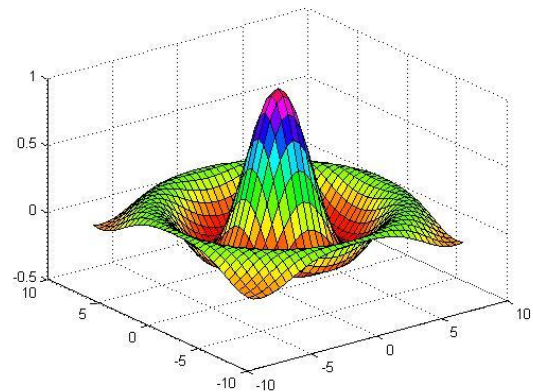
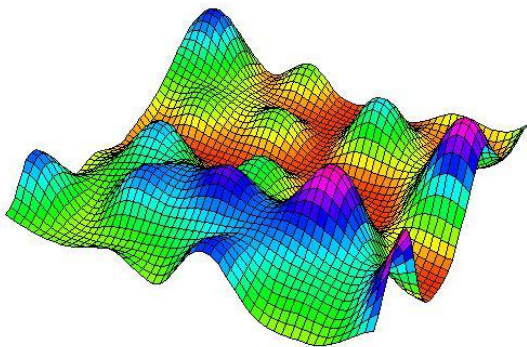


# Introductory Course to MATLAB with Financial Case Studies

---

*Panayiotis C. Andreou, PhD*



---

December 2014

# ***Table of Contents***

1	Introduction	1
1.1	Learning Matlab	1
1.2	Basic Definitions	3
1.3	Information of How to Read This Handout	5
1.4	Starting Up Matlab	6
1.5	Matlab's Windows	7
1.5.1	Command Window - Matlab as a Calculator	9
1.5.1.1	Controlling Command Window Input and Output	14
1.5.2	Editor / Debugger	17
1.5.3	Help Options	18
2	Manipulating Vectors and Matrices	20
2.1	Row Vectors	20
2.1.1	The Colon Notation	23
2.1.2	Column Vectors	24
2.1.3	Transporting Vectors	25
2.1.4	Vector Manipulations Related to Products, Division, and Powers	27
2.1.4.1	Scalar Product	27
2.1.4.2	Dot Product	28
2.1.4.3	Dot Division and Power	29
2.1.4.4	Some Useful Vector Function	31
2.2	Two Dimensional Arrays (Matrices)	33
2.2.1	Transpose of a Matrix	35
2.2.2	Elaborating on Parts of Matrices - The Colon Notation	36
2.2.3	Matrix Basic Manipulations	40
2.2.4	Matrix Manipulations Related to Products, Division, and Powers	

	_____	40
2.2.4.1	Matrix Dot Product, Division and Powers _____	40
2.2.4.2	Matrix to Vector Product - Matrix to Matrix Product _____	41
2.2.5	Special Cases of Matrices _____	43
2.2.6	Additional Useful Matrix Function _____	44
2.2.7	Example: System of Linear Equations _____	44
3	Plots and Graphs (2D and 3D) _____	47
3.1	Creating 2D Line Plots _____	47
3.1.1	Plot Edit Mode _____	48
3.1.2	Function and Style Facilities Related to Plots _____	50
3.2	Creating 3D Graphs _____	55
3.2.1	Creation of 3D Line Plots _____	55
3.2.2	Creation of 3D Mesh and Surface Graphs _____	55
3.2.2.1	Examination of a Function's Critical Points _____	60
4	Control Flow _____	63
4.1	Logical and Relational Operators _____	63
4.1.1	The any and all Function _____	66
4.2	Conditional Statements _____	66
4.2.1	The if Statement _____	67
4.2.2	The switch Statement _____	68
4.3	Loop Expressions _____	69
4.3.1	The for Loop _____	69
4.3.2	The while Loop _____	70
4.3.3	Nested Conditional and Loop Expressions _____	70
4.3.4	Additional Control Flow Statements _____	71
5	m-files: Scripts and Function _____	73
5.1	m-files: Function _____	73
5.1.1	Basic Parts of a Function _____	75
5.1.1.1	Function Definition Line _____	75

5.1.1.2	The H1 Line	76
5.1.1.3	The Help Text	77
5.1.1.4	The Body Text	77
5.2	Scripts	78
5.3	Testing and Debugging	80
5.3.1	Code Sections	81
5.3.2	Debugging	83
6	Cells and Structures	86
6.1.1	Cell indexing	87
6.1.2	Content indexing	88
6.2	Structures	90
6.2.1	Building structure arrays using assignment statements	90
6.2.2	Building structure arrays using the struct function	92
7	Entering and Saving Data Files	95
7.1	Import Data Interactively	95
7.2	Import Data using build-in functions	98
8	Data Analysis	100
8.1	Descriptive statistics	100
8.2	Statistical plots	102
8.3	Statistical Tests	104
8.4	Regression Analysis	106
8.4.1	Linear Regression - ordinary least squares (OLS)	107
8.4.2	Logistic Regression	111
9	Case Studies	114
9.1	The Black - Scholes- Merton Options Pricing Formula	114
9.1.1	Implementation of the BSM Formula	115
9.1.2	BSM Derivatives	116
9.1.3	BSM Plots and Surfaces	117
9.1.4	BSM Implied Volatility	118

9.2	Function Minimization and Plots	120
9.3	Portfolio Optimization	126
	References	131

## **List of Figures**

Figure 1: The Matlab desktop	7
Figure 2: The Matlab Workspace Browser	8
Figure 3: A basic calculation	9
Figure 4: A four-element vector	10
Figure 5: A 3-by-3 magic table	11
Figure 6: A 2-element vector	12
Figure 7: Matlab's Editor/Debugger	18
Figure 8: The figure window	49
Figure 9: An example of a Function	74
Figure 10: Function Definition Line	75
Figure 11: An example of a script	79
Figure 12: Warnings indication	81
Figure 13: Script with code sections	82
Figure 14: Increment Value dialog box	82
Figure 15: Variables Editor	84
Figure 16: A 2-by-2 cell array	86
Figure 17: Example of stricter (Figure copied from [2])	90
Figure 18: The import tool	96
Figure 19: Importing choices	97
Figure 20: Automatically generated function for importing data	97
Figure 21: Browsing through data import functions in the command window.	99
Figure 22: Script for generating descriptive statistics	101
Figure 23: Plotting Histograms output	103
Figure 24: Linear Regression example script using regress and regstats functions	108
Figure 25: Code section for performing linear regression using fitlm	110
Figure 26: Code section to create Fixed Effects Variables	110
Figure 27: Code section for performing linear regression using fixed effects with fitlm function	111
Figure 28: Code Section for performing logistic regression	112

## ***List of Tables***

Table 1: The elementary build-in function_____	13
Table 2: Various function that can be used with vectors _____	32
Table 3: Various function that can be used with vectors. _____	43
Table 4: Various function that can be used with vectors. _____	44
Table 5: Plot colour and line styles_____	50
Table 6: Various functions that help in the creation of 2D-plots. _____	50
Table 7: The if statement syntax and examples _____	67
Table 8: The switch statement syntax and example. _____	68
Table 9: The for statement syntax and example. _____	69
Table 10: The while statement syntax and example. _____	70
Table 11: Nested conditional and loop statements. _____	71
Table 12: Additional control flow statements. _____	71
Table 13: File formats that Matlab can handle_____	98
Table 14: Functions for descriptive statistics _____	100
Table 15: Functions for Statistical Plotting _____	102
Table 16: List of Statistical tests when to be used and the corresponding Matlab function_____	105
Table 17: Fields of the ouput structure of function regstats _____	109
Table 18: Fields of the structure LogicStat _____	113

# 1 Introduction

This hand-out demonstrates a comprehensive introduction to the basic utilities of a high technical programming language encapsulated under the computing environment of Matlab. It was prepared based on version **8.4 Release 2014a**, but since it does not have many differences from previous ones, it can also operate for older versions as well. Upon reading it, you will be in a position to create programming code to solve elementary (or even intermediate) financial problems.

## 1.1 Learning Matlab

Matlab (the name stands for: *Matrix Laboratory*) is a high performance programming language and a computing environment that uses vectors and matrices as one of its basic data types (MATLAB® is a registered trademark of the MathWorks, Inc.). It is a powerful tool for mathematical and technical calculations and it can also be used for creating various types of plots. It performs the basic function of a programmable calculator whereas someone can write, run/execute and save a bundle of commands. In the last few years, Matlab has become very popular because it enables numerous ways to solve various problems numerically via a user-friendly programming language. It is particularly easy to generate a programming code, execute it to get the desire solution, draw some relevant graphs and look at the interesting features of the problem under consideration. What makes Matlab so convenient to many fields (including finance) is that it integrates computation, visualization and programming in an easy to use environment. Just for historical purposes, the first version of Matlab was written in 1970s by a numerical analyst names Cleve Moler, and since then has become a successful retail product.

Matlab features a family of add-on application-specific solutions called toolboxes. A toolbox is a comprehensive collection of Matlab functions (M-files) that extend the Matlab environment to solve particular classes of problems. For example, the Financial Toolbox includes ready to use functions that provide a complete integrated computing environment for



financial analysis and engineering. The toolbox has everything you need to perform mathematical and statistical analysis of financial data and display the results with presentation-quality graphics. With MATLAB and the Financial Toolbox, you can: compute and analyse prices, yields, and sensitivities for derivatives and other securities, and for portfolios of securities; analyse or manage portfolios; perform asset pricing exercises; design and evaluate hedging strategies and many more.

This hand-out is about a brief introductory course in the most important parts of Matlab. After finishing this, you will be able to:

- i.** Utilize the basic mathematical operations;
- ii.** Get know the general purpose commands of Matlab;
- iii.** Become familiar with some of the elementary function, matrix and numerical linear algebra function, as well as some graphic and plot commands;
- iv.** Manipulate matrices (i.e. create and edit vectors and matrices, build a larger matrix from a smaller one, etc);
- v.** Learn how to use the relational operators (<, >, <=, >=, ==, ~=) and logical operators (&& AND, || OR, ~ NOT);
- vi.** Recognize build-in variables, define new ones and performing computations with them;
- vii.** Learn how to use *if* and *switch* statements;
- viii.** Learn how to correctly utilize loop commands, such as the *for* loop, and the *while* loop;
- ix.** Write and edit your own m-files and function for solving a specific problem;
- x.** ...Numerous other utilities and uses depending on your will to learn and utilize this technical language.

It is better to use this hand-out in direct use with the Matlab. It will be more beneficial if while reading these notes you sit in front of a PC and type the various commands and execute the given examples.

