

2013

## A MATLAB Primer in Four Hours with Practical Examples

Jerome Casey  
jerome.casey@tudublin.ie

Follow this and additional works at: <https://arrow.tudublin.ie/schmuldissoft>

 Part of the [Computer Sciences Commons](#), and the [Engineering Commons](#)

---

### Recommended Citation

Casey, J. (2013). A MATLAB primer in four hours with practical examples. Software guide for undergraduate students. Technological University Dublin

This Other is brought to you for free and open access by the School of Multidisciplinary Technologies at ARROW@TU Dublin. It has been accepted for inclusion in Instructional Guides by an authorized administrator of ARROW@TU Dublin. For more information, please contact [arrow.admin@tudublin.ie](mailto:arrow.admin@tudublin.ie), [aisling.coyne@tudublin.ie](mailto:aisling.coyne@tudublin.ie).



This work is licensed under a [Creative Commons Attribution-Noncommercial-Share Alike 4.0 License](#)

# **A MATLAB Primer in 4 Hours with Practical Examples**

## **Engineering Computing**

DT023-1, DT025-1

Jerome Casey

## Table Of Contents

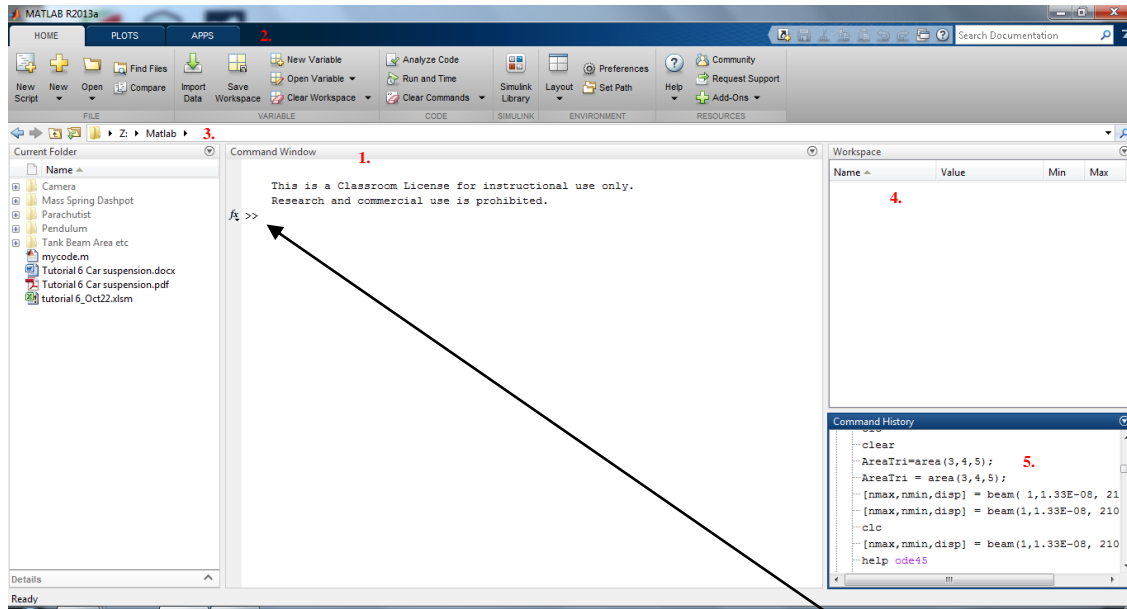
<b>MATLAB EXERCISE 1 .....</b>	<b>1</b>
INTRODUCTION TO THE MATLAB INTERFACE .....	1
<b>MATLAB EXERCISE 2 .....</b>	<b>2</b>
NUMERICAL METHODS: WORKING WITH MATRICES .....	2
<b>MATLAB EXERCISE 3 .....</b>	<b>7</b>
ENTERING & PROCESSING DATA IN MATLAB, SAVING DATA TO EXTERNAL FILES .....	7
<b>MATLAB EXERCISE 4 .....</b>	<b>10</b>
READING FROM EXTERNAL FILES / WORKING WITH PLOTS: LINE COLORS & STYLES, TITLES .....	10
<b>MATLAB EXERCISE 5 .....</b>	<b>13</b>
WORKING WITH SUBPLOTS .....	13
<b>MATLAB EXERCISE 6 .....</b>	<b>14</b>
LOADING AND WRITING RESULTS TO EXTERNAL FILES: LOAD & SAVE COMMANDS, VARIABLE EDITOR ...	14
<b>MATLAB EXERCISE 7 .....</b>	<b>16</b>
WRITING SCRIPTS & FUNCTIONS .....	16

## **Matlab Exercise 1**

### Introduction to the Matlab Interface – Setting the Current Folder

“MATLAB” stands for “Matrix Laboratory.” It is an interactive software program for performing numerical computations. It was initially designed by Cleve Moler in the 1970s for use as a teaching tool, but it has since become a very successful commercial package produced by The MathWorks Ltd.<sup>1</sup> One of MATLAB’s best features, from the point of view of the computational scientist, is its large built-in library of numerical routines and graphical visualisation tools.

Click on the desktop shortcut to open [MatlabR2013a](#). The main window is the MATLAB **Command Window**, where you write your instructions.



In the MATLAB **Command Window**, Matlab will execute the instructions after the **Command prompt** `>>` when you press ‘Enter’ on your keyboard. You can also use the arrow key `↑` to cycle back through previously entered commands.

Set the desktop layout to the default setting by selecting **Desktop > Desktop Layout > Default**.

**Note:** Create a Matlab folder in your Computing folder of your Home folder and set the **Current Folder** (in 3. above) to where you will save your files such as: `z:\Computing\Matlab`

In these exercises you will cover:

- The MATLAB Desktop and Desktop Tools including the *Command Window*, the *Launch Pad*, the *Help Browser*, the *Current Folder Browser* and the *Editor/Debugger*.
- Matrices in MATLAB: Generating Matrices, Sum, Transpose, Subscripts, concatenation, deleting rows and columns and the Colon Operator. Expressions: Variables, Numbers, Operators.
- Scripts & Functions, the Load command, M-Files.
- Demonstration of Numerical methods.

<sup>1</sup> see [www.mathworks.com](http://www.mathworks.com) for more examples and help on code.

