpoints, etc.

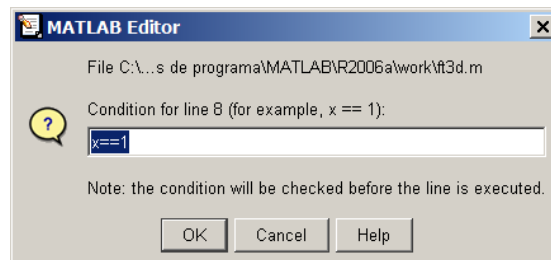
While the breakpoint mode is used, the prompt in MATLAB is **k>>**. Also, the current line is indicated by means a green arrow inside the M-file:

5  pzmap(num,den) ,

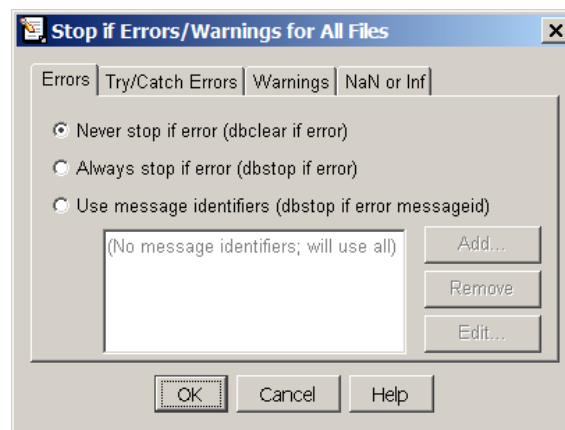
Breakpoints are of several types:


- **Standard:** They are the red ones. When the running code finds a standard breakpoint, the running is paused.

- **Conditional:** They are the yellow ones. To insert a conditional breakpoint, click on the selected line and select **Debug → Set/Modify Conditional Breakpoint...** A dialog box is opened to introduce the condition that will pause the code execution:



- **Error:** They also are red. To insert an error breakpoint click on the selected line and select **Debug → Stop if Errors/Warnings**. A dialog box is opened to introduce the error type, warning type or value (**NaN**, **Inf**) that will stop the code execution:



M-Lint Code Check: In v7, to activate it go to **Tools → Save and Check Code with M-Lint**. In v8 go to  **Analyze Code**.

The example in Fig. 5 illustrates the code checking. The report indicates the following issues:

- It says that functions **meshdom** and **fmins** do not exist anymore and it suggests using **meshgrid** and **fminsearch** instead.
- It indicates which lines do not have semicolon.
- It warns us of the existence of variables that have not been used (**hdb**).
- Finally, it says that we cannot declare a function in the middle of a script file (functions can only be declared inside other functions). Therefore it is necessary

to convert the script `ft3d` to a function. To do so, the first line in the file must be `function ht3d`.