Although the Matlab package includes extensive help facilities that can be viewed via a very user-friendly Help Browser, if it is needed, you can also

find additional online (through Internet) help at the following electronic address:

<http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml>

Moreover, various download, trials and the Matlab Central File Exchange that contains hundreds of files contributed by users and developers of Matlab and related products can be found at:

<http://www.mathworks.com/matlabcentral/fileexchange/>

Additionally, any function that have been created for examples and all those that are needed to work with the case studies are located in the folder named as: *Matlab Examples* on my personal website here:

<http://www.pandreou.com/miscellaneous.php>

## 1.2 Basic Definitions

Before moving to the introduction of issues specific to Matlab, some very basic definitions on computers and programming should be set forth. These include:

### General [1]

- *bit* (short for binary digit) is the smallest unit of information on a computer. A single bit can hold only one of two values: 0 or 1. More meaningful information is obtained by combining consecutive bits into larger units, such as byte.
- *byte* consists a unit of 8 bits and is capable to hold a single character. Large amounts of memory are indicated in terms of kilobytes (1024 bytes), megabytes (1024 kilobytes), and gigabytes (1024 megabytes).
- *data* is information represented with symbols such as numbers, words, images, etc.
- *command* is a collection of programming words that instruct the computer to do a specific task.
- *algorithm* is a set of commands that aim to solve a predetermined

problem.

- *program* is the implementation of the algorithm suitable for execution by a computer.
- *variable* is a container that can hold a value. For example, in the expression:  $z+a$ , both  $z$  and  $a$  are variables. Variables can hold both numerical data (e.g. 10, 98.5) and characters or strings (e.g. 'd' or 'dubs').
- *constant* is a value that never changes. It can be a numeric, a character or a string.
- *bug* is an error in a program, causing the program to stop running, not to run at all or to provide wrong results. Some bugs can be very subtle and hard to find. The process of finding and removing the bugs is called *debugging*.

### **Matlab Specific [2]**

- *workspace* is the memory allocated to Matlab and is used to temporarily store variables.
- *m-file* is a file that contains Matlab's language code. *m-files* can be *function* that accept arguments and produce output, or they can be *scripts* that execute a series of Matlab statements. For Matlab to recognize a file as an *m-file*, its file name extension must be ".m".
- *function* are *m-files* that can accept input arguments and return output arguments. The name of the *m-file* and the calling syntax name 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*. Functions are useful for extending the existing Matlab language for personal applications (e.g. create a function that returns the expected return and standard deviation related with a set of companies).
- *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*. *Scripts* are useful for automating a series of steps that are needed to be performed many times (e.g. to create a *script* that executes a series of

function related to portfolio optimization).

- Every *variable* has a *name* and a *data type*. With Matlab, accepted *variables names* do not start with symbols (like ~, +, -) or numbers, use lower and upper case letters do not exceed 63 characters and do not resemble *reserved words* and *build-in functions*. Acceptable definitions include: *Time*, *x*, *y*, *XYZ*, *Ray\_value*, *U3e23* etc. Non-acceptable definitions are the followings: *+Time*, *3587num*, *\_XYZ*, *rayX-values*, *for*, *end*, *while*, *ca.se*, *day* etc. The *data type* is a classification of particular type information. Among the most usable types are: *integer* a whole number, a number without any fraction (e.g. 12, 10, 89); *floating point* a number with a fractional part (e.g. 25.7, 78, 1, 0.000005, 5e-5 which is equal to  $5 \cdot 10^{-5}$ ); and *character* readable text character (e.g. 'k'). With Matlab, it is not need to type or declare variables used in the *m-files*. Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.
- Every *build-in* or user made *function* has a *calling syntax* that is unique for each function. In Matlab, to call a *function* you type the function's name and you enclose the input arguments in brackets (if more than one input argument exist, then commas should be used to separate the expressions).
- Matlab is handled through the use of various windows, that each is related with a certain utility. For example, the *Command Window* is used to enter variables and run *function* and *m-files*, the *Command History Window* makes a diary with the commands that you have recently entered in the *command window*, the *Workspace Browser* allows you to view the variables that are stored in the *workspace*, etc. Any reference to such windows that is made in the main body of these notes will be explained in detail if it is needed.

### 1.3 Information of How to Read This Hand-out

In the main body of this hand-out, words that refer to a keyboard instruction will be written in brackets and in boldface letter. For example, **[Enter]** will represent one key-press of the Enter button of the keyboard. Words that

represent Matlab's *reserved words* such as *for*, *end*, *who*, etc., words that imply a Matlab term such as *m-file*, *script*, *function*, etc., or key-words related with the interface menu and toolbars will appear in *italics*. Words in the main body that refer to Matlab's vector and matrix names, Matlab's *build in functions* or user's new *functions* will be presented with a shadow style. For example, if in the main body a reference is made to a vector saved in the *workspace* with the name "Hours" it will look like: **Hours**. The reference to Matlab's function that creates 2D figures will be as: **plot**. Moreover, note that although each *function* has a specific *calling syntax*, usually only the name of the function will be displayed; if not presented in the body text of this hand-out, the user should be responsible to find the exact *calling syntax* of the *function* via the Matlab's help facilities.


Other Matlab output such as warnings, tips or Matlab commands will be enclosed in double quote marks " ". The definition of Matlab related words and expression in windows (as you will see later) consist exceptions from these rules. Lastly, references to books and other reading material are numbered in square brackets (e.g. [2] is the Matlab online help facilities; see the section with the references). Additionally, if a figure or a table heading that has a superscript number enclosed in square brackets will indicate the source from where it was copied (for instance, Figure<sup>X</sup>: Figure heading, indicates that the current figure was copied from Matlab online help facilities).

## 1.4 Starting Up Matlab

The following instructions usually help for starting up the Matlab in a computers lab:

**Step #1:** Use the [Ctrl] + [Alt] + [Del] combination to bring up the logon screen (at this point you should enter the user name and your password and after to press [Enter])

**Step #2:** After few seconds, you view the PC's Desktop screen with all available icons. Find the Matlab's shortcut icon (labelled as "Matlab" and

looks like  ) and double click on it. After few moments, the Matlab

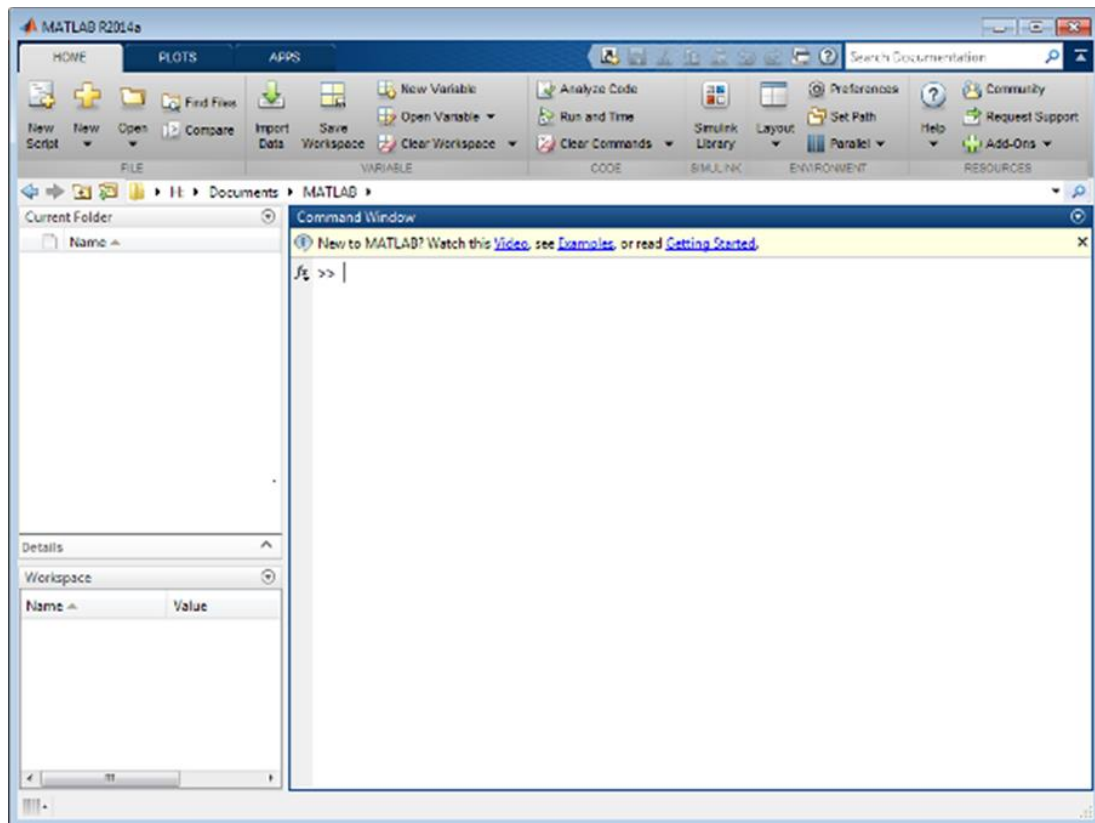
starts up and the following sentence appear in one of the screens:

“To get started, select "MATLAB Help” from the Help menu”

**Step #3:** The Matlab is now ready to be used (if you want to quit Matlab, from the window named as *Command Window* either type quit or exit from the toolbar choose: *File > Exit Matlab*).

## 1.5 Matlab's Windows

When you start up Matlab, you will view the following interface (Figure 1 - a difference interface may appear if someone before you has changed the active windows from the *View* menu):



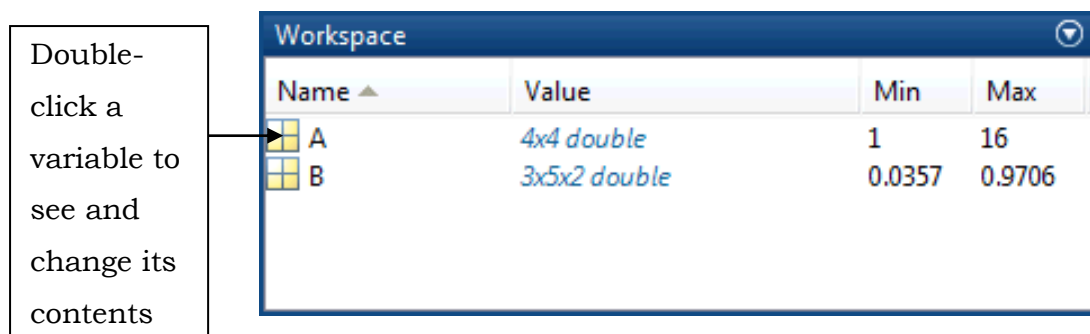
**Figure 1: The Matlab desktop**

The above is termed as the Matlab Desktop (graphical user interfaces) and contains tools for managing files, variables, and applications associated with Matlab. Think of the desktop as your instrument panel for Matlab. The tabs in the Matlab desktop provide easy access to frequently used operations.


Hold the cursor over a button and a tooltip appears describing the item. Note that some of the tools also have toolbars within their windows [2].

You can change the way your desktop looks by opening, closing, moving, and resizing the windows in it. *You can also move tools outside the desktop or move them back into the desktop (docking).* All the desktop tools provide common features such as context menus and keyboard shortcuts. You can specify certain characteristics for the desktop tools by selecting *Preferences* from the *Environment* section on the *Home* tab. For example, you can specify the font characteristics for *command window text*. For more information, click the *Help* button in the *Preferences* dialog box [2].