## **Matlab Exercise 2**

### Numerical Methods: Working with Matrices

#### Creating Matrices:

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional rectangular array of real or complex numbers that represents a linear transformation. The linear algebraic operations defined on matrices have found applications in a wide variety of technical fields.

MATLAB has dozens of functions that create different kinds of matrices. Two of them can be used to create a pair of 3-by-3 example matrices. The first example matrix is **symmetric**:

```
>> A = pascal(3)
```

```
A =
     1     1     1
     1     2     3
     1     3     6
```

The second example is not symmetric:

```
>> B = magic(3)
```

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

The command `>> sum(B)` sums the elements in each column to produce:

```
15 15 15
```

The command `>> sum(B)'` transposes the matrix, sums the columns of the transpose, and then transposes the results to produce the row sums:

```
15
15
15
```

The command `>> sum(diag(B))` sums the **main diagonal** of B, which runs from the upper left element to the lower right element, to produce 15

An **m x n** matrix is a rectangular array of numbers having m rows and n columns. A **column vector** is an **m-by-1** matrix, a **row vector** is a **1-by-n** matrix and a **scalar** is a **1-by-1** matrix. Enter the following statements:

<code>&gt;&gt; col = [3; 1; 4]</code>	produces	a <b>column</b> vector, <code>&gt;&gt; col = 3</code> 1 4
<code>&gt;&gt; row = [2 0 -1]</code>	produces	a <b>row</b> vector, <code>&gt;&gt; row = 2   0   -1</code>
<code>&gt;&gt; s = 7</code>	produces	a <b>scalar</b> : <code>&gt;&gt; s =</code> 7

**Note** the square brackets create a matrix; rows are delineated by a semi-colon and columns by a space.

#### Accessing Matrix Elements:

A 2D matrix is specified first by rows and then by columns.

```
>> col1 = A(:, 1)   % select all row entries for column 1
```

```
col1 =
     1
     1
     1
```

```
>> elem1 = A(2, 2)   % select element at row 2 and column 2
```

```
elem1 =
     2
```

**Note** the `:` operator denotes 'all'.

#### Info:

An **nxn** matrix that has the same number of rows and columns is called a **square** matrix.

A matrix is said to be **symmetric** if it is equal to its transpose (i.e. it is unchanged by transposition).

**Adding and Subtracting Matrices:**

Addition and subtraction of matrices is defined just as it is for arrays, element-by-element. To add 2 matrices together they must have the same dimensions:

```
>> C = A + B
```

```
C =
    9    2    7
    4    7   10
    5   12    8
```

Addition and subtraction require both matrices to have **the same dimension**, or one of them to be a **scalar**. If the dimensions are incompatible you will get an error. Add a scalar to each element of a matrix:

```
>> D = C + 1
```

```
D =
   10    3    8
    5    8   11
    6   13    9
```

**Vector Products and Transpose:**

For real matrices, the transpose operation interchanges  $a_{ij}$  and  $a_{ji}$ . In other words, transposing a vector changes it from a row to a column vector and vice versa. The extension of this idea to matrices is that transposing interchanges rows with the corresponding columns: the 1st row becomes the 1st column, and so on. MATLAB uses the apostrophe operator (') to perform a complex conjugate transpose, and the dot-apostrophe operator (.') to transpose without conjugation. For matrices containing all real elements, the two operators return the same result.

```
>> Z = B'
```

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

**Deleting Matrix Rows and Columns:**

```
>> F = magic(5) % create this 5 x 5 matrix
```

```
F =
   17   24    1    8   15
   23    5    7   14   16
    4    6   13   20   22
   10   12   19   21    3
   11   18   25    2    9
```

```
>> F(3,:)=[]; % delete row 3
>> F(:,5)=[]; % delete column 5
% F is now a 4x4 matrix
```

**Tip:**

The **semi-colon** suppresses the output to the Command Window.

Double-click on the variable F when it appears in the **Workspace** to open it in the **Variable Editor** and watch the values update as you enter the commands.

**Matrix Concatenation:**

```
>> Z = ones(4,1) % create this 4x1 matrix of ones
```

```
>> F = [F Z] % concatenate the 4x1 matrix to the end of the 4x4 matrix
```

**Matrix Multiplication:**

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is **not commutative**; AB is usually not equal to BA:

```
>> X = A*B, Y = B*A
```

```
X =
   15   15   15
   26   38   26
   41   70    3
```

```
Y =
   15   28   47
   15   34   60
   15   28   43
```

**Dot-Product of Matrices:**

Instead of doing a matrix multiply, we can multiply the corresponding elements of two matrices or vectors using the `.*` operator. The **dot-product** is also known as **element-wise multiplication**. The dot product works as for vectors: corresponding elements are multiplied together, thus the matrices involved must have the same size:

```
>> Dot = A .* B
```

```
Dot =
```

```
8   1   6
3  10  21
4  27  12
```

**Sparse Matrices:**

Sparse matrices are usually large matrices that have only a very small proportion of non-zero entries.

Create a sparse 5x4 matrix  $S$  having only 3 non-zero values:  $S_{1,2} = 10$ ,  $S_{3,3} = 11$  and  $S_{5,4} = 12$ .

First create 3 vectors containing the  $i^{\text{th}}$  index, the  $j^{\text{th}}$  index and the corresponding values of each term and then use the **sparse** command.

```
>>i = [1, 3, 5]; j = [2, 3, 4];
```

```
>>v = [10 11 12];
```

```
>>S = sparse(i,j,v)
```

```
S =
```

```
(1,2)    10
(3,3)    11
(5,4)    12
```

```
>> Sfull = full(S)
```

```
Sfull =
```

```
0  10  0  0
0   0  0  0
0   0  11 0
0   0  0  0
0   0  0  12
```

The matrix  $S_{full}$  is a "full" version of the sparse matrix  $S$ .