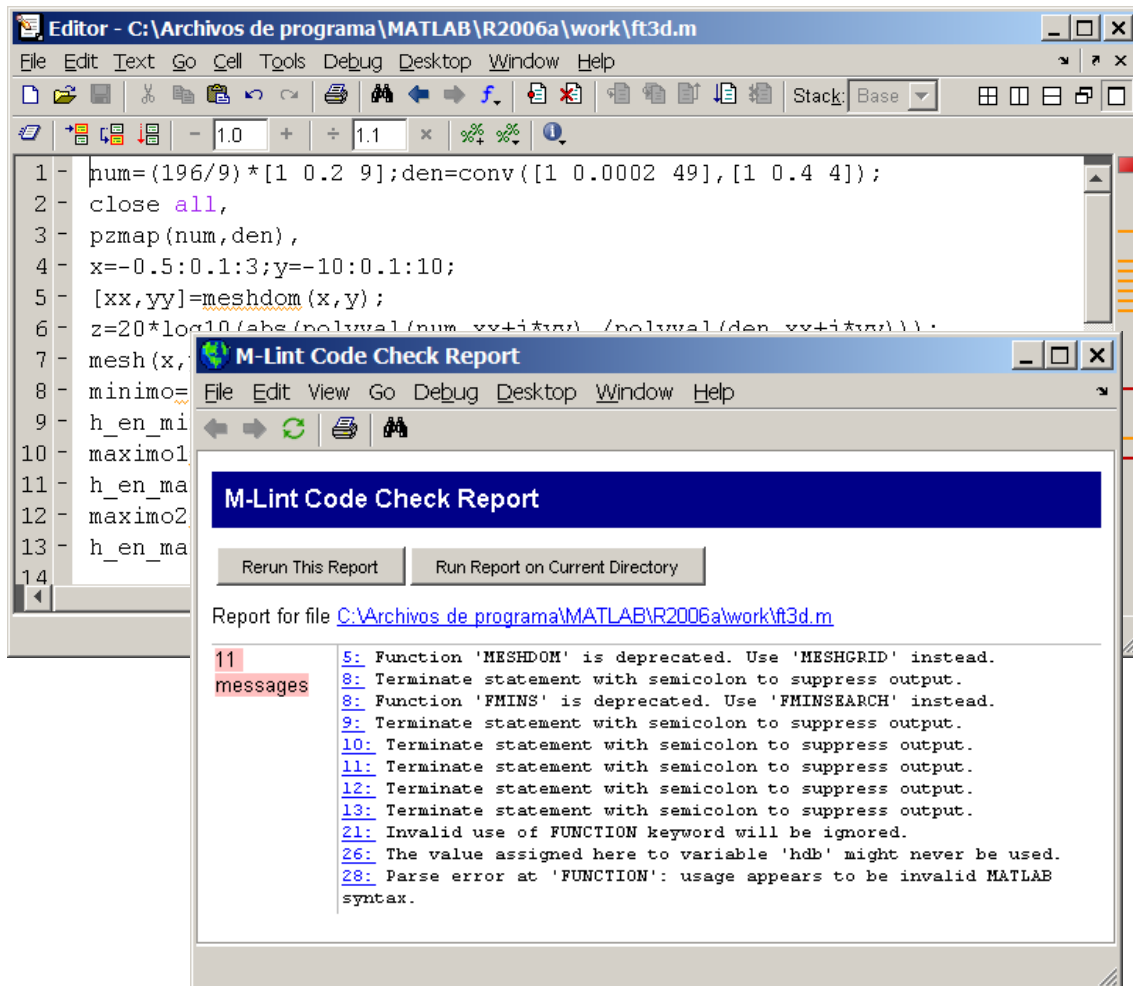



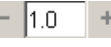
Fig. 5. Code analysis in MATLAB v7

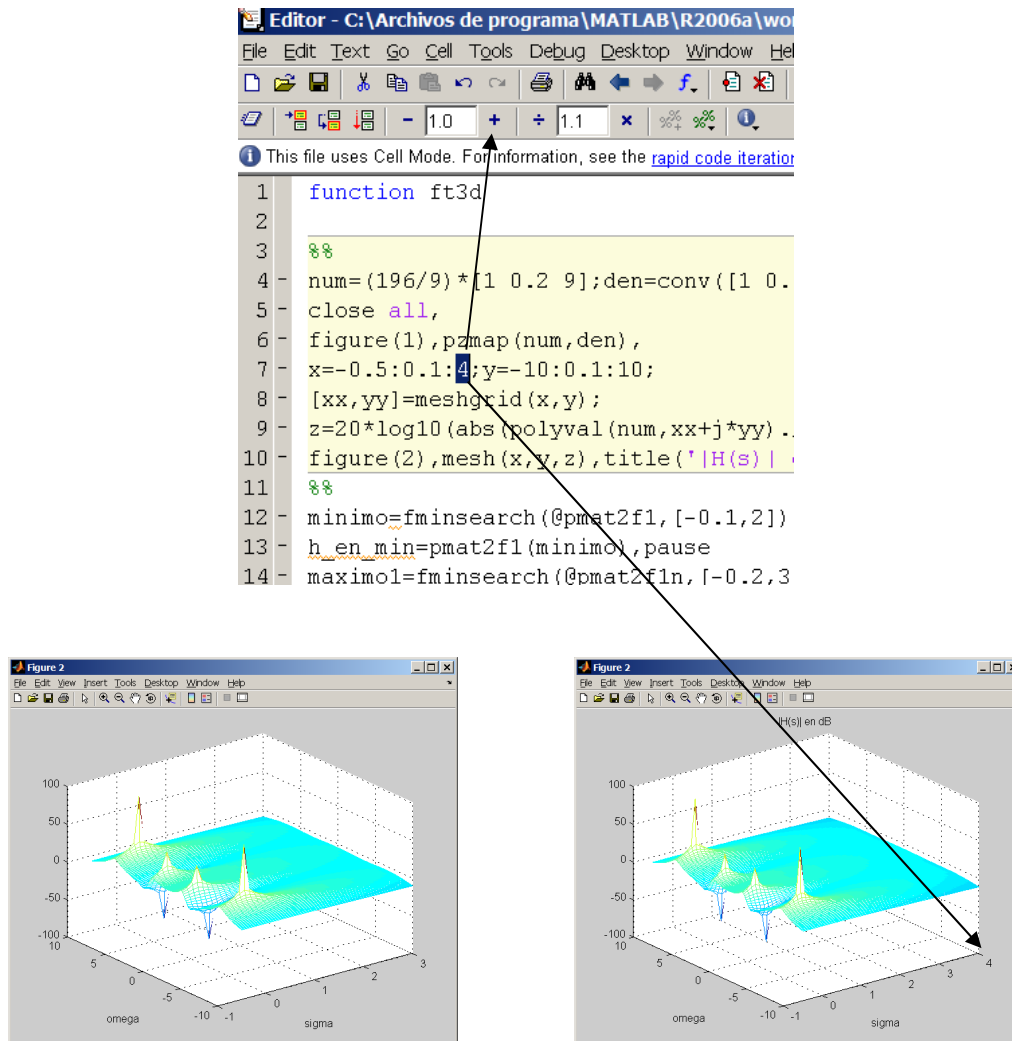
Cell mode: To activate cell mode select **Cell** → **Enable Cell Mode**. The following bar corresponds to the cell mode:



The cell mode is useful to execute selected parts of the code or to change parameters without having to save the M-file.

To specify which lines form a cell, type `%%` before and after the selected lines (or use the options in the menu bar). To run a cell, select it and click on .

To change a parameter value and immediately see the effect, select the parameter and use the `+` and `-` signs in the button . The cell will run automatically (it is the so-called “rapid code iteration”).



Profiler: Select the option **Tools** → **Open Profiler** and, once inside, click on the **Start Profiling** button.

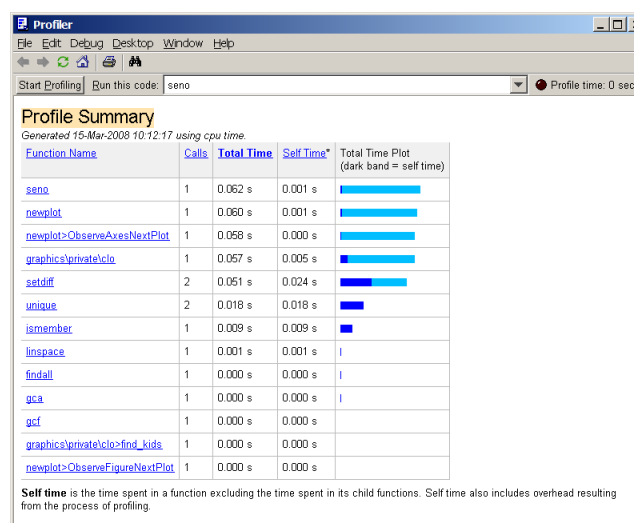



Fig. 6. Profiler in v7

The profiler tool allows seeing the time used by every line of the M-file code and how many times each command is called. This tool is useful to eliminate redundancies and to write more efficient files.

Publishing: Click on  to create an html document with the script commands as well as the script results. For instance:

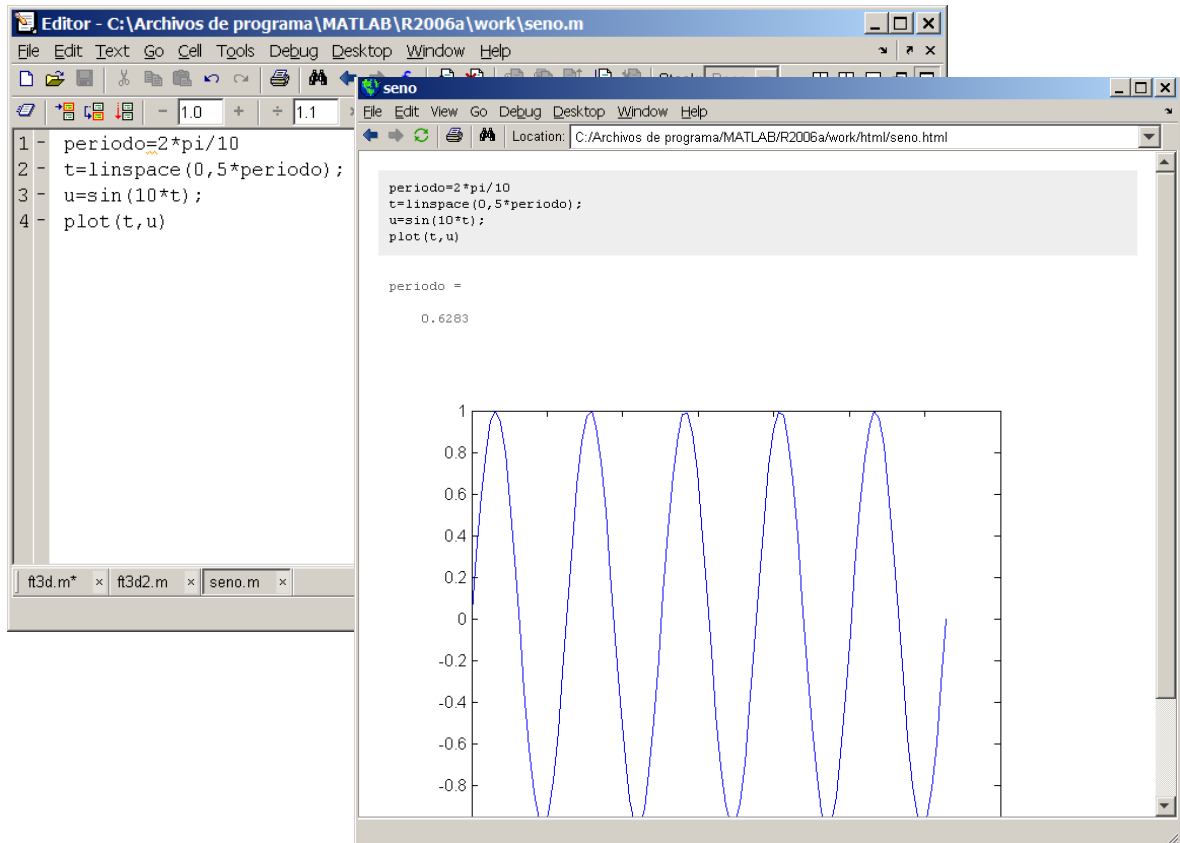


Fig. 7. Publishing in v7

2. Scripts and functions

There are two types of M-files: scripts and functions.

Scripts:

- They have not input or output arguments.
- To execute a script file you can simply type its name (with no extension) in the command window.
- All variables created by the script are stored in the workspace.

Functions

- Function files may have input and/or output arguments.
- To execute a function file you have to specify the value of its input arguments, if there are any.
- Internal variables created inside the function are not stored in the workspace (only variables declared as output arguments or declared as global variables are stored in the workspace).

2.1 Script programming

A script is a text file, of extension ***.m**, that contains a sequence of MATLAB commands (simple commands, functions and invocations to other scripts), written as if they were called from the command window (but without the prompt **>>**).

Example 1. Script programming

Next figure shows the script called **curva.m**.

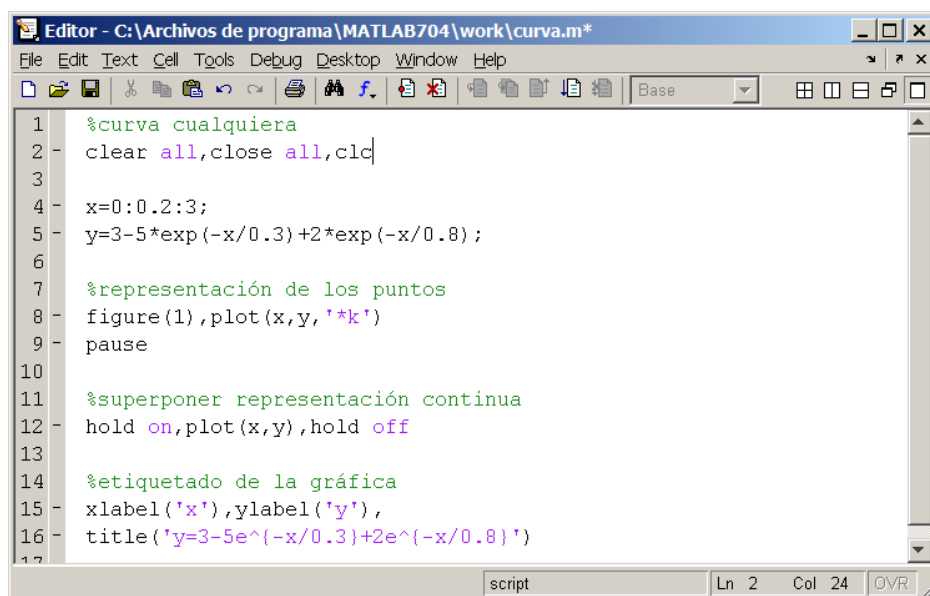
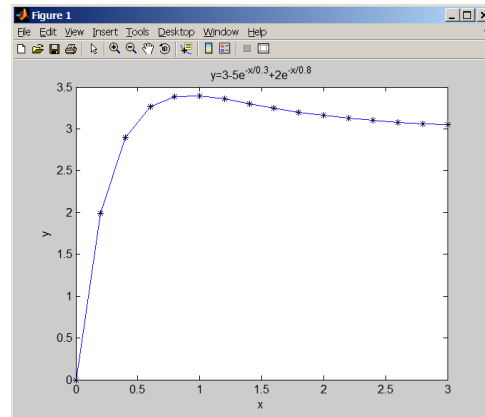
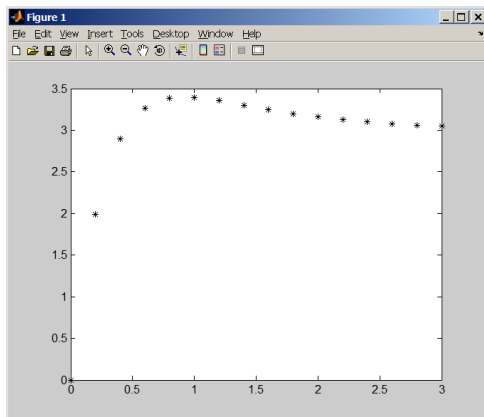




Fig. 8. Example of a script type M-file

Its execution results in the two following figures:



Scripts use the variables stored in the MATLAB workspace, and the variables created by the script are stored in the workspace as well.

To execute a script it is enough to type its name (with no extension) in the command window. Also in the M-file editor you can select the menu option **Debug** → **Run** (**F3**) or click in the  icon (or in the  icon, depending on the version).