Additionally, two more windows that can be seen in Figure 1 can help the user during the programming time. The first one and most important is the *workspace browser* (Figure 2). The Matlab *workspace* consists of the set of variables (named arrays) built up during a Matlab session and stored in memory. You add variables to the *workspace* by using *functions*, running *m-files*, and loading saved *workspaces*



**Figure 2: The Matlab Workspace Browser**

The *workspace* is not maintained after you end the Matlab session. To save the *workspace* to a file that can be read during a later Matlab session, click on  and then *Save*.

The third window that appears in Matlab's desktop is the *Current Folder*. The *Current Folder browser* shows the files and folders in the current directory. The path to the current directory is listed near the top of the MATLAB desktop. Matlab file operations use the *current directory* and the *search path* as reference points. Any file you want to run must either be in the *current directory* or on the *search path* (the *search path* is a default list of paths that

include all folders with Matlab *build-in functions* and toolboxes; to access this window from the *Home tab* click on *Set Path* in the *Environment section*).

Another useful window is the *command history*. Statements you enter in the *Command Window* are logged in the *Command History*. In the *Command History*, you can view previously run statements, and copy and execute selected statements.

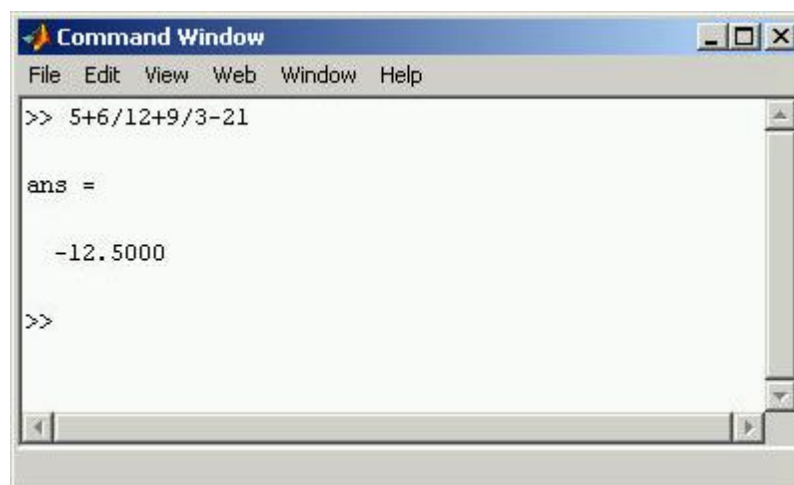
To activate or close these and some other windows, on the *Home tab*, in the *Environment section*, click *Layout*. In the following few subsections, the most basic from this screen are documented.

### 1.5.1 Command Window - Matlab as a Calculator

It is the main window in which the user communicates with the software. In the *command window*, the user can view the prompt symbol “>>” which indicates that Matlab is ready to accept various commands by the user. Via this window, the user can employ the basic arithmetic operators like: “+” (addition), “-” (subtraction), “\*” (multiplication), “/” (division), “^” (powers) and the “( )” (brackets), as well as many other build in elementary and other *Function* and commands that will be referred to later.

As a first example, enter the following command that performs a basic calculation (Figure 3):

“5+6/12+9/3-21”

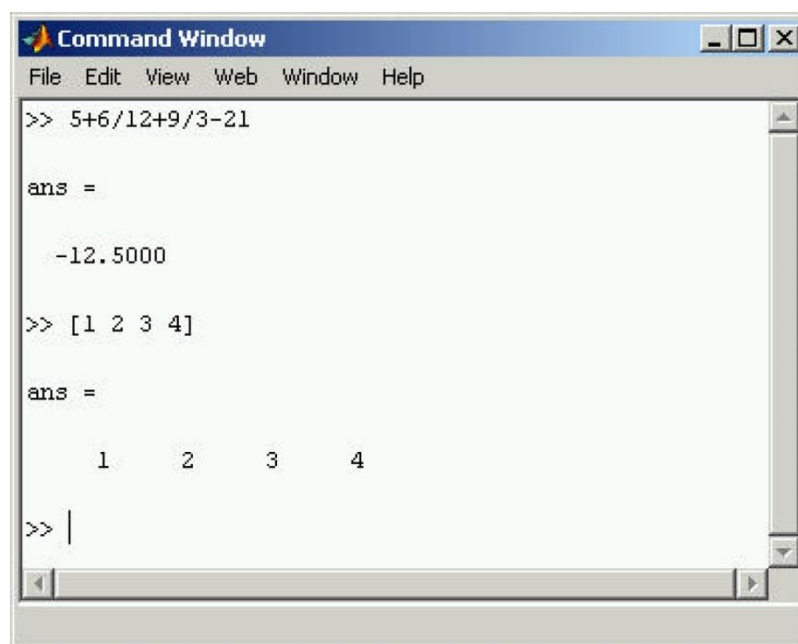


**Figure 3: A basic calculation**



The Matlab displays the results into a variable named as `ans`. This is a default variable name that is used by the *command window* to display the most recent results instructed by the user that has not defined a specific name. Continue by entering the following command that creates a four-element vector (Figure 4):

```
"[1 2 3 4]"
```



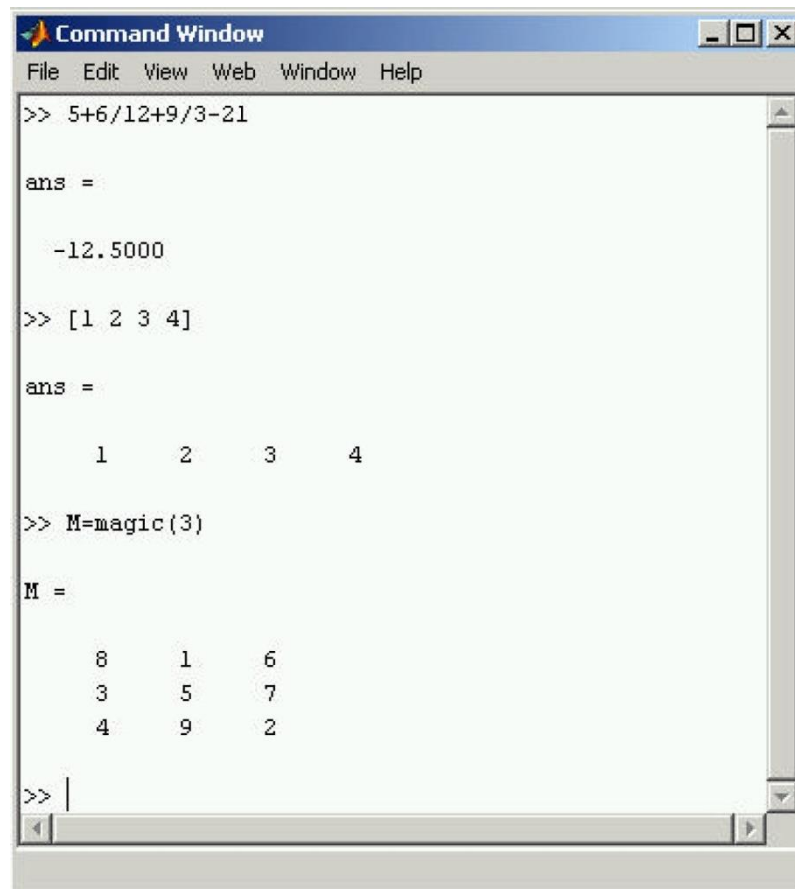
**Figure 4: A four-element vector**

The square brackets “[ ]” indicate the definition of the vector. Also, the space between the vector numbers separates the vector’s elements. The comma “,” is another way of separating the elements. Additionally, note that the default variable `ans` has lost its previous value in order to store the results of the most recent operation.

As another example, enter the following command that creates a 3-by-3 magic square saved in the matrix variable `M`:

```
"M=magic(3)"
```

and after pressing [**Enter**] you get the Matlab's result (Figure 5):



```
Command Window
File Edit View Web Window Help
>> 5+6/12+9/3-21
ans =
-12.5000
>> [1 2 3 4]
ans =
1 2 3 4
>> M=magic(3)
M =
8 1 6
3 5 7
4 9 2
>> |
```

**Figure 5: A 3-by-3 magic table**

The magic square is created using the element function `magic` that is already *built-in* Matlab.

Type also the following:

`"x=[(2^2+1)^2-15/4*6.1, 1.23e-2]"` and after pressing [**Enter**] you get

the Matlab's result (Figure 6)

```

Command Window
File Edit View Web Window Help
>> [1 2 3 4]
ans =
     1     2     3     4
>> M=magic(3)
M =
     8     1     6
     3     5     7
     4     9     2
>> x=[(2^2+1)^2-15/4*6.1, 1.23e-2]
x =
     2.1250     0.0123
>>

```

**Figure 6: A 2-element vector**

Observe that the very first command that you have entered in the *command window* has vanished with the additional of the last one (of course it depends on the size of the window, in here the *command window* is minimized so earlier commands vanish too quickly). You can use the *scroll bar* on the right to navigate the *command window* and go up to see the earlier commands (or view them via the *command history window*).

Matlab calculates the quantities as follows:

- First the quantities in brackets.
- Power calculation follow: (e.g.  $5 + 2^2 = 5 + 4 = 9$ )
- “\*” and “/” follow by working from left to right: (e.g.  $2*8/4 = 16/4 = 4$ )
- “+” and “-” follow last, from left to right: (e.g.  $6-7+2 = -1+2 = 1$ )

Note that “e” notation is used for very large or very small numbers. By definition:  $1e1 = 1 \times 10^1$  and  $1e-1 = 1 \times 10^{-1}$ .

Matlab can handle three different kinds of numbers: *integers*, *real numbers* and *complex numbers* (with imaginary parts). Moreover, it can handle nonnumber number expressions like: NaN (Not-a-Number) produced from mathematically undefined mathematically undefined operations like: 0/0,  $\infty * \infty$  and inf produced by operations operations like 1/0. Matlab as a calculator includes a variety of build-in mathematical mathematical function like: *trigonometric*, *exponential*, *complex*, *rounding and remainder*, etc.

Table 1, depicts these elementary mathematical *build-in functions* (to learn how to use these commands, review the Matlab help facilities located in section 1.5.3).

**Table 1: The elementary build-in function**

<b>Trigonometric</b>		<b>Exponential</b>	
sin	Sine.	exp	Exponential
sinh	Hyperbolic sine.	Log	Natural logarithm.
asin	Inverse sine.	log10	Common (base 10) logarithm.
asinh	Inverse hyperbolic sine.	log2	Base 2 logarithm and dissect floating
cos	cosine.	pow2	Base 2 power and scale floating point
cosh	Hyperbolic	realpow	Power that will error out on complex
acos	Inverse cosine.	reallog	Natural logarithm of real number.
acosh	Inverse hyperbolic cosine	realsqrt	Square root of number greater than or
tan	Tangent	sqrt	Square root.
tanh	Hyperbolic tangent.	nextpow2	Next higher power of 2.
atan	Inverse tangent.		
atan2	Four quadrant inverse tangent.	<b>Complex</b>	
atanh	Inverse hyperbolic tangent.	abs	Absolute value.
sec	Secant	angle	Phase angle
sech	Hyperbolic secant	complex	Construct complex data from real and imaginary parts
asec	Inverse secant	conj	Complex conjugate.
asech	Inverse hyperbolic secant	imag	Complex imaginary part.
csc	Cosecant	real	Complex real part.
csch	Hyperbolic cosecant	unwrap	Unwrap hase angle.
acsc	Inverse cosecant	isreal	True for real array
acsch	Inverse hyperbolic cosecant	cplxpair	Sort numbers into complex conjugate pairs
cot	Cotangent.	<b>Rounding and Remainder</b>	
coth	Hyperbolic cotangent		
acot	Inverse cotangent	floor	Round towards minus infinity.

acoth	Inverse hyperbolic cotangent	ceil	Round towards plus infinity.
coth	Hyperbolic cotangent	round	Round towards nearest integer.
acoth	Inverse hyperbolic cotangent	mod	Modulus (signed remainder after remainder after division.
		rem	remainder after division.
		sign	Signum

Note the *build-in function* exhibited in Table 1 have their own *calling syntax*. For example, if you type `sin` in the command window, Matlab returns the following error message:

```

“Error using sin
    Not enough input arguments.”