**Solving Simultaneous Equations:**

A general system of linear equations can be expressed in terms of a co-efficient matrix A, a right-handside (column) vector b and an unknown (column) vector x as follows:

$$\mathbf{A} * \mathbf{x} = \mathbf{b}$$

or, component wise, as:

$$a_{1,1}x_1 + a_{1,2}x_2 + \dots a_{1,n}x_n = b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \dots a_{2,n}x_n = b_2$$

...

$$a_{n,1}x_1 + a_{n,2}x_2 + \dots a_{n,n}x_n = b_n$$

The solution is obtained by  $\mathbf{x} = \mathbf{A}^{-1} * \mathbf{b}$

Given

$$3v - 3w + 6x - 2y + z = 14$$

$$3v - 6w + x - y + z = 25$$

$$2v - 4w + 4x - 4y + 3z = 5$$

$$3v - 6w + 5x - y + 2z = 30$$

$$2v - 4w + 9x + y + z = 30$$

Solve for v, w, x, y and z using the matrix functions

Enter the following in the **Command Editor**:

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

```
>> b = [14; 25; 5; 30; 30];
```

```
>> x = inv(A)*b (Though formally correct >> x = A \ b is generally used in Matlab)
```

```
x =
    6.3333
   -1.3333
         0
    7.0000
    5.0000
```

The inverse looks better if it is displayed with a rational format.

```
>> format rat
```

```
>> x
```

```
x =
    19/3
    -4/3
         0
         7
         5
```

The following statement restores the output format to its default.

```
>> format short
```

**Diagonal Matrices:**

A diagonal matrix contains non zero diagonal entries with zeros everywhere else. D is a 3x3 diagonal matrix. To construct this in Matlab, we could type it in directly as follows:

```
>> D = [-3 0 0; 0 4 0; 0 0 2]
```

```
D =
   -3    0    0
    0    4    0
    0    0    2
```

However this becomes impractical when the dimension is large (e.g. a 100 x100 diagonal matrix). We then use the **diag** function. Create a row vector d, say, containing the values of the diagonal entries (in order) then diag(d) gives the required matrix:

```
>> d = [-3 4 2], D = diag(d)
```



On the other hand, if A is any matrix, the command `diag(A)` extracts its diagonal entries:

```
>> H= diag(A)
```

```
H =
     3
    -6
     4
    -1
     1
```

	1	2	3	4	5
1	3	-3	6	-2	1
2	3	-6	1	-1	1
3	2	-4	4	-4	3
4	3	-6	5	-1	2
5	2	-4	9	1	1

### Initialising Matrices:

The functions **ones** and **zeros** are typically used to initialise a matrix - the size of which you may already know, but the values which may change during processing. Pre-allocation of memory by initialising a matrix allows for more efficient programs.

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

```
P =
     1     1     1
     1     1     1
```

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

```
Q =
     0     0
     0     0
     0     0
```

```
>> R= ones(size(C))
```

```
R =
     1
     1
     1
     1
     1
```

This example shows how we can construct a matrix based on the size of an existing one.

### The Identity Matrix:

The  $n \times n$  identity matrix is a matrix of zeros except for having ones along its leading diagonal (top left to bottom right).

```
>> I = eye(3)
```

```
I =
     1     0     0
     0     1     0
     0     0     1
```

This is called **eye(n)** in Matlab since mathematically it is usually denoted by **I**.

### Using Nested For Loops in 2 Dimensional Matrices:

A nested for loop is typically used to cycle through the entries of a 2 dimensional ( $m \times n$ ) matrix in so called raster order. The outer for loop controls the movement along the rows, beginning at the first row, whilst the inner loop controls the movement along the columns. The following code uses the matrix A created previously:

```
% update the diagonal elements to 10
[numRows,numCols] = size(A); % A was created above
for i = 1: numRows
    for j = 1: numCols
        if i==j
            A(i , j) =10;
        end
    end
end
```

Other important quantities in computational linear algebra that have functions in Matlab are matrix norms, trace, rank, condition number, eigenvalues and singular values.

### **Matlab Exercise 3**

#### **Entering & Processing Data in Matlab, Saving Data to External Files**

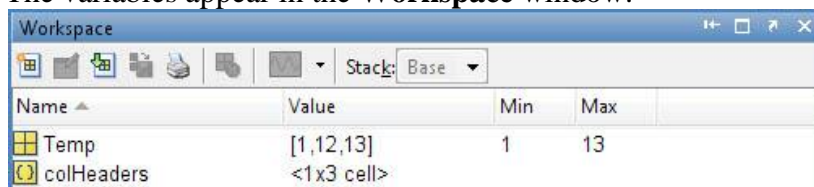
The basic arithmetic operators are  $+$   $-$   $*$   $/$   $^$  and these are used in conjunction with brackets  $()$ .

#### **Entering Data:**

This task is to enter Temperature data <sup>2</sup> for the first 10 days of the Month of April for various locations in the country and carry out some processing of the data. First create the variables:

```
>> colHeaders = {'April' 'Dublin' 'Kilkenny'};
>> Temp = [1,12,13];
```

The variables appear in the **Workspace** window:



#### Method 1: - Entering Data via the Variable Editor

The **Variable Editor** allows you to view array contents in a table format and edit the values. Open the **Temp** variable that you just created in the **Variable Editor** by double clicking on it in the **Workspace** pane and continue entering the following data:

1	12	13
2	15	14
3	22	20
4	22	20
5	23	21
6	22	22
7	21	21
8	20	19
9	15	16
10	12	14

#### Method 2 - Entering Data by importing from an External file:

Delete the **Temp** variable from the workspace.

Create an Excel file with the same data above laid out in 3 columns. Search the help to see how to import the numerical data into MATLAB from Excel and assign it to the **Temp** variable.

#### **Processing Data in Matlab:**

Calculate the following:

- the average temperature over the 10 days for a location
- the average temperature on each day for both locations

You can process data in Matlab by calling on a myriad of built-in functions. Later you will also create your own functions.

<sup>2</sup> Met Éireann use Matlab to process data, see <http://www.met.ie/climate-ireland/rainfall.asp> and the R package [http://www.met.ie/climate/dataproducts/Estimation-of-Point-Rainfall-Frequencies\\_TN61.pdf](http://www.met.ie/climate/dataproducts/Estimation-of-Point-Rainfall-Frequencies_TN61.pdf)

Enter the following at the **Command** prompt:

```
>> avgTempLocation1 = mean(Temp(:,2))
```

```
avgTempLocation1 =  
18.4000
```

```
>> avgTempLocation2 = mean(Temp(:,3))
```

```
avgTempLocation2 =  
18
```

```
>> avgTemp = mean(Temp(:,2:3),2)
```

```
avgTemp =  
12.5000  
14.5000  
21.0000  
21.0000  
22.0000  
22.0000  
21.0000  
19.5000  
15.5000  
13.0000
```

#### Notes:

- The built in **mean** function was called to calculate the average (mean).
- Temp is actually an array variable, similar to a matrix with 10 rows and 3 columns.
- A matrix is referenced by specifying the row reference followed by the column reference, separated by a comma. e.g. Temp(:,2) accesses all the rows of column 2, i.e. column 2. The colon operator **:** is shorthand for **all**.
- The various calls to mean here calculate the average of a column or the average across a row. Can you see which is which? Check the help to see the various forms of the mean function.