Inserting comments: To insert a text comment, simply type the % symbol at the beginning of the comment (it is not necessary to close the comment with another %). Notice that green is the default color for comments.

Cleansing commands: Although not mandatory it is recommended to begin a script file with some cleansing instructions such as **hold off**, **clear all**, **axis**, **subplot**, **close all**, **close all hidden**, **clc**...

Other useful commands:

- **pause**: it pauses the running script until the user press any key. It is also possible to stop the script a particular time, for instance, **pause(2)**.
- **disp**: displays text in the command window. For instance, the commands

```
N=15.6;
disp(['And the N value is... ', num2str(N), '!!!'])
```

give the following result:

```
And the N value is... 15.6!!
```

(**num2str** means number-to-string. Notice that the input argument in **disp** is a character string). The following syntax is also valid (now, the input argument of **disp** is a **cell** of strings. The cell is indicated by means the { } symbol):

```
>> disp({'hi';'bye'})
      'hi'
      'bye'
```

- **figure**: Before a plot command, it is a good idea to include a command **figure**, **figure(1)**, **figure(2)**,... This way, when the script is running, figure windows will be directly opened in the front of the screen.
- **echo on/off**: This command is used in many demos in order to present, during the running of the script, the commands that contain the script.
- **tic** and **toc**, also **etime**: They show the elapsed time spent by a command or a group of commands.
- **...**: if a command is too long you can put the three points and continue in the next line (this is useful when entering large matrix data).

2.2 Function programming

Scripts execute a series of MATLAB statements. Functions also execute a series of MATLAB statements but they accept input arguments and may produce one or several outputs. Functions are preferred to scripts in situations where a same code has to be used several times, for different parameters which can be introduced in an external way.

Example 2. Function programming

Next figure shows the **suma_resta** function file.

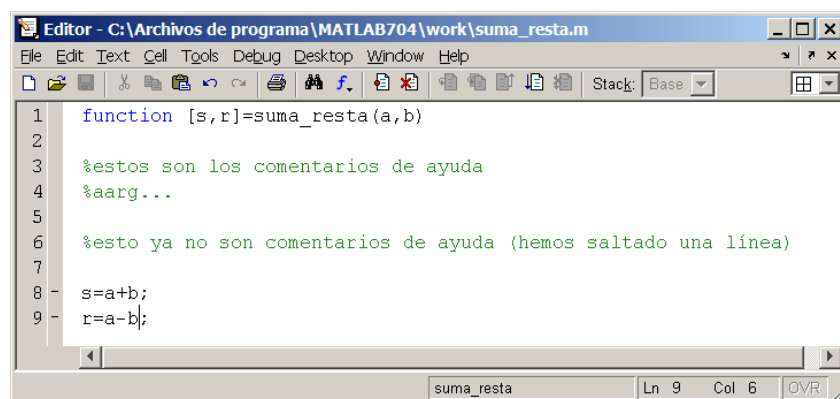


Fig. 9. Example of a function type M-file.

In the first line we declare the function name along with the list of input and output arguments.

It is possible to declare several functions inside the same file. However the file name must be the primary function name.

Running a function: Next figure shows the use of the function:

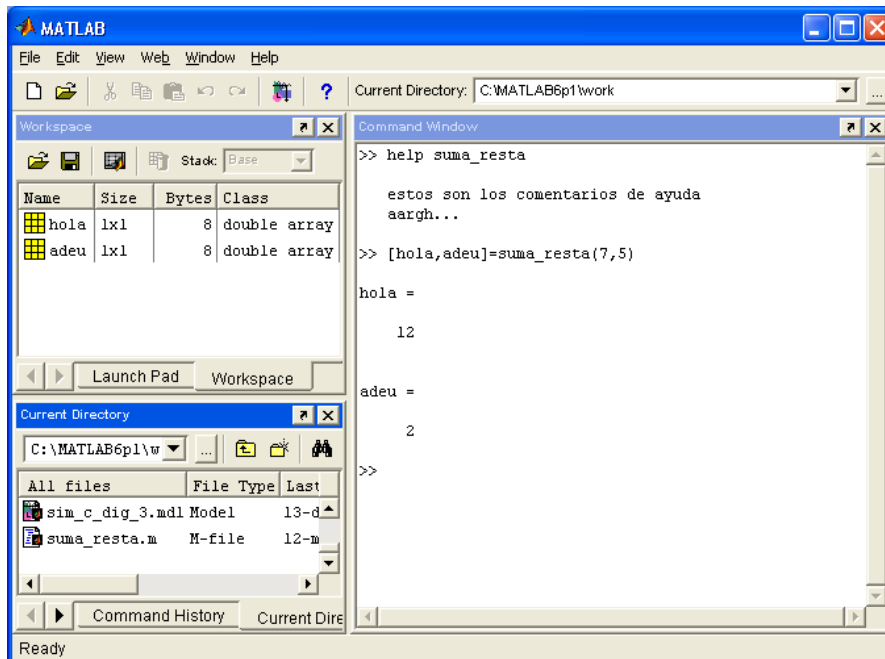


Fig. 10. Help and run of suma_resta function in the command window.

Structure: A function M-file consists of three parts:

- Header: Here we declare the function:

```
function [sine,cosine,tangent]=function_name(ang)
```

- Help comments: They are optional. Help comments is what appear in the command window when we type `>>help function_name`

```
% FUNCTION_NAME computes sine, cosine and tangent of
% the angle stored in variable 'ang'
```

If a blank line is inserted in between the help comments lines, help comments will be considered only the ones before the blank line.

- Commands:

```
sine=sin(ang) ;
cosine=cos(ang) ;
tangent=tan(ang) ;
```

To call a function one must use the following syntax:

```
»[output_arguments] = function_name (input_arguments)
```

Notice that:

- Output arguments are inside brackets
- Input arguments are inside parentheses
- It is not necessary to finish the file with the instruction **end**. However, if you close a function with an **end**, then all functions inside the same M-file must be closed with an **end**.
- The name of the file must coincide with the name of the main function
- The comments are preceded by the symbol **%** (it is not necessary put another **%** to close the comment)
- The helps of the toolboxes' functions indicate the function aim and its syntax. Although in these helps the name of the function always appears in uppercase, the functions in MATLAB are usually called in lowercase.

Global variables: M-file scripts do not need global variables since they operate directly on the data in the workspace.