```

This happened because the `sin` function requires an *input expression* like a number enclosed in brackets. If you enter:

```

“sin(5)”

```

then you get an answer:

```

“ans =
    -0.9589”

```

In the rest of this hand-out, only the name of the build-in function will be given. Beside some exceptions where the correct *calling syntax* of a function is fully tabulated, the user is responsible in knowing the necessary input arguments to the function. Additionally the use of the *command window* will be apparent as you read this manuscript since the learning of a computer programming language is pure a “**learning by doing**” process.

### 1.5.1.1 Controlling Command Window Input and Output

This section presents ways by which the user can control the *command window* input and output.

#### **List of Useful Commands:**

`who` Lists current variables located in the *workspace*.

**whos** A long form of who. It lists all the variables in the current *workspace*, together with information about their size, bytes, class, etc.

**clear** Clear variables and *Function* from memory.

**home** Moves the cursor to the upper left corner of the *command window* and clears the visible portion of the window. Use the scroll bar to see what was on the screen previously.

**clc** Clears the *command window* and homes the cursor.

**quit** Terminates Matlab.

**The format function:**

The format function controls the numeric format of the values displayed by Matlab. The *function* affects only how numbers are displayed, not how Matlab computes or save them. Here are the different *formats*, together with the resulting output produced from a vector x with components of different magnitudes [2]. In the command window type:

“format short”

and afterwards:

“x”

for retrieving the x vector that you have created before to get:

“x =

2.1250 0.0123”

Note that with this format only 4-decimals place are being used (in total 5 digits). Try also the following format commands to see what you get:

**format short e** Floating-point format with 5 digits.

**format short g** Best of fixed or floating-point format with 5 digits.

**format long e** Floating-point format with 15 digits.

**format long g** Best of fixed or floating-point format with 15 digits.

**format bank** Fixed format for dollars and cents.

## Suppressing Output

If you simply type a statement and press [**Enter**], Matlab automatically displays the results on screen. However, if you end the line with a semicolon “;” Matlab performs the computation but does not display any output [2]. This is particularly useful when you generate large matrices. For example,

```
“A = magic(100);”
```

although Matlab creates a 100-by-100 matrix saved in the *workspace*, the result is not displayed on the *command window*. Moreover, when a command ends with “;”, is allowed to enter additional commands before pressing the [**Enter**] key. For instance,

```
“A = magic(100); B=A;”
```

will perform two commands the one after the other. In here, after creating the 100-by-100 magic square that is stored in the A (matrix) variable, matrix B is created to hold the same values as with A. Additionally, note that if two or more statements are separated with commas, Matlab will display the results in the screen in the order that these statements were entered.

## Entering Long Statements

If a statement does not fit on one line, use an ellipsis (three periods), “...”, followed by [**Enter**] to indicate that the statement continues on the next line. For example,

```
“s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...  
- 1/8 + 1/9 - 1/10 + 1/11 - 1/12”
```

to get:

```
0.65321”
```

Blank spaces around the “=”, “+”, and “-“operators are optional, but they improve readability.

## Command Line Editing

Various arrow and control keys on your keyboard allow you to recall, edit, and re-use statements you have typed earlier. For example, suppose you mistakenly enter:

```
“rho = (1 + sqrt(5))/2”
```

You have misspelled square root: `sqrt`. Matlab responds with:

```
“Undefined function 'sqrt' for input arguments of type 'double'.”
```

Instead of retyping the entire line, simply press the up- arrow key  $\uparrow$ . The statement you typed is redisplayed. You can recall previous commands by pressing the up- and down-arrow keys,  $\uparrow$  and  $\downarrow$ . Press the arrow keys either at an empty command line or after you type the first few characters of a command. You can also copy previously executed statements from the command *history window* [2]

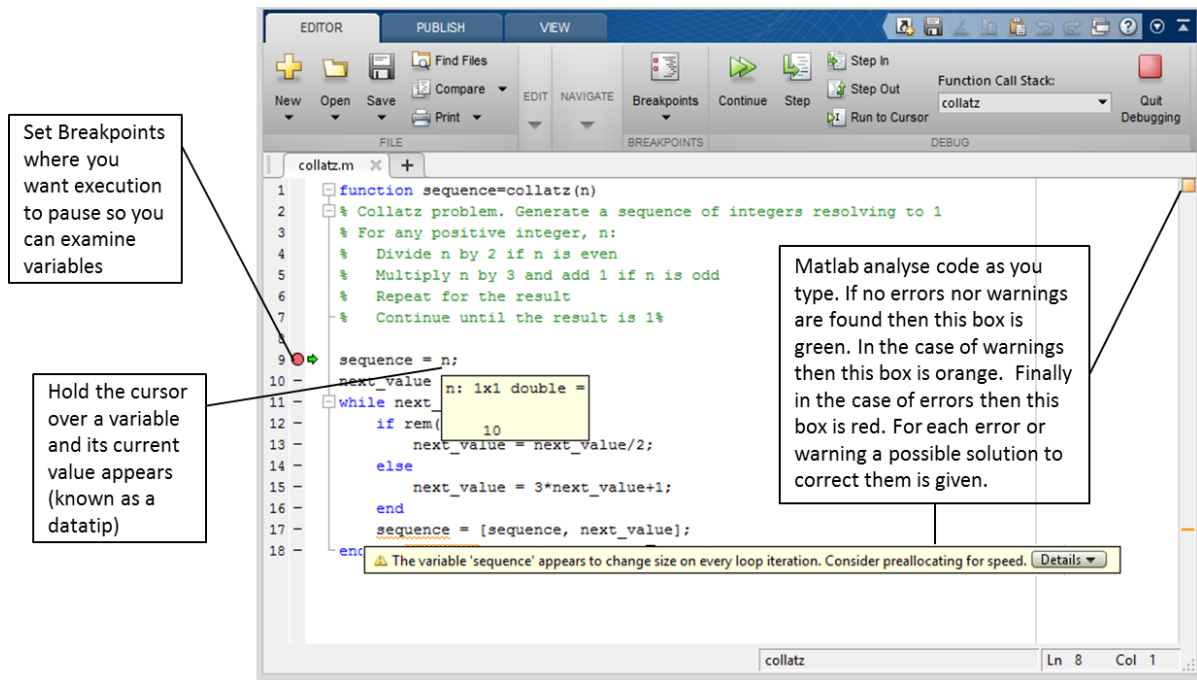
## Interrupting a Command, a Script or a Function

It is often that a user writes a command (or *script* or a *function*) that enters to an endless loop, so it keeps running. Pressing **[Ctrl] + [C]** once or few times, it will terminate the execution. Sometimes where the user runs a script that calls other function, the interruption with **[Ctrl] + [C]** might take few seconds to few minutes to take place.

## 1.5.2 Editor / Debugger

Use the Editor/Debugger to create and debug m-files, which are programs you write to run Matlab functions [2]. The Editor/Debugger provides a graphical user interface for basic text editing, as well as for m-file debugging (Figure 7).





**Figure 7: Matlab's Editor/Debugger**

It's like a text editor (e.g. MS Word) suitable for writing executable Matlab code. Notice that reserved words (e.g. `while`, `if`, `else`, `end`, *functions* etc) appear in blue styling whilst accompanied comments in the programming code (that can be placed after the comments symbol: “%”) appear in green letters [2]. Exploit the various menus to become familiar with this window.

### 1.5.3 Help Options


All MATLAB functions have supporting documentation that includes examples and describes the function inputs, outputs, and calling syntax. There are several ways to access this information from the command line:

- Open the function documentation in a separate window using the `doc` command.  
“`doc mean`”
- Display function hints (the syntax portion of the function documentation) in the Command Window by pausing after you type the open parentheses for the function input arguments.  
“`mean( “`
- View an abbreviated text version of the function documentation in the

Command Window using the help command.

“help mean”

The problem with the above ways is that the user must be familiar with the topic under consideration and the word following the help command must be exact and spelled correctly. For example, the Matlab function that returns the inverse of a matrix is named as: `inv` (type “`help inv`” to see what you will get). If the user types “inverse” or “inverce” instead, then Matlab help will return that such functions cannot be found.

A more flexible way for pursuing help from Matlab, is to access the complete product documentation by clicking the help icon . Through this screen, the user can navigate around a variety of topics by double clicking on them (this browser displays html help pages and can be operate like the Internet Explorer) or search for a function using a description of the required function. Returning, to the previous example by just typing “inverse of a matrix” all functions that can perform the operation inverse of a matrix will be displayed.

To become familiar with the Matlab’s accompanied toolboxes, navigate in the contexts, open up the dropping down choices and get know various topics.

## 2 Manipulating Vectors and Matrices

A matrix or an array is the basic element on which Matlab can operate. A *1-by-1* matrix forms a *scalar* or a single number, whereas a matrix with only one row or column forms a *row* or *column* vector respectively. This section exhibits the mathematical manipulation of vectors (arrays) and of two-dimensional matrices.

### 2.1 Row Vectors

A vector is a list of numbers separated by either space or commas. Each different number/entry located in the vector is termed as either element or component. The number of the vector elements/components determines the length of the vector. In Matlab, square brackets “[ ]” are used to both define a vector and a matrix. For instance, the following command returns a row vector with 5 elements:

Matlab's command:

```
>> y=[ 5 exp(2) sign(-5) sqrt(9) pi]
```

Matlab's response:

```
y =  
      5      7.3891      -1      3      3.1416
```

Note that the definition of the row vector, the user is free to use any *build-in function* as long as this is used properly. In the above definition, `exp` is the exponential, `sign` returns the sign, `sqrt` is the square root and `pi` represents  $\pi$ . General speaking and except some special cases, when a *function* is applied to a *1-by-1* scalar, the result is a scalar, when applied to a row or column vector is a row or column vector and when applied to a matrix the output is again a matrix. This happens because Matlab applied the *build-in function* element-wise. The length of the above vector is obtained via the

following command:

Matlab's command:

```
>> length(y)
```

Matlab's response:

```
ans =
```

```
5
```

Note that `length` is a *build-in function* that given a vector, it returns its length. Since the result of this function is not stored in a user-defined variable, Matlab saves the result in the `ans` variable. If it is need to save the vector's length to a variable named `Ylength` the command would be:

Matlab's command:

```
>> Ylength=length(y)
```

Matlab's response:

```
Ylength =
```

```
5
```

With Matlab, vectors can be easily multiplied by a scalar and added or subtracted with other vectors of similar length. Moreover, a scalar can be added or subtracted to or from a vector and smaller vectors can be used to construct larger ones. All these operations are performed element-wise. Note that each vector represents a variable. Some examples follow:

Matlab's command:

```
>> a=[-1 2 -3]; b=[2 1 2];
```

```
>> c=5*a
```

Matlab's response:

```
c =
```

```
-5 10 -15
```

Comments:

An element-by-element multiplication of vector `a` by 5.