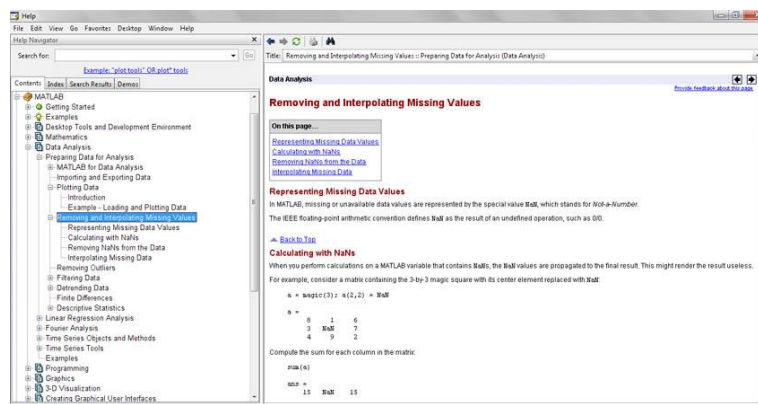
#### Help Browser:

For Help on Matlab functions enter the following after the command prompt and it will print within the pane:

```
>>help mean
```

Alternatively select **Help/Product Help** and type in the query in the *Search for* box. e.g. type in *mean* and hit return. The help returns info on the *mean* function.

**Take time to explore the Help. You can teach yourself Matlab from all the examples shown in the various sections. In particular have a look at the sections: Getting Started, Examples, Mathematics, Data Analysis, Programming, Graphics, Functions.**



**Saving Data Variables to External files:****Method 1:** - Saving to a **.mat** file

All the variables can be saved to a Matlab **.mat** file with the numerical values in matrix form and the text values in cell array form using the following command:

```
>> save temp Temp colHeaders avgTemp
```

The first value is the **filename** and all the other values are the **variables** you wish to save in the **.mat** file.

**Method 2:** - Saving to a Text File

All the variables can be saved to a **text file** using a space to delimit the data in column form as follows:

```
fName = 'Apr2013_test';

firstLine = '% Temperature Data in Celsius';
secondLine = '% Column Headers: ';

for j = 1:length(colHeaders)
    secondLine =[secondLine,colHeaders{1,j},' ']; % add a space between column headers
end

dlmwrite(fName,firstLine,'delimiter','');
dlmwrite(fName,secondLine,'-append','delimiter','');
dlmwrite(fName,Temp,'-append','delimiter',' ','precision',15); % use a space as the delimiter
```

**The code here is adaptable. The output file name fName could be adapted in code if you decide to process a number of different months separately and then save them as different files.**

Check that the text file 'Apr2013\_test' has been created in your current folder.

Now clear all variables from the **Workspace**:

```
>> clear
```

Clear the screen of your **Command Window**:

```
>> clc
```

The next time you load the **.mat** file in Matlab all saved variables will load into the workspace. See how this is done in the next exercise.

**The Editor Window:**

Select the lines of code in the **Command History**. Right-click and select **Create Script**. The selected lines of code are transferred to the **Editor** window where you can edit the code and save it as an **.m file** (a file containing MATLAB statements).

You can also reuse commands from the **Command History** by dragging them from the **Command History** and dropping them in the **Command Window**.

You will use the **Editor** window again to write Scripts and Functions.

### **Matlab Exercise 4**

#### Reading from External Files / Working with Plots: Line Colors & Styles, Titles

Create a bar chart plot of the temperatures at both locations from the previous exercise and add a line overlay of the average temperature. The following tables will be useful when creating and formatting the plots:

<code>plot(x,y,'r:')</code>	Will plot x,y in a <b>red dotted line</b>
<code>plot(x,y,'r', x,z,'gx')</code>	Will plot y in a <b>red solid line</b> (the default) and z in a <b>green line with crosses</b> as marker points
<code>axis([0, 1, 0, 30])</code>	Defines the x and y axis to <b>[0,1]</b> for x and <b>[0, 30]</b> for y

Graph Annotation	
<code>grid</code>	Adds dotted gridlines to the chart
<code>text</code>	Adds Text annotation at a location on the plot area
<code>title</code>	Adds the Graph title
<code>xlabel</code>	Adds x-axis label
<code>ylabel</code>	Adds y-axis label
<code>legend</code>	Adds a legend

The different colours and styles are:

Colours	
<b>y</b>	yellow
<b>m</b>	magenta
<b>c</b>	cyan
<b>r</b>	red
<b>g</b>	green
<b>b</b>	blue
<b>w</b>	white
<b>k</b>	black

Styles			
<b>.</b>	Dotted line	<b>d</b>	diamond
<b>o</b>	Circled line	<b>v</b>	triangle (down)
<b>x</b>	X Crosses line	<b>^</b>	triangle (up)
<b>+</b>	+ Crosses line	<b>&lt;</b>	triangle (left)
<b>-</b>	Solid	<b>&gt;</b>	triangle (right)
<b>*</b>	Stars line	<b>p</b>	pentagram
<b>:</b>	Dotted	<b>h</b>	hexagram
<b>-.</b>	Dash - Dot	<b>s</b>	Square
<b>--</b>	Dashed line		

#### Specifying the Color and Size of Markers:

You can also specify other line characteristics using graphics properties:

**LineWidth** — Specifies the width (in points) of the line.

**MarkerEdgeColor** — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

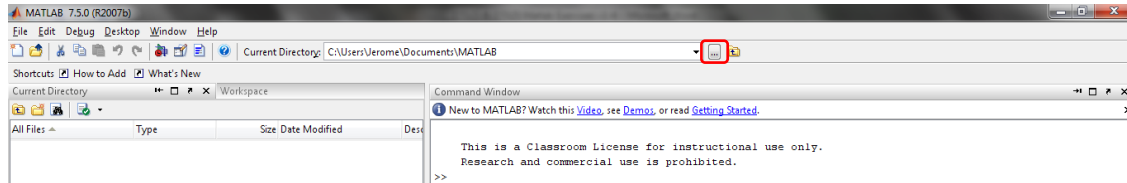
**MarkerFaceColor** — Specifies the color of the face of filled markers.