On the contrary, for M-file functions, internal variables (provided they are not output arguments for the function) are by default *local* to the function and they are not passed to the workspace. To overcome this situation, we can define global variables (outside and inside the function).

```
>>global SAMPLING_PERIOD
```

Although not mandatory, it is usual to name such variables with uppercase long names.

Number of input and output arguments. Optional parameters: Functions **nargin** (number of input arguments) and **nargout** (number of output arguments) allow different usages for a same function.

Consider for example the function **step** (from the *Control Systems Toolbox*): with no output arguments, this function plots the step response of a system. When called with an output argument **y**, it stores the step response samples in variable **y** and does not plot them.

Example 3. Using nargin and nargout

```
function [numz,denz]=discret(nums,dens,Type)

%Syntax: [numz,denz]=discret(nums,dens,Type)
%           Type can be 'Tustin', 'BwdRec' or 'FwdRec'

if nargin==0
    disp('Enter the system, now!!!')
else
```

```

if Type=='Tustin'
    statements to perform the following discretization  $s = \frac{2}{T} \frac{z-1}{z+1}$ 
end
if Type=='BwdRec'
    statements to perform the following discretization  $s = \frac{z-1}{Tz}$ 
end
if Type=='FwdRec'
    statements to perform the following discretization  $s = \frac{z-1}{T}$ 
end
end

if nargout==0
    disp('buf!')
end

```

```

>> help discret

Syntax: [numz,denz]=discret(nums,dens,Type)
        Type can be 'Tustin', 'BwdRec' or 'FwdRec'

>> [a,b]=discret

Enter the system, now!!!
Warning: One or more output arguments not assigned during call
to 'discret'.

>> discret(2,3,'Tustin')

buf!

>>

```

Other interesting functions are **varargin** (variable number of input arguments) and **varargout** (variable number of output arguments); and **nargchk** (check number of input arguments) and **nargoutchk** (check number of output arguments).

Example 4. Using varargin and varargout

A function can be called with a varying number of input arguments if we declare it as follows:

```

function y=func1(x,varargin)

%with, the following statements, for instance
figure,plot(x,varargin{:})

```

```
varargin,
x1=varargin{1},
x2=varargin{2},
x3=varargin{3},
x4=varargin{4},
```

Thus, we can use a variable number of input arguments after x. If we call the function as below we are using 4 additional input arguments among x:

```
>>func1(sin(0:.1:2*pi), 'color',[1 0 0], 'linestyle', ':')
```

The inner variable “varargin” is a 4x1 cell array and its components are, respectively, varargin (1) = 'color', varargin (2) = [1 0 0],...

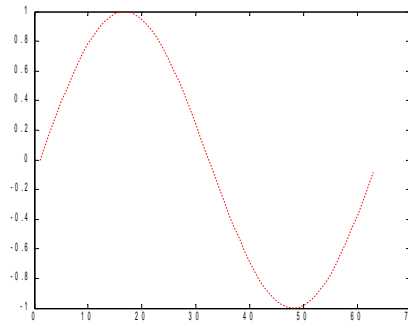
```
varargin =
    'color'    [1x3 double]    'linestyle'    ':'

x1 =
color

x2 =
     1         0         0

x3 =
linestyle

x4 =
:
```



If we want a function which generates a variable number of output arguments we can use varargout:

```
function [varargout]=func2(x,varargin)
```

Auxiliary and nested functions: A function type M-file can include several functions.

If these functions are auxiliary functions used by the main function, they are all declared at the end of the file and it is not necessary to finish none of them with the word **end**. However, if a function is finished with an **end**, then all functions must be finished with an **end** word. In next example **pmat2f1** and **pmat2f1n** are auxiliary functions for the main function **ft3d**.

To access the auxiliary functions use the symbol >.

```
>> help ft3d>pmat2f1
    |H(s)| en dB
```

Example 5. Auxiliary functions

```

function ft3d

num=(196/9)*[1 0.2 9];den=conv([1 0.0002 49],[1 0.4 4]);
x=-0.5:0.1:4;y=-10:0.1:10;[xx,yy]=meshgrid(x,y);
z=20*log10(abs(polyval(num,xx+j*yy)./polyval(den,xx+j*yy)));
mesh(x,y,z),title('|H(s)| en dB'),xlabel('sigma'),ylabel('omega'),

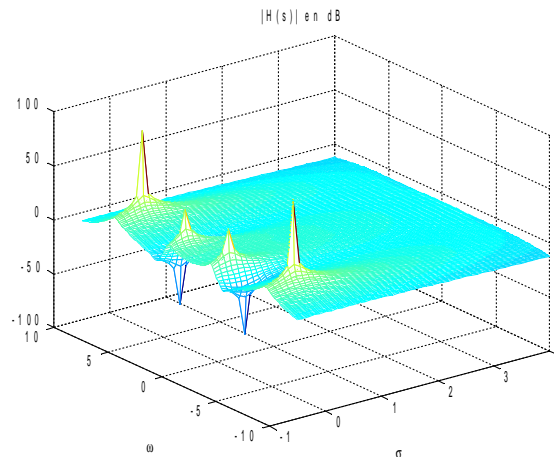
minimo=fminsearch(@pmat2f1,[-0.1,2])
h_en_min=pmat2f1(minimo),pause
maximo1=fminsearch(@pmat2f1n,[-0.2,3])
h_en_max_1=pmat2f1(maximo1),pause
maximo2=fminsearch(@pmat2f1n,[-0.002,9])
h_en_max_2=pmat2f1(maximo2),pause

function [hdb]=pmat2f1(in)
% |H(s)| en dB
s=in(1);
w=in(2);
num=(196/9)*[1 0.2 9];den=conv([1 0.0002 49],[1 0.4 4]);
hdb=20*log10(abs(polyval(num,s+j*w)./polyval(den,s+j*w)));

function [hdb]=pmat2f1n(in)
% |1/H(s)| en dB
s=in(1);
w=in(2);
num=(196/9)*[1 0.2 9];den=conv([1 0.0002 49],[1 0.4 4]);
hdb=-20*log10(abs(polyval(num,s+j*w)./polyval(den,s+j*w)));

```

When we run the function, the result is:



```

minimo =
    -0.1000    2.9983
h_en_min =
   -317.0491
Exiting: Maximum number of function evaluations has been exceeded
- increase MaxFunEvals option.
Current function value: -309.711370

maximo1 =
    -0.0001    7.0000
h_en_max_1 =

```



```

309.7114
maximo2 =
    -0.0001    7.0000
h_en_max_2 =
    306.8104

```

In the case of nested functions, these can be declared everywhere inside the function file but it is mandatory to close them with an **end**. The main function must be closed with an **end** too.

```

function main
    ...
    function nested
    ...
    end
    ...
end

```

Local functions (inline functions): You can create local functions in the command window, in scripts and in functions. The advantage is that there is no need to store them in a separate file, but they have limitations (they cannot be nested and they have only one output argument).