Matlab's command:

```
>> c1=5+b
```

Matlab's response:

```
c1 =  
    7    6    7
```

Comments:

An element-by-element addition of vector **b** with 5.

Matlab's command:

```
>> d=a+b
```

Matlab's response:

```
d =  
    1    3   -1
```

Comments:

An element-by-element addition of two equal length vectors.

Matlab's command:

```
>> e=[2*a, c, d]
```

Matlab's response:

```
e =  
   -2    4   -6   -5   10  -15    1    3   -1
```

Comments:

A larger vector is created after certain manipulations.

To refer to specific elements of the vector, the vector's name is followed by the element's rank in brackets. For instance we can change the values of the **e** vector with the following command:

Matlab's command:

```
>> e(2)=-99; e(4)=-99; e(6)=-99;  
>> e
```

Matlab's response:

```
e =  
   -2  -99   -6  -99   10  -99    1    3   -1
```

### 2.1.1 The Colon Notation

The colon notation “:” can be used to pick out selected rows, columns and elements of vectors, matrices, and arrays. It is a shortcut that is used to create row vectors. Moreover, the colon notation is used to view or extract parts of vectors (afterwards, with matrices, the colon can be used to view a certain part of a matrix). For instance, the following commands produce three different row vectors:

Matlab’s command:

```
>> v1 = 1:8, v2=-4:2:2, v3=[0.1:0.2:0.6]
```

Matlab’s response:

```
v1 =  
    1     2     3     4     5     6     7     8  
  
v2 =  
   -4    -2     0     2  
  
v3 =  
    0.1  0.3  0.5
```

The careful viewer should have notice that the vector creation via the use of the colon notation has the form: *starting\_value:step:finishing\_value*. The *starting\_value* consists the first value/element of the vector, the *finishing\_value* is the last value/element of the vector and all other elements differ by a value equal to *step*. Also, when the interval between *finishing\_value* and *starting\_value* is not divisible by the *step*, the last value of the vector is not the *finishing\_value* (i.e. in **v3** the last element is 0.5 and not 0.6). The colon notation is also used to view or extract parts of a vector. See the examples to get an idea.

Matlab’s command:

```
>> v4=v1(2:4)
```

Matlab’s response:

```
V4 =  
    2     3     4
```

Comments:

Extracting the 2<sup>th</sup>, 3<sup>rd</sup> and 4<sup>th</sup> elements of v1.

Matlab's command:

```
>> v1(2:3:8)
```

Matlab's response:

```
ans =  
    2    5    8
```

Comments:

Extracting the 2<sup>th</sup>, 5<sup>th</sup> and 8<sup>th</sup> elements of v1.

## 2.1.2 Column Vectors

Column vectors are created via the use of semi-colon “;” instead of commas and spaces. Operations with column vectors are similar as with row vectors. The following examples depict the manipulation of column vectors.

Matlab's command:

```
>> cv=[1;4;7;9]
```

Matlab's response:

```
cv =  
    1  
    4  
    7  
    9
```

Comments:

Creating a four-element column vector.

Matlab's command:

```
>> length(cv)
```

Matlab's response:

```
ans =  
    4
```

Comments:

Taking the vector's length.

Matlab's command:

```
>> CV=[cv(1:2)+2; cv(2:3)*5]
```

Matlab's response:

```
CV =  
    3  
    6  
   20  
   35
```

Comments:

Creating **CV** that is a four-element column vector. Its first two elements are the two first elements of **cv** increased by 2 whereas the last two elements are the 2<sup>nd</sup> and 3<sup>rd</sup> elements of **cv** multiplied by 5.

### 2.1.3 Transpose of Vectors

To perform operation with column and row vectors of similar length, it is first needed to transpose the vectors in order to make them either all row vectors or column ones. The apostrophe symbol “ ' ” is used to convert a column vector to a row one and vice versa. Remember that the length of the vectors must match; otherwise, Matlab will return an error. Experiment with the following examples to learn the use of transpose command.

Matlab's command:

```
>> z1 = [2 -1 3], z2 = z1', z3 = z2'''
```

Matlab's response:

```
z1 =  
    2   -1    3  
  
z2 =  
    2  
   -1  
    3  
  
z3 =  
    2   -1    3
```

Comments:

**z1** is a three element row vector. **z2** is the transpose of **z1**. **z3** is similar with **z1** since after transposing **z2** three times is similar as transposing once the **z2**.



Matlab's command:

```
>> z1'+z2, z1+z2
```

Matlab's response:

```
ans = 4 -2 6
```

```
??? Error using ==> +  
Matrix dimensions must agree.
```

Comments:

$z1'+z2$  operation involves the addition of two column vectors.  $z1+z2$  is an illegal operation since a row vector cannot be added with a column vector.

Note that that Matlab can store empty vectors. This is done as follows:

Matlab's command:

```
>> Z=[], Z=1:3, Z=[], length(Z)
```

Matlab's response:

```
Z =
```

```
 []
```

```
Z =
```

```
 1  2  3
```

```
Z =
```

```
 []
```

```
ans =
```

```
 0
```

Comments:

Definition of an empty vector.

Empty vector operation is usually used in cases where the user wants to turn a non-empty vector to an empty one (to reset all the elements of a vector).

## 2.1.4 Vector Manipulations Related to Products, Division, and Powers

Matlab can perform scalar products (or inner products) and dot products, as well as dot division and power operations. The only restriction is that the length of the vectors must be the same. The priorities concerning vector manipulations is the same as in the case that we use Matlab as a calculator (power operations first followed by “\*” and “/” followed by “+” and “-“)

### 2.1.4.1 Scalar Product

The scalar product or otherwise termed as inner product, concerns the multiplication of two equal length vectors. The symbol “\*” is used to carry out this operation. Given a row vector  $W$  and a column vector  $U$  of length  $t$

$$\tilde{w} = [w_1, w_2, \dots, w_t], \tilde{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_t \end{bmatrix}$$

The inner product is defined as the linear combination:

$$\tilde{w}\tilde{u} = \sum_{i=1}^t w_i u_i$$

The following depicts some examples with inner products.

Matlab's command:

```
>> w=[1 0 2 -1]; u=[2; 4; -2; 0.5];
```

```
>> prod1=w*u, prod2=(2+w)*(u/2), prod3=w*w', prod4=w*u*w*u
```

Matlab's response:

```
prod1 =  
-2.5
```

```
prod2 =  
3.25
```

```
prod3 =  
6
```

```
prod4 =  
    6.25
```

Comments:

Producing the inner product of various operations. For example `prod1` is computed as: `prod1 = (1*2)+(0*4)+(2*(-2))+((-1)*0.5)=-2.5`.

### 2.1.4.2 Dot Product

The dot product involves the multiplication of two similar vectors (to be either column or row vectors with same length) element-by-element. With the dot product, a new vector is created. The dot product between the row vectors  $\tilde{w}$  and  $\tilde{u}^T$  (where the superscript  $T$  represents the transpose symbol) is another row vector with the following form:

$$\tilde{w}\tilde{u}^T = [w_1u_1, w_2u_2, \dots, w_tu_t]$$

The dot product in Matlab is performed via the “.” symbol. By the use of dot product we can get the inner product. This is explained in the following examples:

Matlab's command:

```
>> dprod1=w.*u', dprod2=u'.*u'.* w
```

Matlab's response:

```
dprod1 =  
    2    0   -4  -0.5  
  
dprod2 =  
    4    0    8  -0.25
```

Comments:

Producing the dot product of various operations. For example `dprod1` is computed as: `dprod1 = [1*2, 0*4, 2*(-2), (-1)*0.5]`.

Matlab's command:

```
>> inner1=sum(w.*u'), inner2=sum((2+w).*(u'/2))
```

Matlab's response:

```
inner1 =  
   -2.5
```

```
inner2 =  
    3.25
```

Comments:

Producing the summation of dot product of various operations. When the `sum` function is used with a dot product, the result is the inner product of that operation.

In vector manipulations, dot product is performed after dot power (see the following section).

### **2.1.4.3 Dot Division and Power**

The dot division does not exist mathematically, but for programming purposes, Matlab includes such operation. The dot division can be used to divide a vector with another vector of similar characteristics and also to divide a vector with a scalar. Through the following examples the use of dot division is outlined. Pay attention on the cases where a dot division produces pathological results.

Matlab's command:

```
>> d_div1=-3:1.5:3, d_div2=1:5  
>> d_div2./d_div2, d_div1./d_div2, d_div2./5
```

Matlab's response:

```
d_div1 =  
   -3.0000   -1.5000         0    1.5000    3.0000  
  
d_div2 =  
     1     2     3     4     5  
  
ans =  
     1     1     1     1     1  
  
ans =  
   -3.0000   -0.7500         0    0.3750    0.6000  
  
ans =  
     0.2000    0.4000    0.6000    0.8000    1.0000
```

Comments:

Examples of dot division.

Matlab's command:

```
>> d_div2./d_div1
>> d_div2(3)=0; d_div1./d_div2
>> 1/d_div1
```

Matlab's response:

Warning: Divide by zero.  
(Type "warning off MATLAB:divideByZero" to suppress this warning.)

```
ans =
    -0.3333 -1.3333   Inf  2.6667 1.6667
```

Warning: Divide by zero.  
(Type "warning off MATLAB:divideByZero" to suppress this warning.)

```
ans =
   -3.0000 -0.7500   NaN  0.3750 0.6000
```

Error using ==> /  
Matrix dimensions must agree.

Comments:

Examples of pathological results of dot division. In the first case, Matlab returns the result after displaying a warning concerning the division of a number with zero that leads to an infinity quantity. Observe that Matlab returns an Inf for such operations. Likewise, in the second case, the division of 0/0 is not defined and a Not-a-Number, NaN, is returned in the vector. In the third case, the inverse of each vector element is required, but Matlab returns an error since for such operation the “./” and not “/” should have been used instead.

**Example:** Examine the following limit:

$$\lim_{x \rightarrow \infty} \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Matlab's command:

```
>>x=[0.5 1 5 25 50 100 300];
>>(exp(x)-exp(-x))./(exp(x)+exp(-x))
```

Matlab's response:

```
ans =
    0.4621 0.7616    0.9999  1.0000  1.0000  1.0000  1.0000
```

Comments:  
Apparently, the limit converges to unity.

The dot power of vectors works in a similar way as with other dot products. Usually there is the need to square (or to rise to some other power) the elements of a vector. This can be done with the dot power operation as explained in the following example set.