**MarkerSize** — Specifies the size of the marker in units of points.

To do this exercise you will need to have the variables used in the previous exercise Temp, avgTemp and colHeaders (stored in file `temp.mat`) is loaded in the **Workspace**.

### Loading Files and Variables:

MATLAB can only access files that are in its working path or in the “current Folder.” Ensure your Working Directory is set to where the file `temp.mat` is located, if not browse to where it is located.



Enter the following at the **Command** prompt:

```
>> load temp.mat
```

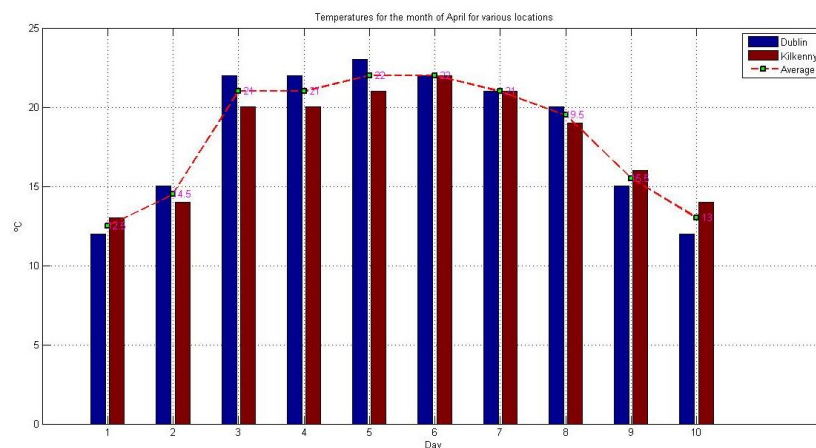
The **Workspace** should now contain the 3 variables.

Enter the following code at the **Command** prompt:

```
x = Temp(:,1);
y = Temp(:,2:end);

bar(x,y,'grouped');
grid;

title(['Temperatures for the month of ', colHeaders{1,1}, ' for various locations'])
xlabel('Day')
ylabel('°C')
hold on % holds the current picture -> used to achieve the overlay
plot(x,avgTemp,'--rs','LineWidth',2,... % red dashed line, square markers
      'MarkerEdgeColor','k',...
      'MarkerFaceColor','g',...
      'MarkerSize',5)
legend('Dublin','Kilkenny','Average');
text(x,avgTemp,num2str(avgTemp),'FontSize',10,'Color','m'); % places value at marker
```



The code above is designed to be adaptable for input files of the same layout being loaded with additional columns appended if necessary. The only line that needs to be recoded is that containing the `legend`. Update the file containing the April data by adding temperature data for another location. Edit the code and Run it again.

**Notes:**

- `y = Temp(:,2:end);` sets the y variable to be the entire block of data in Temp, excluding column 1. The value `end` allows flexibility in the code, if the user decides to add another column of temperature data in the previous data processing; this line will still be valid.
- `bar(x,y,'grouped');` This command creates a bar chart with each data series side by side or grouped. There are other formats of this function such as a stacked bar chart, changing its orientation to horizontal instead of vertical. See the Help for more info.
- `grid` This command plots gridlines on the chart. This is useful in identifying the value of a data point.
- the `title` function adds the chart title to the chart. In this example you want to use a mixture of text and the value from a variable. This is achieved by adding the square brackets within the function. `title(['Temperatures This line also gives us flexibility to set these lines up as a function where the month can be passed as a parameter if we want to process the months individually.])`
- `hold on` This command holds the current figure in place with the bar chart allowing the subsequent line chart to be plotted on the same window figure rather than in a new window.
- the `plot` function creates the chart. See the help for the many forms for this function.
- the `text` function is used to annotate the graph. Here the first 2 parameters identify the x and y location on the plot where you want the annotated text to be placed.

**Numeric Variables**

You can specify numeric variables in text strings using the `num2str` (number to string) function. For example, if you type on the command line

```
>>x = 21;
>> ['Today is the ',num2str(x),'st day.']
```

MATLAB concatenates the three separate strings into one.

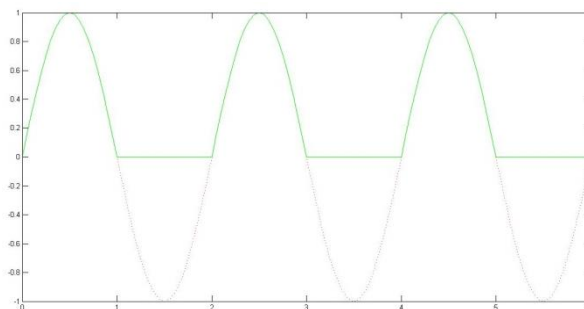
```
>>Today is the 21st day.
```

[See the Help on annotating](#)

**Masking Plots:**

Enter the following:

```
>>x = 0:0.05:6; y = sin(pi*x);
>>Y = (y>=0).*y; % see section on dot-product and element-wise multiplication
>>plot(x,y,'r:',x,Y,'g-')
```



## **Matlab Exercise 5**

### Working with Subplots

The subplot command divides the current figure (such as a graph plot) into rectangular panes that are numbered row-wise. Each pane contains an axes object. Subsequent plots are output to the current pane. For example, enter the following code that creates a figure with 8 subplots of 4 rows and 2 columns.