Example 6. Local functions (**inline**)

If we want to compute the area of a standard Gaussian bell between $\pm\sigma$, $\pm2\sigma$, and $\pm3\sigma$, we can do the following:

```

f=inline('normpdf(x,0,1)','x') %f is the normal pdf with sigma=1
f =
    Inline function:
    f(x) = normpdf(x,0,1)

a=quad(f,-1,1)
a =
    0.6827

a=quad(f,-2,2)
a =
    0.9545

a=quad(f,-3,3)
a =
    0.9973

```

Local functions can be also created by using the symbol `@` instead of the **inline** function. For instance, consider the following function

```
>> f = @(x) x.^2-3
f =
    @(x) x.^2-3

>> f(2)
ans =
    1
```

We can obtain its zero by means of **fzero**. As first input argument, we can refer to the existing function **f** or directly introduce its expression as a local function:

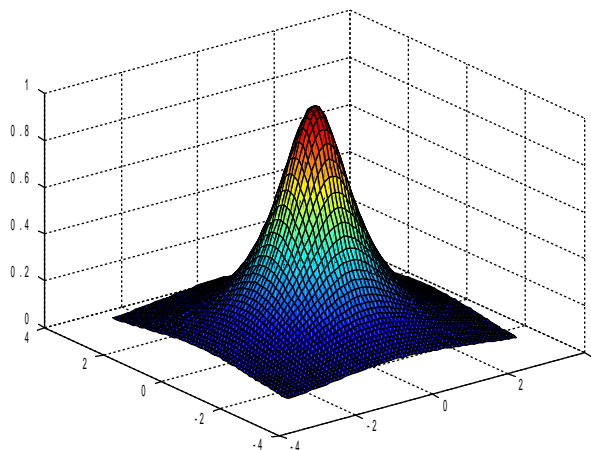
```
>> fzero(@(x) x.^2-3,0.2)
ans =
   -1.7321

>> fzero(f,0.2)
ans =
   -1.7321
```

Finally, inline functions can also present several input arguments:

```
>> g=@(x,y) 1./(1+x.^2+y.^2)
g =
    @(x,y) 1./(1+x.^2+y.^2)

>> [x,y]=meshgrid(-3:.1:3); surf(x,y,g(x,y));
```



Private functions: These are functions that are within the `<private>` directories. They can only be seen and called by functions and scripts that are in the directory immediately above the `<private>` directory. Since they are available only for a few functions, their name can be the same as other standard functions of MATLAB, for example, `bode.m`. When from the directory immediately above to `<private>` we call the `bode` function, then `bode.m` of `<private>` is executed instead of standard `bode.m`.

3. Programming language

3.1 Input and output commands

Input and output commands allow the communication with the user. Look the help of the following commands and try the different available options.

Input from keyboard: Use the function **input**

```
>> x=input('Enter a number: ')
Enter a number: 7

x =
    7

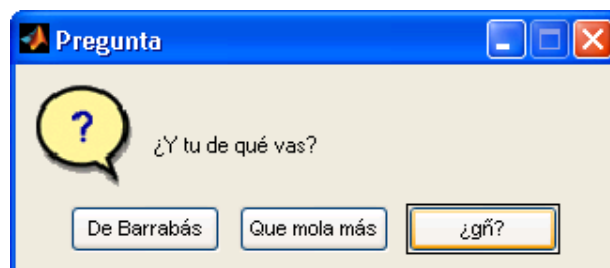
>> y=input('Enter a text:', 's')
Enter a text: hello

y =
hello
>>
```

(Notice that in the workspace, **y** is a **char array (string)** of dimension 1x5 and 8 bytes. This information can be accessed directly from the workspace or from the command window by typing **>>whos**)

Input from mouse: It is possible to open a dialog box with **questdlg**,

```
respuesta=questdlg('¿Y tu de qué vas?', 'Pregunta', ...
'De Barrabás', 'Que mola más', '¿gñ?', '¿gñ?')
```



(The default answer button is the last input argument, see the function help for more details)

Output to screen: Functions **disp** and **sprintf** display text and results in the command window.

disp	displays character string data
sprintf	displays data in a chosen format

disp can be used along with the following format conversion commands:

- **num2str** (*number-to-string*)
- **int2str** (*integer-to-string*)
- **mat2str** (*matrix-to-string*)
- **poly2str** (*polynomial-to-string*)
- **lab2str** (*label-to-string*)
- **rats** (*rational approximation*)

```
>> disp(['polynomial=' poly2str([3 4 5],'x')])
polynomial= 3 x^2 + 4 x + 5

>> mat2str(ones(2))
ans =
[1 1;1 1]

>> rats(3.4)
ans =
17/5
```

sprintf has many options: \n, \r, \t, \b, \f, \l, %, %, ... and formats: %d, %i, %o, %u, %x, %X, %f, %e, %E, %g, %G, %c, %s

```
>> sprintf('%0.2f', (1+sqrt(5))/2)
ans =
1.62

>> sprintf('Result is %0.5f', (1+sqrt(5))/2)
ans =
Result is 1.61803

>>
```

It is recommended to use the **help** command for the previous functions and explore the different options.

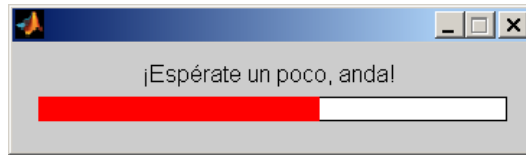
Other formats: Other useful possibilities for data labeling are:

```
>> title('\omega_n')
>> xlabel('\Theta_n')
>> text(0.2,0.2, '\it \phi^n')
```

In the example, the **text** command writes ϕ^n in the (0.2,0.2) coordinates of the last figure selected. Type **>>help LaTeX** for more information.