Matlab's command:

```
>> d_power=sqrt(0:log(3):6)
>> d_power.^ 2, d_power.^ 4, d_power.* d_power, ans.*ans
```

Matlab's response:

```
d_power
0  1.0481  1.4823  1.8154  2.0963  2.3437

ans =
0  1.0986  2.1972  3.2958  4.3944  5.4931

ans =
0  1.2069  4.8278  10.8625 19.3112 30.1737

ans =
0  1.0986  2.1972  3.2958  4.3944  5.4931

ans =
0  1.2069  4.8278  10.8625 19.3112 30.1737
```

Comments:  
Examples of dot power. The second or the forth power of a certain vector can be defined either with the dot power or the dot product operation.

Before leaving this section, the reader should note that dot power of vectors is performed before any other operation, followed by the dot product and the dot division. Off course, if a complex vector command involves manipulations into brackets, these are executed before any other action.

#### **2.1.4.4 Some Useful Vector Functions**

Table 2 lists some useful function that can be used with row and column vectors. This list is by no way exhaustive.

**Table 2: Various function that can be used with vectors**

Basic Information		Elementary Matrices and Arrays	
disp	Display array	linspace	Generate linearly spaced vectors
display	Display array	logspace	Generate logarithmically spaced vectors
isempty	True for empty vector	ones	Create array of all ones
isequal	True if arrays are identical	rand	Uniformly distributed random numbers and arrays
length	Length of vector	zeros	Create array of all zeros
Operations and Manipulation			
cumprod	Cumulative product	mean	Average of array
cumsum	Cumulative sum	median	Median of array
end	Last index	min	Minimum elements of array
find	Find indices of nonzero elements	sort	Sort elements in ascending order
max	Maximum elements of array	sum	Sum of array elements

Use the help facilities to see how these functions can be used. The following illustrate the use of some of these functions.

Matlab's command:

```
>> clear; clc;
>> x=[]; isempty(x)
>> x=linspace(1, 10, 6), x=logspace(0.1, 0.5, 6); x(7)=0.1; x
>> sort(x), cumsum(x), [Max, I]=max(x), x(end)
```

Matlab's response:

```
ans = 1
```

```

x =
    1.0000    2.8000    4.6000    6.4000    8.2000   10.0000

    x =
    1.2589    1.5136    1.8197    2.1878    2.6303    3.1623    0.1000

    ans =
    0.1000    1.2589    1.5136    1.8197    2.1878    2.6303    3.1623

    ans =
    1.2589    2.7725    4.5922    6.7799    9.4102   12.5725   12.6725

Max =
    3.1623

I =
    6

ans =
    0.1000

```

Comments:

Examples with vector function. The first command instructs for the creation of an empty vector. The `isempty` function returns 1 for the true statement of an empty vector. Afterwards, `linspace` is used to generate a row vector of 6 linearly equally spaced points between 1 and 10 whereas `logspace` is used to generate a row vector of 6 logarithmically equally spaced points between decades  $10^{0.1}$  and  $10^{0.5}$ . “`x(7)=0.1`” instructs for the creation of a seventh element for the `x` vector. “`sort(x)`” sorts the elements of `x` in ascending order, “`cumsum(x)`” is a vector containing the cumulative sum of the elements of `x`, “[`Max, I`]=`max(x)`” returns in `Max` the maximum value of the vector `x` and in `I` the index of the maximum element. The last command, “`x(end)`” returns the last element of the `x` vector.

## 2.2 Two Dimensional Arrays (Matrices)

A  $m \times n$  matrix is a rectangular array that has  $m$  rows and  $n$  columns. For example, a 2-by-4 matrix can be the following one:

$$A_{2 \times 4} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$$

It is obvious that the above matrix can be decomposed to either 2 row vectors of  $1 \times 4$  dimensions or to 4 column vectors of  $2 \times 1$ . So, row and column vectors are special cases of a two dimensional array (matrix).



If we let,  $a_{11} = 1$ ,  $a_{21} = -2$ ,  $a_{12} = -2$ ,  $a_{22} = 2$ ,  $a_{13} = 5$ ,  $a_{23} = 4$ ,  $a_{14} = -3$  and  $a_{24} = 1$ , then the “A” matrix is reformed to:

$$A_{2 \times 4} = \begin{pmatrix} 1 & -2 & 5 & -3 \\ -2 & 2 & 4 & 1 \end{pmatrix}$$

The subscripts in each element of the matrix denote the element’s position. For instance, the position of  $a_{23}$  is 2<sup>nd</sup> row, 3<sup>d</sup> column. Matlab stores matrices using this rational. Having in mind that a two dimensional array is a composition of row vectors, we can use spaces (or commas) to define its row vectors and semicolons to separate them. The above matrix can be easily created as follows:

Matlab’s command:

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

Matlab’s response:

```
A =
     1  -2  5  -3
    -2   2  4   1
```

Comments:

Creation of a 2-by-4 matrix.

To show that row and column vectors are special cases or a two dimensional array (matrix) elaborate on the following examples:

Matlab’s command:

```
>> B=linspace(1, 4, 5); C=1:5; D=[-4:-1, NaN, Inf 1];
>> A1 = [B; C], A2=[A1; A1], A3=[B -9 -5;D]
```

Matlab’s response:

```
A1 =
     1.0000     1.7500     2.5000     3.2500     4.0000
     1.0000     2.0000     3.0000     4.0000     5.0000
```

```
A2 =
     1.0000     1.7500     2.5000     3.2500     4.0000
     1.0000     2.0000     3.0000     4.0000     5.0000
```

```

1.0000    1.7500    2.5000    3.2500    4.0000
1.0000    2.0000    3.0000    4.0000    5.0000

```

```

A3 =
1.0000    1.7500    2.5000    3.2500    4.0000   -9.0000   -5.0000
-4.0000  -3.0000  -2.0000   -1.0000     NaN         Inf    1.0000

```

Comments:  
Various manipulations with matrices.

The examples above, illustrate the case where a larger matrix can be created from smaller ones. Also, pay attention that Matlab returns an error if the user tries to combine row or column vectors with different lengths to create a two dimensional array.

Usually, it is imperative to store the size of a matrix in some variables. This can be done via a *build in Function* named as size:

Matlab's command:

```

>> [m1 n1]=size(A1); [m2 n2]=size(A2); [m3 n3]=size(A3);
>> M_N=[m1 n1; m2 n2; m3 n3]

```

Matlab's response:

```

M_N =
     2     5
     4     5
     2     7

```

Comments:  
Using the `size` function to find the size of various matrices. Each row of the `M_N` matrix contains the size of matrices `A1`, `A2` and `A3` respectively.

## 2.2.1 Transpose of a Matrix

Recall matrix “A” that was defined earlier. The transpose of “A”, symbolized in linear algebra as “A<sup>T</sup>” is the following:

$$A_{2 \times 4} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}, A_{2 \times 4}^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \\ a_{14} & a_{24} \end{pmatrix}$$

It is the same idea with the transpose of a row vector to a column one and vice versa. The transpose of an  $m \times n$  matrix is an  $n \times m$  one that is found if the rows of the original matrix become columns in the transposed one. Recall that the transpose in Matlab is performed with “'”. See the following examples to digest this issue.

Matlab's command:

```
>> A4=[1 2 3; 4 5 6], size(A4), A4', size(A4'), A5=[A4' A4']
```

Matlab's response:

```
A4 =
     1     2     3
     4     5     6

ans =
     2     3

ans =
     1     4
     2     5
     3     6

ans =
     3     2

A5 =
     1     4     1     4
     2     5     2     5
     3     6     3     6
```

Comments:

Exploiting the transpose of a matrix.

## 2.2.2 Elaborating on Parts of Matrices - The Colon Notation

As noted before, in Matlab, a matrix element position is indexed according to the row and column number. Recall the aforementioned “A” matrix:

$$A_{2 \times 4} = \begin{pmatrix} 1 & -2 & 5 & -3 \\ -2 & 2 & 4 & 1 \end{pmatrix}$$

If someone wants to extract the elements:  $a_{12}$ ,  $a_{23}$ , and  $a_{24}$ , this can be done in the following way (A has already been created and should be stored in the

workspace; use the *who function* to view the variables that you have saved in the workspace or view them from the *workspace browser*):

Matlab's command:

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

Matlab's response:

```
ans =  
    -2
```

```
ans =  
     4
```

```
ans =  
     1
```

Comments:

Extracting specific elements from a matrix.

So, to refer to a certain matrix element, after the matrix name the element's index/position is enclosed in parenthesis. This is similar with the indexing of vectors elements with the exception that in the case of matrices, two indexes should be used (the first one indicates the row number whilst the second the column number). In general, the syntax for manipulating a matrix element in Matlab is:

"K(row\_index, column\_index)"

where K is an  $m \times n$  matrix, *row\_index* indicates the row in which an element lays and *column\_index* the according column. The upper left position in a matrix has (1,1) coordinates. *row\_index* can take all integer values smaller than m, inclusive, while *column\_index* can also take all integer values smaller than n, inclusive (note that Matlab can handle multidimensional arrays just by putting additional indexes to control the higher dimensions).

Moreover, Matlab has an additional way to refer to matrix elements just by using a single index number in the brackets after the declaration of the matrix name. Starting from the upper left element that represents the first element (1) of the matrix, further reference to additional elements is done incrementally one by one and column-wise. That is, the element with

position (2, 1) is taken to be the second (2), the (3, 1) the third and finally the (m, n) element is the last one (n\*m). View the following example to understand this kind of matrix indexing.

Matlab's command:

```
>> H=magic(4), H(1), H(2), H(9), H(16)
```

Matlab's response:

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

```
ans =
    16
```

```
ans =
     5
```

```
ans =
     3
```

```
ans =
     1
```

Comments:

Extracting specific elements from a matrix using single indexing.

For matrices, the colon notation “:” plays an important role. Colon can be used to extract either a specific part of a matrix, or to view a whole row, a whole column or a sub-matrix extracted from the original one. Earlier, the colon was used to define vectors and it has been explained that it creates elements from a *starting\_value* through a *finishing\_value* equally spaced according to a defined *step* (if the step is not defined, then the step is taken to be 1). In Matlab, a certain row of a matrix can be extracted by placing the colon in the *row\_index*, “K(:,column\_index)”, and a certain column by placing the colon in the *column\_index*, “K(row\_index,:)”. A sub-matrix that is composed by the *r1* to *r2* rows ( $r1=r2$ ) and *c1* to *c2* columns ( $c1=c2$ ) of the original matrix can be retrieved according to the following syntax:

“K(r1:r2, c1:c2)”

In addition, there are special cases where the *step* is defined to be a different integer number other than 1. Some paradigms follow:

Matlab's command:

```
>>A(:,1), A(1,:), A(:,1:3), A=[A', A'.*2], A(2:3,3:4)
>> A(1:3,1:2)=A(2:end,3:end)
```

Matlab's response:

```
ans =
     1
    -2
ans =
     1    -2     5    -3

ans =
     1    -2     5
    -2     2     4

A=
     1    -2     2    -4
    -2     2    -4     4
     5     4    10     8
    -3     1    -6     2

ans =
    -4     4
    10     8

ans =
    -2    -4

A=
    -4     4     2    -4
    10     8    -4     4
    -6     2    10     8
    -3     1    -6     2
```

Comments:

The first command, instructs for the extraction of the first column whilst the second for the extraction of the first row. The following one instructs for the extraction of the 1<sup>st</sup>, 2<sup>nd</sup> and 3<sup>rd</sup> columns. Afterwards, A is re-built; the following command instructs for the extraction of the sub-matrix that is composed from the elements of A that are included in: 2<sup>nd</sup> and 3<sup>rd</sup> row - 3<sup>rd</sup> and 4<sup>th</sup> columns. The following command is a special case where the step in the colon expression is different than 1. The last command is a more tricky use of the colon.