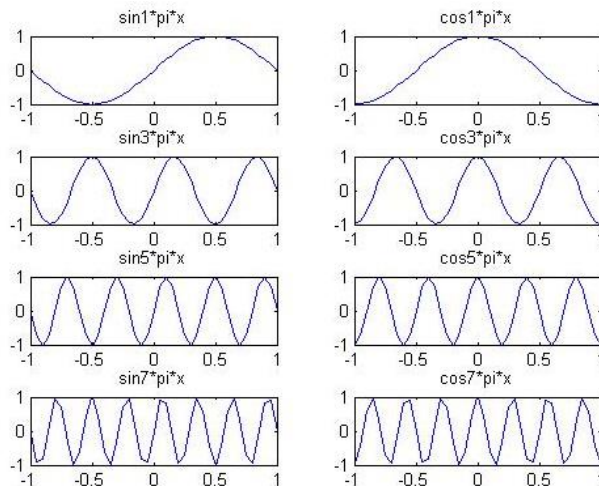
```
x = -1:.05:1;

for n = 1:2:8
    subplot(4,2,n), plot(x,sin(n*pi*x)),title(['sin',num2str(n),'*pi*x'])
    subplot(4,2,n+1), plot(x,cos(n*pi*x)),title(['cos',num2str(n),'*pi*x'])
end
```

Save the file as **subplots.m**

#### Keyboard-Accelerators:

You can recall previous Matlab commands by using the  $\uparrow$  and  $\downarrow$  cursor keys. Repeatedly pressing  $\uparrow$  will review the previous commands (most recent first) and, if you want to re-execute the command, simply press the **return** key. To recall the most recent command starting with **p**, say, type **p** at the prompt followed by  $\uparrow$ . Similarly, typing **pr** followed by  $\uparrow$  will recall the most recent command starting with **pr**. Once a command has been recalled, it may be edited (changed). You can use  $\leftarrow$  and  $\rightarrow$  to move backwards and forwards through the line, characters may be inserted by typing at the current cursor position or deleted using the **Del** key. This is most commonly used when long command lines have been mistyped or when you want to re-execute a command that is very similar to one used previously as above for the subplot command.



The following combinations produce asymmetrical arrangements of subplots.

```
subplot(2,2,1) % this subplot uses pane 1
subplot(2,2,3) % this subplot uses pane 3
subplot(2,2,[2 4]) % this subplot uses panes 2 and 4
```

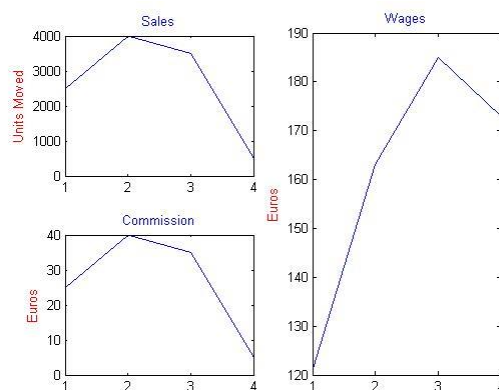
Enter the following code in the **Command** window:

```
unitsmoved = [2500 4000 3500 490];
hourlyrate = 30;
hoursworked = [3.2 4.1 5.0 5.6];
basic = hoursworked*hourlyrate;
commission = unitsmoved*0.01;
wages = basic + commission;

subplot(2,2,1); plot(unitsmoved)
title('Sales','color','b')
ylabel('Units Moved','color','r')

subplot(2,2,3); plot(commission)
title('Commission','color','b')
ylabel('Euros','color','r')

subplot(2,2,[2 4]); plot(wages)
title('Wages','color','b')
ylabel('Euros','color','r')
```





## **Matlab Exercise 6**

### **Loading and Writing Results to External Files: Load & Save Commands, Variable Editor**

Files required: [TempAnalysis.m](#) and [annual\\_temps.mat](#). Download these files to your current folder.

#### **File Importing and Exporting within MATLAB**

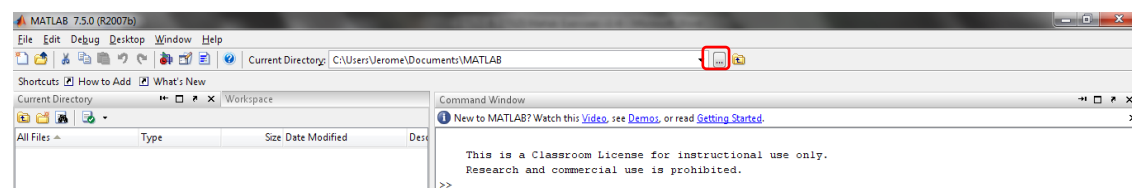
- MATLAB formatted data has the file extension **.mat**. These files are imported using the load command and exported using the save command as shown previously. Variables from the MATLAB workspace are saved.

- Text files are imported using the load command and exported using the save **-ascii** command.

For more see [http://www.mathworks.es/academia/student\\_center/tutorials/ps\\_solve/player.html](http://www.mathworks.es/academia/student_center/tutorials/ps_solve/player.html)

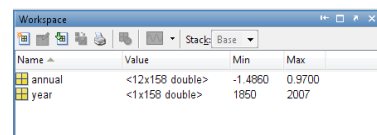
#### **Loading Files**

MATLAB can only access files that are in its working path or in the “Current Folder unless you specify the full filepath”. Ensure your Current Folder is where the files [annual\\_temps.mat](#) and [TempAnalysis.m](#) are located, if not browse to where they are located.

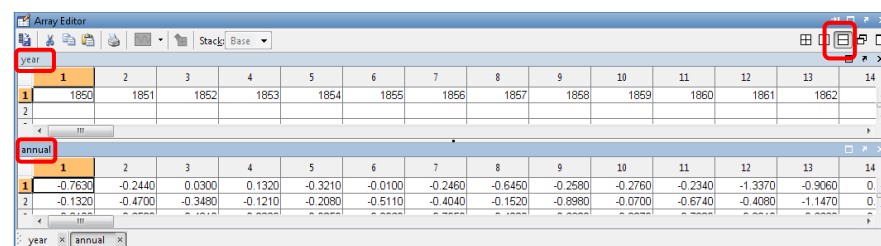


>> load [annual\\_temps.mat](#)

The **Workspace** should now contain the 2 variables as follows:



Double-click on both variables. The **Variable Editor** will open. To view the data as shown select the **Top/Bottom Tile** icon:



#### **Saving Files and Variables**

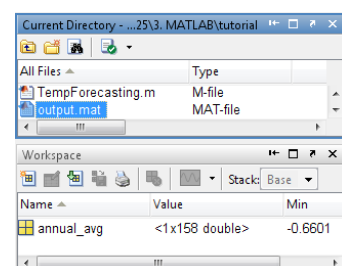
Run the script [TempAnalysis.m](#) by typing its name in the command prompt and hitting return.