Wait bars: For instance:

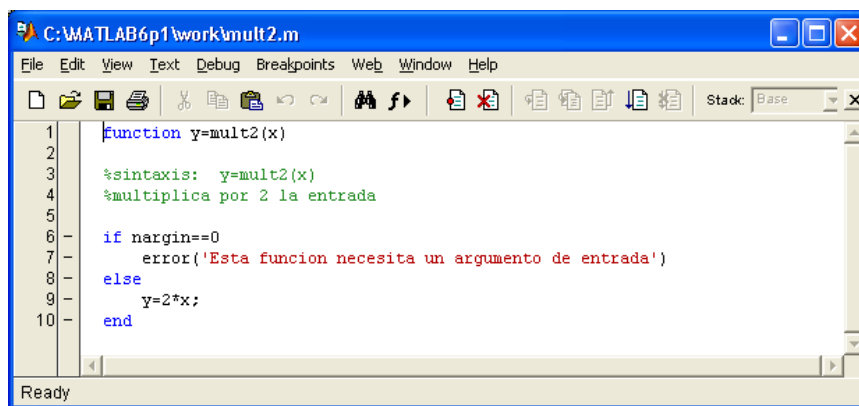
```
h=waitbar(0,'¡Espérate un poco, anda!');
for i=1:5,
    pause(1),waitbar(i/5,h);
end,
pause(1),close(h)
```



Error messages: Error messages are produced by means of the **error** function. See next example.

Example 7. Error messages

Consider the function called **mult2.m**



If we call function **mult2** without specifying any input argument, the result is:

```

>> z=mult2
??? Error using ==> mult2
Esta funcion necesita un argumento de entrada
>>
  
```

Note: **nargin** (number of input arguments) gives the number of input arguments the function has been called.

3.2 Indexing

Indexes: First index in MATLAB is “1”.

x	=	[0.3	0.2	0.5	0.1	0.6	0.4	0.4];
Index:		<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>

Use parentheses, (), to access the elements of vectors and matrices.

Check that if you type **>>x(0)**, MATLAB gives the following error message:

```
>> x(0)
??? Subscript indices must either be real positive integers
or logicals.

>> x(5)
ans =
    0.6
```

Function “find”: Use this function to find vector elements that satisfy a given Boolean condition. For instance,

```
>> x=[0.3 0.2 0.5 0.1 0.6 0.4 0.4];

>> i=find(x==0.9)
i =
     []

>> i=find(x>0.4)
i =
     3     5
```

Note: This function also works with matrices: `[i,j]=find(A~=0)` .

Other useful functions are **sort**, **rot90**, **flipud**, and **fliplr** (**fliplr(x)** is equivalent to **rot90(x,2)**).

Multidimensional arrays. They are an extension of matrices. A matrix is a two dimension array consisting of rows and columns. A three dimension array consists of rows, columns and pages. Dimensions equal or greater than four do not have a particular name. Next, we illustrate a three dimension array consisting of 2 rows, 2 columns and 3 pages:

```
>> A=[1 2;3 4];
>> A(:,:,2)=eye(2);
>> A(:,:,3)=eye(2)*2;
>> A
A(:,:,1) =
     1     2
     3     4
A(:,:,2) =
     1     0
     0     1
A(:,:,3) =
     2     0
     0     2
```

Function “squeeze”: This function removes singleton dimensions in multidimensional arrays. To illustrate the usage, consider the following *Control Systems Toolbox* example: When the function **bode** is applied to transfer function (**tf**) objects, the results **mag** and **phase** are three dimension arrays. If we want to extract only the magnitude values we can use the function **squeeze** to eliminate the singleton dimensions:

```
>> G=tf(1,[1 1]);
>> [mag,phase]=bode(G);
>> size(mag)
ans =
     1     1    45
>> mag=squeeze(mag);
>> size(mag)
ans =
    45     1
```

“Cell” and “struct” data types: Sometimes we want to store data of different types and/or dimensions in an only one variable. In this case it is useful to consider **cell** and **struct** data types:

Example with **cell**:

```
>> cosas_varias=cell(1,3)
%we create the cell variable of dimension 1 x 3
cosas_varias =
     []     []     []

>> cosas_varias{1}=1
%to access any cell element use the symbol {   }
cosas_varias =
     [1]     []     []

>> cosas_varias{2}=eye(2)
cosas_varias =
     [1]    [2x2 double]     []

>> cosas_varias{3}=eye(3)
cosas_varias =
     [1]    [2x2 double]    [3x3 double]

>> cosas_varias{3}
ans =
     1     0     0
     0     1     0
     0     0     1
```

Example with **struct**:

```
>> mis_cosillas.cosa1='hola'
mis_cosillas =
    cosa1: 'hola'

>> mis_cosillas.cosa2=1:5
mis_cosillas =
    cosa1: 'hola'
    cosa2: [1 2 3 4 5]
```

```
>> mis_cosillas.cosa3=zeros(3)

mis_cosillas =

    cosa1: 'hola'
    cosa2: [1 2 3 4 5]
    cosa3: [3x3 double]

>> mis_cosillas.mas_cosas.cosa4=8.2

mis_cosillas =

    cosa1: 'hola'
    cosa2: [1 2 3 4 5]
    cosa3: [3x3 double]
    mas_cosas: [1x1 struct]
```

Date format, “datestr” and “datenum”: MATLAB handles dates in a numeric format. Date 1 corresponds to the year 0, January 1st.

The function **datestr** shows the date in a char array format:

```
>> datestr(1)
ans =
01-Jan-0000
```

The function **datenum** convert a char array date in a number date. The second input argument specifies the date format (for example “dd/mm/yy” or “mm/dd/yy”).

```
>> datenum('1/2/07')
ans =
    733044

>> datestr(ans)
ans =
02-Jan-2007

>> datenum('1/2/07','dd/mm/yy')
ans =
    733074

>> datestr(ans)
ans =
01-Feb-2007
```

Other useful functions are **now** and **today**.

```
>> now
ans =
    7.3552e+05

>> datestr(ans)
ans =
08-Oct-2013 16:51:55
```


3.3 Flow control

There are many flow control statements in MATLAB:

for...end: It executes a group of statements a fixed number of times. The syntax is the following:

```
>>for index=start:increment:end,
    statements,
end
```

```
>> for i=1:3
    disp('bah!')
end
bah!
bah!
bah!
>>
```

The default increment is **1**. You can specify any increment, including negative ones. For example, **i=0:0.1:7** or **m=-5:-0.1:0**. For positive indexes, execution ends when the value of the index exceeds the end value. It is possible to nest multiple loops. A useful function when programming loops is the command **length**, for example: **i=1:length(x)**.