### 2.2.3 Matrix Basic Manipulations

Subtraction and multiplication with a scalar are similar as with the vector case. Matrix dimensions (size) must match when performing manipulations with addition and subtraction. Some examples follow:

Matlab's command:

```
>> J=[-1 2 3 1; 5 2 4 2; 1 1 2 0]; I=[1 4 5 2; 8 7 5 1; -1 1 -1 1];  
>> J+2*I, J(:,1)+I(:,2), J(1,:)-1/4*I(2,:)
```

Matlab's response:

```
ans =  
     1    10    13     5  
    21    16    14     4  
    -1     3     0     2  
  
ans =  
     3  
    12  
     2  
  
ans =  
 -3.0000    0.2500    1.7500    0.7500
```

Comments:

Basic matrix addition, subtraction and scalar multiplication.

### 2.2.4 Matrix Manipulations Related to Products, Division, and Powers

Similar rules as with the case of vectors apply to matrix manipulations related with dot operations and products of matrices.

#### 2.2.4.1 Matrix Dot Product, Division and Powers

It is straightforward to generalize the properties of dot operations from the case of vectors to the case of two-dimensional arrays. The operations are again performed element-by-element. Work on the following examples to learn the use of dot operations with matrices (remember which operations come first).

Matlab's command:

```
>> clear; U=[-1 2 1; 0 -2 3]; V=[1:3; 5 2 4];  
>> U.^2, U.*V, (1./V).^2.*U, U.^U
```

Matlab's response:

ans =

```
1 4 1  
0 4 9
```

ans =

```
-1 4 3  
0 -4 12
```

ans =

```
-1.0000 0.5000 0.1111  
0 -0.5000 0.1875
```

ans =

```
-1.0000 4.0000 1.0000  
1.0000 0.2500 27.0000
```

Comments:

Dot operations with two-dimensional arrays.

#### **2.2.4.2 Matrix to Vector Product - Matrix to Matrix Product**

A matrix can be multiplied with a vector or a matrix as long as the following rule concerning their dimensionality/sizes holds:

$$(m \times n) \times (n \times l)$$

That is, the first's matrix columns equal the second's rows. A matrix "A" with dimensions  $m \times n$  can be multiplied with a matrix "B" with dimensions  $n \times l$  resulting to a matrix "C" with dimensions  $m \times l$ :

$$C_{m \times l} = A_{m \times n} \times B_{n \times l}$$

The above matrix "A" can be multiplied with a  $1 \times m$  row vector "R1" from left to produce a  $1 \times n$  row vector "R" while "A" can be multiplied with an  $n \times 1$  column vector "C1" to the right to produce a column vector "C" with  $m \times 1$  dimensions:

$$R_{1 \times n} = R1_{1 \times m} \times A_{m \times n}$$

$$C_{m \times 1} = A_{m \times n} \times C1_{n \times 1}$$

Let's define the "A" and "B" matrices:



$$A_{2 \times 4} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}, B_{4 \times 2} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix}$$

Like linear algebra, Matlab products with matrices (and/or vectors) are calculated as follows:

$$C_{2 \times 2} = A_{2 \times 4} \times B_{4 \times 2} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

$$C_{2 \times 2} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} \end{pmatrix}$$

That is, the element in the (1,1) position of the new matrix “C” is the *inner* product of the 1<sup>st</sup> row of “A” with the 1<sup>st</sup> column of “B”, the (1,2) element of “C” is the *inner* product of the 1<sup>st</sup> row of “A” with the 2<sup>nd</sup> column of “B” and so on. Work through the following examples to digest this topic.

Matlab’s command:

```
>> clear; A=[ 1 -2 5 -3; -2 2 4 1]; B=[2 3;1 0;2 -1;1 4]; A_B=[A;B'];
>> R1 = [-2 -1 3 2]; C1 = [1; -1; 2; 0];
>> C=A*B, Row=R1*A_B, Col= A_B*C1, A*(A_B*A_B)*B
```

Matlab’s response:

```
C =
    7 -14 7 -6

Row =
    12    5 -10    16

Col =
    13
     4
     5
     1

ans =
    39 -310 297 314
```

Comments:

Matrix-vector and matrix-matrix products.

## 2.2.5 Special Cases of Matrices

Matlab has some very useful *build-in* matrices *function* that ease the manipulation of two dimensional arrays. These *function* usually take as input (at least) the matrix dimension and return a certain result. Table 3 exhibits these very useful functions accompanied with a short explanation of their use.

**Table 3: Various function that can be used with vectors.**

eye(m,n)	In the case where m is equal to n this returns an identity matrix
ones(m,n)	Creates an m-by-n matrix of ones
diag(V)	When V is a <i>m-by-m</i> matrix returns the elements of the main diagonal.
rand(m,n)	Creates an <i>m-by-n</i> matrix with random entries
zeros(m,n)	Creates an <i>m-by-n</i> matrix of zeros

Examples with the above matrix-function follow:

Matlab's command:

```
>> eye(3,2), ones(2,3), diag([2 3; 1 -2]), rand(2,2), zeros(1,2)
```

Matlab's response:

```
ans =  
 1  0  
 0  1  
 0  0
```

```
ans =  
 1  1  1  
 1  1  1
```

```
ans =  
 2  
-2
```

```
ans =  
 0.9501  0.6068  
 0.2311  0.4860
```

```
ans =  
 0 0
```

Comments:

Examples with matrices *build-in functions*.

## 2.2.6 Additional Useful Matrix Functions

Functions that are exhibited in Table 1 can also be applied to a two dimensional dimensional arrays because vectors are special cases of matrices (and vice versa). Additional *build-in functions* are illustrated in

Table 4.

**Table 4: Various function that can be used with vectors.**

Basic Information		Elementary Matrices and Arrays	
ndims	Number of dimensions	blkdiag	Block diagonal concatenation
numel	Number of elements		
Operations and Manipulation			
det	Determinant	rank	Matrix rank
expm	Matrix exponential	sortrows	Sort rows in ascending order
fliplr	Flip matrices left-right	sqrtm	Matrix square root
flipud	Flip matrices up-down	trace	Sum of diagonal elements
inv	Matrix inverse	tril	Lower triangular part of matrix
logm	Matrix logarithm	triu	Upper triangular part of matrix
norm	Matrix or vector norm		

Moreover, type: “help matfun” to view some very useful matrix *functions* related to numerical linear algebra.

## 2.2.7 Example: System of Linear Equations

Many times, a researcher has to solve a system of linear equation simultaneously. In matrix notation, such problem is formulated as:

$$Ax = b$$

where “A” is a coefficient matrix, “x: is the set (column) of unknowns and “ b” is a column vector. In an equation-wise format, the above is equal to:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \dots + a_{1m}x_m &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \dots + a_{2m}x_m &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nm}x_m &= b_n
\end{aligned}$$

The solution of the above system can be expressed as:

$$x = A^{-1}b$$

given that “ $A^{-1}$ ” (that represents the inverse matrix of “ $A$ ”) exists and the number of equation,  $n$ , is equal or greater than the number of unknowns,  $m$ . A unique solution to the above equation system exists when:  $n=m$ .

Matlab has standard and efficient specialized routines to solve such systems like the `mldivide` function (called alternatively with “`\`”). Otherwise, a quick way (but sometimes non practical and inaccurate if the number of equations is extremely large) to solve the above system is via the inverse function, `inv` (note that the `inv` can be used only with square matrices). View the following example to understand how you can solve a small system of linear equations:

Matlab’s command:

```
>> clear; A=[2 4 3; -2 -4 2; 6 4 2]; b=[2 ;4;1];
>> x1=inv(A)*b, x2=A\b
```

Matlab’s response:

```
x1 =
    0.0500
   -0.4250
    1.2000

x2 =
    0.0500
   -0.4250
    1.2000
```

Comments:

Two ways for solving a system of linear equations.

To see that these two procedures lead to different results, elaborate on the following example (search online help to see the use of the command: `pascal`):

Matlab's command:

```
>> clear; A=pascal(10); b=[5; -2; 4; 8; 1; 2; 0; -6; 2; 3];  
>> x1=inv(A)*b; x2=A\b;  
>>diff=x1-x2
```

Matlab's response:

```
diff =  
-1.0328 e-006  
9.0204 e-006  
-3.4539 e-005  
7.6293 e-005  
-0.00010781  
0.0001013  
-6.3407 e-005  
2.5521 e-005  
-5.9979 e-006  
6.2738 e-007
```

Comments:

For even moderate linear systems, the two methods (inv and “\”) lead to different solutions. If high precision is required, then the researcher should prefer the “\”.

The most proper way for large systems is the use of “\”. See the help facilities for details on `inv` and on “\”.

## 3 Plots and Graphs (2D and 3D)

It is quite easy to create a plot or a graph by using the variables or parameters that have already been stored in the Matlab's *workspace*. Matlab offers a variety of *build-in Function* for creating simple two dimensional plots, 3 dimensional surface plots, to combine plots to a larger one and so on. The following subsections are a very brief and an elementary reference to the visualization capabilities of Matlab.