The data [annual\\_avg](#) were calculated using this script as were other variables. To save only this variable in an external file type the following:

>> save [output](#) [annual\\_avg](#)

where [output](#) is the name of a new **.mat** file and [annual\\_avg](#) is the variable name of the data.

To check the file contents clear the **Workspace** and then double-click on the [output](#) file within the current Folder: [annual\\_avg](#) should be the only variable in the Workspace.



**Preloaded .mat files and Image Processing:**

Matlab comes preloaded with many **.mat** files that can be called using the load command without the file existing in the current folder.

Enter the following:

```
>>load durer % loads the file which contains the variables X, caption, map
>>image(X)
>>colormap(map)
```

This image command displays matrix X as an image within a figure window. In this example X is a 2-dimensional **m x n** matrix and each element of the matrix specifies the colour of a rectilinear patch in the image. Each 'pixel' is actually an index into a colormap called map which contains the actual value of the colour specified by an index. In this example there are 128 index values representing 128 different colours:



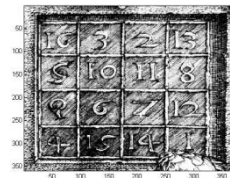
e.g. the color [0 0 0] represents black and [1 1 1] represents white. [1 0 0] is pure red, [.5 .5 .5] is gray. The 3 values represent the color intensities of Red, Green, and Blue light.

If all 3 values of R, G and B are the same you have a grayscale value such as in the **durer** image. If however you see a value in a colormap with differing values for RGB you have a color value: e.g. [127/255 1 212/255] is aquamarine.

Identify the magic square we met previously. To focus in on it in more detail load the following **.mat**:

```
>>load detail
```

Now run the image and colormap commands again to visualise the data.



There are a number of built-in colormaps which you can use.

For more info type:

```
>>help colormap or search the help to see some sample images.
```

The demos directory contains a CAT scan image of a human spine to which you can apply the built-in bone colormap. To view the image, type the following commands:

```
>>load spine
>>image(X)
>>colormap bone
```



Another preloaded example is the **.mat** file of the coastline around Cape Cod:

```
>>load cape
>>image(X)
>>colormap(map)
```

We will use this file again within a **script** in the next exercise. It does a real-time prediction of the rising sea levels over time and updates the image on screen.

## **Matlab Exercise 7**

### Writing Scripts & Functions

For a more efficient workflow such as repeating common tasks you can write your own scripts and functions. A script is a series of commands whereas a function is a series of commands that may accept arguments and return values.

You can enter commands one at a time at the MATLAB command line, or you can write a series of commands to a file that you then execute as you would any MATLAB function. Use the **Editor** or any other text editor to create your own function files. Call these functions as you would any other MATLAB function or command.

There are two kinds of program files:

1. **Scripts**, which do not accept input arguments or return output arguments. They operate on data in the workspace.
2. **Functions**, which can accept input arguments and return output arguments. Internal variables are local to the function.

If you are a new MATLAB programmer, just create the program files that you want to try out in the current folder. As you develop more of your own files, you will want to organize them into other folders and personal toolboxes that you can add to your MATLAB search path.

### Scripts

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. In addition, scripts can produce graphical output using functions like plot.

For example, open the **Editor** and create a file called `graph1.m` that contains these MATLAB commands or download the file from webcourses:

#### Example1 - Plots Using a Varying Number of Points (Graph Smoothing)

```
% Script file graph1.m

% Graph of the rational function y = x/(1+x^2).
for n=1:1:5
    n10 = 10*n;
    x = linspace(-2,2,n10);
    y = x./(1+x.^2);
    %plot(x,y,'r') % experiment with Graphs with different colors
    %plot(x,y,'g') % and comment out the ones you won't use
    plot(x,y,'b')

    title(sprintf('Graph %g. Plot based upon n = %g points.' ...
        ,n, n10))
    axis([-2,2,-.8,.8])
    xlabel('x')
    ylabel('y')
    grid
    pause(5)
end
```

Make sure the script is in your **Current Folder** and execute the script by typing the command:

```
>> graph1
```

The **script** demos the use of:

(i) a for loop, (ii) the linspace function, (iii) element wise division (iv) the sprint function, (v) the use of format specifiers (vi) adding gridlines, (vii) the pause function. See the **Product help** for more information on each.

### Notes:

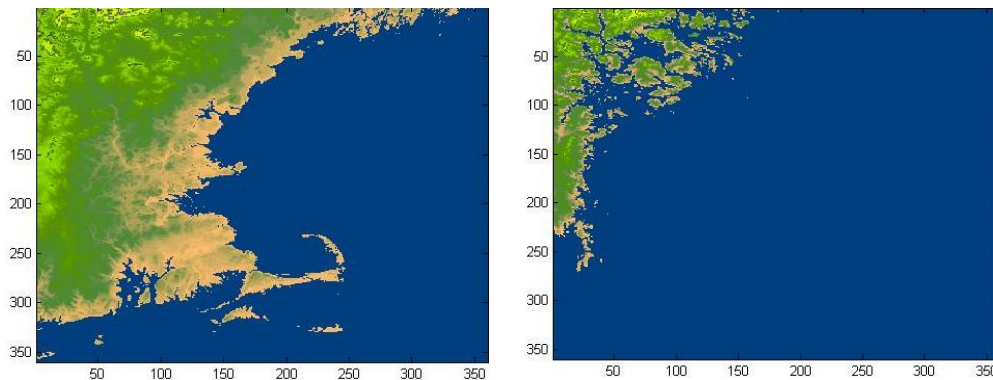
- This script will loop 5 times. On the first iteration the number of points  $n10 = 10$ , and a plot of  $y = x / (1+x^2)$  is created.
- The *linspace* function generates linearly spaced vectors. It is similar to the colon operator ":", but gives direct control over the number of points.
- $x = \text{linspace}(-2,2,n10)$  generates a row vector  $x$  of  $n10$  points linearly spaced between and including -2 and 2.
- The number of points used and the graph number will appear in the title.
- The *sprintf* function formats data into a string.
- On subsequent loops more points are used over the same x-axis range. This has the effect of the plot appearing to smooth on subsequent iterations.