Example 8. FOR loop. Trajectories in a state plane

The following system is described by means its state equations:

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ -3 & -4 \end{bmatrix} \mathbf{x} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u$$

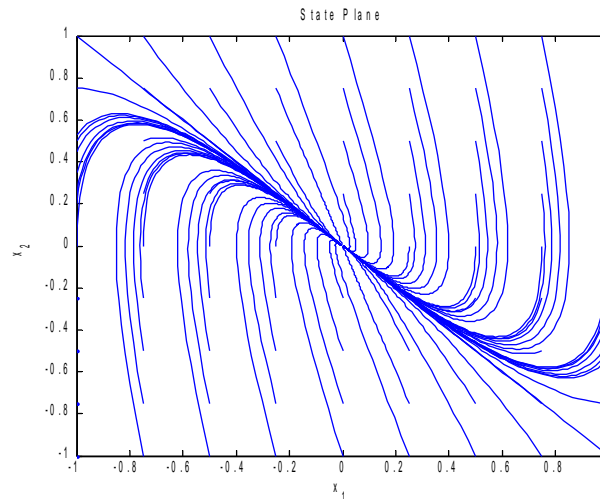
$$y = (2 \ 1) \mathbf{x} + 0 \cdot u$$

If we want to plot the response to different values for the initial conditions $\mathbf{x}(0)$, we can use a for...end loop for each state variable.

```
a=[0 1;-3 -4];b=[0;1];c=[2 1];d=0;
G=ss(a,b,c,d);
```

```
for x1=-1:0.25:1
    for x2=-1:0.25:1
        [y,t,x]=initial(G,[x1;x2]);
        plot(x(:,1),x(:,2)),hold on
    end
end
```

```
axis([-1 1 -1 1]),
title('State Plane')
xlabel('x_1'),ylabel('x_2')
```



Before running a **for** loop is recommended to initialize variables with the **zeros** function to optimize the performance (if not, MATLAB dynamically allocate memory for the variables of the loop, thereby incurring overhead).

while...end: It executes a group of statements a definite number of times, based on some logical conditions. The syntax is the following:

```
>>while (logical_expression),
    statements,
end
```

```
>> i=3;
>> while gt(i,1)
i=i-1;
disp('burp')
end
burp
burp
>>
```

Relational operators: equal to (**==**), less than (**<**), greater than (**>**), greater than or equal to (**>=**), less than or equal to (**<=**), not equal to (**~=**). There exist the corresponding functions: **eq** (*equal*), **ne** (*not equal*), **ge** (*greater than or equal*), **lt** (*less than*),...

It is usual to specify logical expressions between parentheses, (**x==2**)

To get more information, you can see the help for the dot “.”, that is: **>>help .**

Example 9. Relational operators

We want to evaluate the following expression [mag,05]

$$f(x) = \begin{cases} e^{-x/2} & a \leq x < b \\ 0 & \text{otherwise} \end{cases}$$

for $a=-1$ and $b=2$:

```
>> a=-1;b=2;
>> x=-3:3;
>> g=exp(x/2).*(a<=x & x<b)
g =
      0      0    0.6065    1.0000    1.6487      0      0
```

The Boolean condition is:

```
>> (a<=x & x<b)
ans =
      0      0      1      1      1      0      0
```

Logical operators: **and** (&), **or** (|), **not** (~), **xor** (xor(x,y))

Other useful functions: **isempty** (is empty?), **ischar**, **isnan**, **isinf**, **isfinite**, **isglobal**, **strcmp** (string comparison)... Value 1 means TRUE and value 0 means FALSE.

```
>> x=[]
x =
     []

>> isempty(x)
ans =
      1

>>
>> y='hello'
y =
hello

>> strcmp(x,'bye')
ans =
      0
```

Other useful commands:

To abort the execution:

break	exits the loop. Kills the script execution
return	returns to keyboard or to the function that has called the loop

To round numbers:

ceil	ceil(1.4)=2	ceil(-1.4)=-1
floor	floor(1.4)=1	floor(-1.4)=-2
fix	fix(1.4)=1	fix(-1.4)=-1
round	round(1.4)=1	round(-1.4)=-1

3.4 Conditional structures

if_elseif_else_end: This control statement executes a group of statements based on some logical conditions. The syntax is:

```
if (logical_expression)
    statements
elseif (logical_expression)
    statements
else
    statements
end
```

switch_case_otherwise_end: This statement executes different groups of statements depending on the value of some logical condition.

For instance, in the next example the variable **hello** can have the following values: **'yes'**, **'no'** or **'maybe'**:

```
switch hello
case {'yes','maybe'}
    statements
case 'no'
    statements
otherwise
    statements
end
```

3.5 Vectorization

In most situations MATLAB extends the definition of operators and functions from scalars to vectors or matrices. This is what we call 'vectorization'. As an example, one can use a vector or a matrix as input to the mathematical function **sin**, and the computation is automatically performed elementwise. This usage of vectors or matrices results in a drastical speedup of the computations, mainly due to the fact that the previous loop flow control structures are very flexible but inefficient, while the internal loops generated by vectorization are executed as fast as in low level compiled languages like C or FORTRAN.

The vectorization techniques provided by MATLAB are really powerful and flexible. A part of the above mentioned vectorization of most mathematical functions, using vectors or matrices as indexes to other vectors or matrices can often remove the need of **for** or

while flow control constructions, specially when the order of the iterations is irrelevant. Let us show some examples:

To select the odd positions of a vector **v** one can write **v(1:2:end)**, both in the left or in the right handside of an assignment expression. For instance,

```
>> v=[2,5,3,7,-2,-5,1];
>> v(1:2:end)
ans =
     2     3    -2     1

>> v(1:2:end) = 0
v =
     0     5     0     7     0    -5     0
```

Vectorization can be also used to permute or replicate elements in a vector:

```
>> v=[2,5,3,7,-2,-5,1];
>> v(end:-1:1)
ans =
     1    -5    -2     7     3     5     2

>> v([2,2,2,3,3,3])
ans =
     5     5     5     3     3     3
```

and the same functionality applies for rows and columns of matrices

```
>> M=[4,2,5;1,0,-2];
>> M([1,1,2], :)
ans =
     4     2     5
     4     2     5
     1     0    -2
```

Now, suppose that we have a vector **v** of length **n** and we want to insert the midpoint between every two consecutive elements in a new vector **w**. Two lines of code solve the problem, one line to copy **v** in the odd positions of **w** and another one to compute the midpoints and write them into the even positions of **w**:

```
>> v=[0.12,0.64,0.82,0.90,0.92]; n=length(v);
>> w(1:2:2*n-1) = v
w =
    0.12     0    0.64     0    0.82     0    0.90     0    0.92

>> w(2:2:end-1) = .5*(v(1:end-1)+v(2:end))
w =
    0.12    0.38    0.64    0.73    0.82    0.86    0.90    0.91    0.92
```