### Example 2 - Receding Coastline Animation (For Loops)

Place the Matlab script file [animationWithForLoops.m](#) in your current folder and execute it by entering its name at the command prompt as follows:

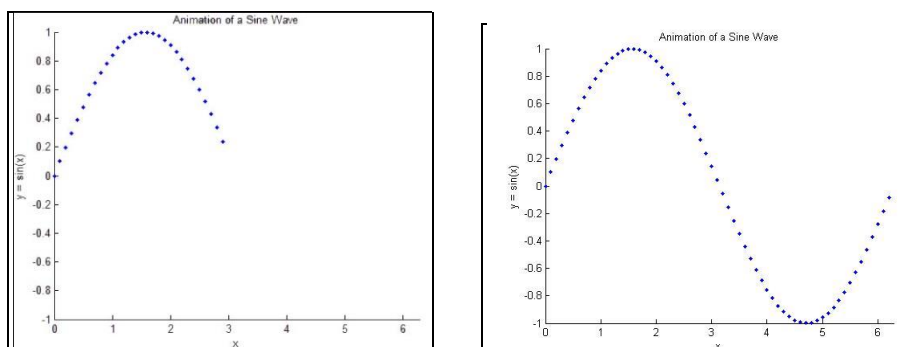
```
>> animationWithForLoops
```

This is a good example of real-time image processing with the rising sea level prediction being shown in real-time.



**Figure:** shows *Before* and *After* images of sea-level elevation.

The script also contains some ‘bonus’ commands that carry out a real-time plot animation:



**Functions:**

Functions are files that can accept **input arguments** and **return** output arguments. The name of the file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A good example is provided by **isTriangle**. The file is shown below.

```
function [isTri,typeTri,areaTri] = isTriangle(L1, L2, L3)
% ISTRIANGLE determines if the three lengths provided can form a triangle.
% isTri = isTriangle(L1,L2,L3) returns 1 if the three lengths can form a triangle
```

The first line of a function starts with the keyword **function**. It gives the function name and order of arguments. In this case, there are 3 input arguments and 3 output arguments.

The next several lines, up to the first blank or executable line, are comment lines that provide the **help** text. These lines are printed when you type:

```
>> help isTriangle
```

The first line of the help text is the H1 line, which MATLAB displays when you use the command:

```
>>lookfor isTriangle or request help on a folder.
```

The rest of the file is the executable MATLAB code defining the function. The variables **s** and **p** introduced in the body of the function are all **local** to the function; they are separate from any variables in the MATLAB workspace.

It is always good programming practice to include an example of how the function is called within the function comment.

```
>> [isTri,typeTri,areaTri] = isTriangle(3,4,5)
```

Copy the function file to your **current folder** and execute the following examples:

```
>> [isTri,typeTri,areaTri] = isTriangle(2,3,2)
>> [isTri,typeTri,areaTri] = isTriangle(3,4,5)
>> [isTri,typeTri,areaTri] = isTriangle(25,4,5)
```

**Exercise:**

Create a function to process some data such as the monthly temperature data shown previously. An input parameter is the month's title, this is parsed and used in both the plot title and in the subsequent jpg of the plot saved.

```

function [isTri,typeTri,areaTri] = isTriangle(L1, L2, L3)
% ISTRIANGLE determines if the three lengths provided can form a triangle.
% isTri = isTriangle(L1,L2,L3) returns 1 if the three lengths can form a
% triangle and 0 otherwise.
%
% [isTri,typeTri,areaTri] = isTriangle(L1,L2,L3) also returns the type of
% triangle: (Regular, Equilateral, or Isosceles), as well as the area enclosed.
%
% Example:
% [isTri,typeTri,areaTri] = isTriangle(2,3,2)
%
%     isTri =
%         1
%     typeTri =
%     Isosceles
%     areaTri =
%         1.9843
%
%     Author: Jerome Casey

% Initialize outputs
isTri = false;
typeTri = '';
areaTri = NaN;

% Create a sorted array of the inputs lengths
s = sort([L1 L2 L3]);

if (s(1)+s(2)) > s(3)
% If the sum of the two shortest sides is greater than the longest side, the
% lengths can form a triangle.

    isTri = true;
    typeTri = 'Scalene';

    if s(1) == s(3)
% If the shortest and longest lengths are equal, it implies that all three
% lengths are equal, and that the three sides form an equilateral triangle.
        typeTri = 'Equilateral';
    elseif s(1)==s(2) || s(2)==s(3)
% If middle length is equal to either the longest or shortest length,
% the lengths form an isosceles triangle.
        typeTri = 'Isosceles';
    end

    % Calculate the area of the triangle using Heron's Formula
    p = sum(s)/2;
    areaTri = sqrt(p*(p-s(1))*(p-s(2))*(p-s(3)));
end

```

**References:**

Griffiths, David F. (2005) *An Introduction to Matlab*, Department of Mathematics, The University of Dundee.

[www.maths.dundee.ac.uk/~ftp/na-reports/MatlabNotes.pdf](http://www.maths.dundee.ac.uk/~ftp/na-reports/MatlabNotes.pdf)

Love, Tim (2004) *Matlab Databook*, University of Cambridge

<http://www-h.eng.cam.ac.uk/help/tpl/programs/Matlab/matlabDatabook>

Moler, Cleve (2011) *Experiments with MATLAB*

<http://www.mathworks.co.uk/moler/exm/chapters.html>

Moler, Cleve (2004) *Numerical Computing with MATLAB*

A traditional textbook print edition, published by the Society for Industrial and Applied Mathematics, is available from the SIAM Web site.

<http://www.mathworks.co.uk/moler/chapters.html>

**Matlab Video Tutorials**

1. MATLAB Tutorials from Mathworks

[http://www.mathworks.es/academia/student\\_center/tutorials/mltutorial\\_launchpad.html#](http://www.mathworks.es/academia/student_center/tutorials/mltutorial_launchpad.html#)

2. Various Math Subjects: <http://mathworld.wolfram.com>

3. Importing Files into Matlab - Excel and other <http://www.mathworks.co.uk/help/techdoc/ref/f16-5702.html>

4. Matlab Scripts and Functions [http://www.mathworks.co.uk/help/techdoc/learn\\_matlab/f4-2525.html](http://www.mathworks.co.uk/help/techdoc/learn_matlab/f4-2525.html)