### 3.1 Creating 2D Line Plots

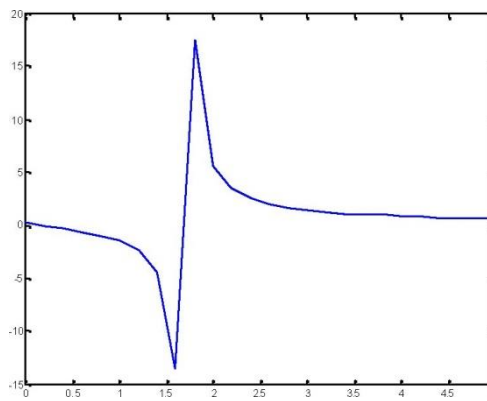
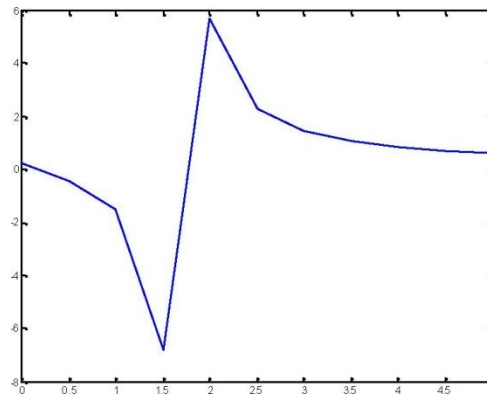
The basic function for the creation of a simple 2D line plot is called: `plot`. Let be a row or column vector with real data named `Y`, with elements “ $y_1, y_2, \dots, y_n$ ”. If `plot` is called as: “`plot(Y)`”, then a linear plot of the elements of `Y` versus its index will appear (the `plot` function creates/plots the pairs:  $(1, y_1), (2, y_2), \dots, (n, y_n)$  and connects them with a line). If for each “ $y_i$ ” we have an accompanied “ $x_j$ ” coordinate, then the function “`plot(X,Y)`” plots all pairs:  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$  and connects them with a line. Note that vectors `Y` and `X` must share the same length. Additionally, if the data to be plotted are stored in a two dimensional array, then the arguments for the *plot function* can be done via the use of colon notation “:”. It is very easy to plot the following function in the range  $[0,5]$ :

$$y = \frac{2x^2 + 5x - 1}{x^3 - 5}$$

Matlab's command:

```
>> clear; x=0:0.5:5; y=(2*x.^2+5*x-1)./(x.^3-5); plot(x,y);  
>> x=0:0.2:5; y=(2*x.^2+5*x-1)./(x.^3-5); plot(x,y);
```

Matlab's response:

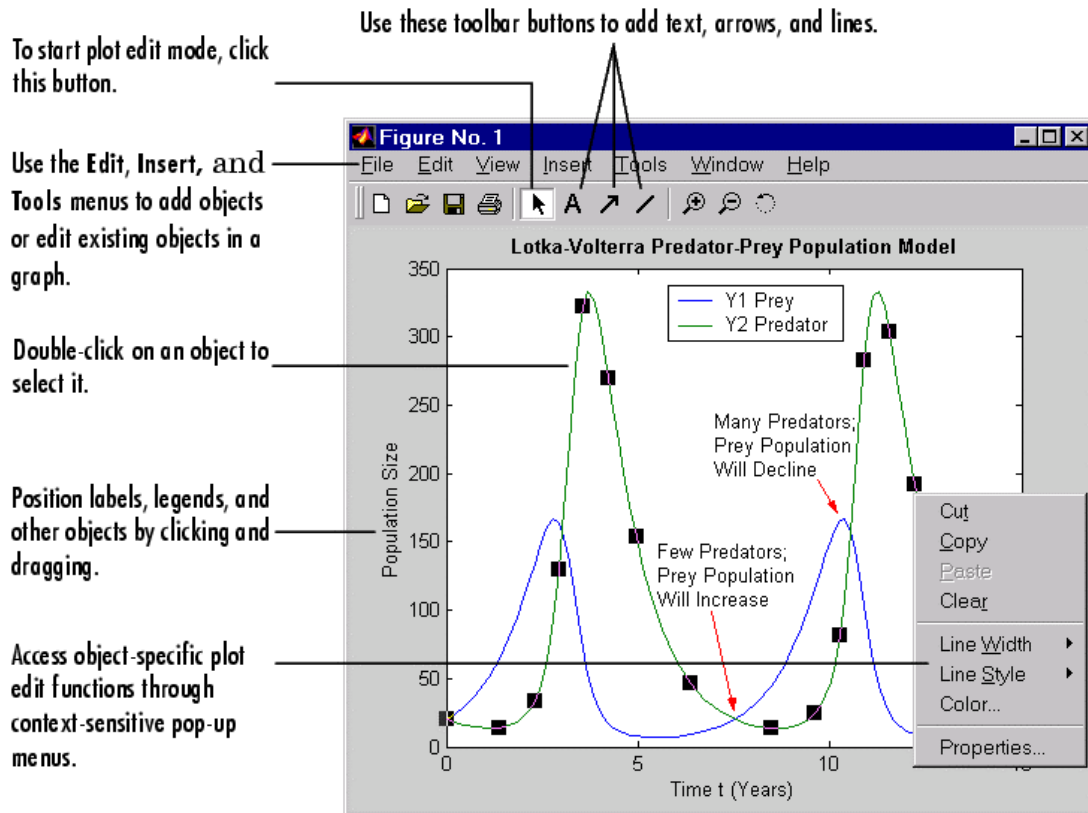


Comments:

Plot of a function. The use of dot operations is prominent for this task. Note that the plot's smoothness is controlled by the density of the x vector. In the second plot where the x is denser, the function plot is smoother.

### 3.1.1 Plot Edit Mode

If you have already type the aforementioned commands on the Matlab *command window*, you will have noticed that the plots that you see, appear in a *figure window* (i.e. Figure 8) that offers a variety of tools that allow you to edit and customize the plot under consideration by using a point and click style editing mode.



**Figure 8: The figure window**

This *figure window* has tools that look like a photo editor (e.g. Microsoft® Photo Editor). You can select objects in a graph, cut, copy, paste, move and resize objects, zoom in and zoom out at certain areas of the plot, change the settings of the axes, print or save the figure for a later use or even to export the plot in a certain format (e.g. \*.emf, \*.bmp, \*.jpg, \*.tif, etc). Create a figure and play around with the various tools to learn their use.

The plots that are included in this handout (except the ones that were copied from [2]) have been exported from the *figure window* in an enhanced metafile format (\*.emf). To export a plot, from the figure window go: *File>Export*, and from the window that appears choose the exporting format, give a name to the file and save it in a folder of your choice. By doing this, you can easily insert plots from Matlab to a text editor like Microsoft® Word.



### 3.1.2 Function and Style Facilities Related to Plots

The blue solid line is the default style for the plots. This of course can change. Matlab gives the user the option to define a different line colour and to mark each plotted point with a symbol. To do this, the user should call the plot function as:

`plot(X,Y, '???)'`

where “?” represents the color, “?” the mark style and “?” the line style (if not given, then Matlab uses a default style). Table 5 tabulates the plot facilities (see also the online help).

**Table 5: Plot colour and line styles**

Symbol	Colour(?)	Symbol	Line Style (?)
g	Green	-,:,-,--	Solid, dotted, dash dot and dashed
r	Red		
c	Cyan	<b>Symbol</b>	<b>Mark Style</b>
m	Magenta	., o, x	Point, circle and x-mark
y	Yellow	+ , *, s	Plus, star and square
k	Black	d, v	Diamond and triangle down
		^, <, >	Triangles up, left and right
		p, h,s	Pentagram, hexagram and solid

Moreover, Table 6 lists additional commands that are useful when using the plot *function*.

**Table 6: Various functions that help in the creation of 2D-plots.**

Function	Comment
axis	Sets the minimum and maximum values of axes.
clf	Clear the current figure.
close (all)	Closes the current (all) figure(s).
figure	Creates a new figure window.
grid on / off	Adds/removes major grid lines to the current axes.
gtext('text')	Adds text to a figure manually, by the use of mouse.
hold on/off	hold on / off holds the current plot and all axis properties so that subsequent graphing commands add to the existing graph.
legend('text')/legend off	Adds (removes) a legend to the graph.
loglog	The same as with plot but logarithmic scale axes

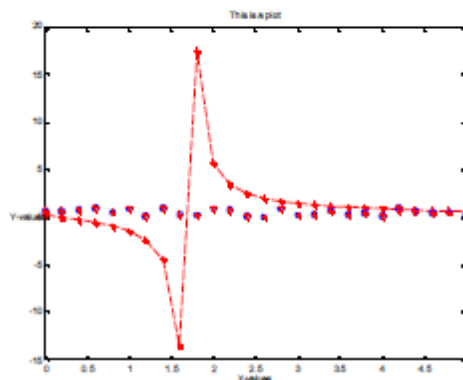
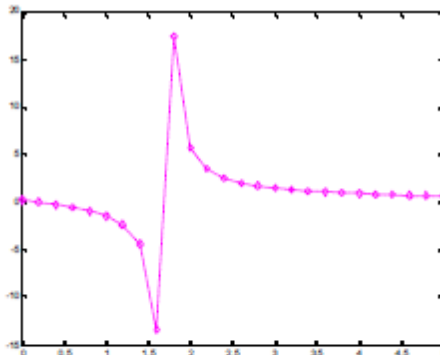
	are used.
semilogx	The same as with plot with the exception that logarithmic scale is used for X-axis.
semilogy	The same as with plot with the exception that logarithmic scale is used for Y-axis.
subplot	Breaks the current figure to subplots.
title('text')	Adds a title above the figure.
xlabel('text')	X-axis label
ylabel('text')	Y-axis label

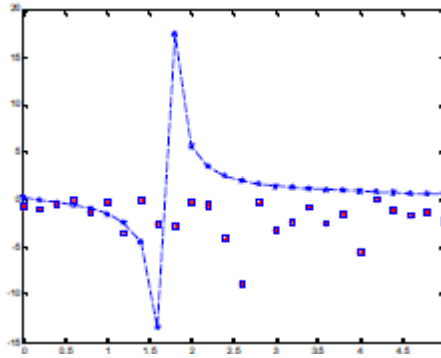
View the following examples to digest the plot facilities.

Matlab's command:

```
>> plot(x, y, 'md: '); figure; plot(x, y, 'r*-'); hold on;
>> y1=rand(1,length(y)), plot(x,y1, 'bo'); plot(x,y1, 'r+');
>> xlabel('X-values'); ylabel('Y-values'); title('This is a plot!');
>> close all; plot(x,y, 'bp--', x, log(y1.^2), 'rh', x, log(y1.^2),'bs');
```

Matlab's response





Comments:

Experimenting with various function and facilities related to the plots. The last plot command shows how few plots can be depicted in one figure without using the hold on command.

The following examples, concern the creation of some subplots in one *figure window*. If you view the online help for the subplot command, you will see that it takes three input arguments. That is, it is called as:

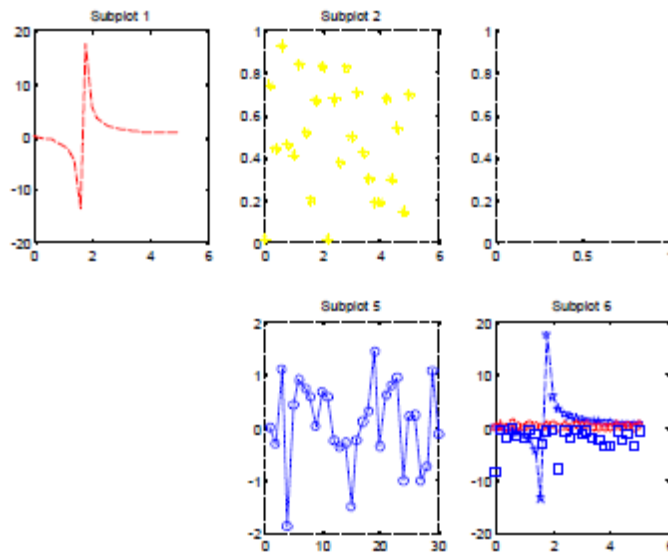
“subplot(m,n,p)”

It breaks the *figure window* into an *m-by-n* matrix of small axes, selects the *p*-th axes for the current plot, and returns the axis handle. The axes are counted along the top row of the *figure window*, then the second row, etc. The subplot function instructs which subplot is handled at the current time. After the subplot command, a plot command should follow. See the following examples:

Matlab’s command:

```
>> close all; subplot(2,3,1); plot(x,y, 'r--'); title('Subplot 1');
>> subplot(2,3,2); plot(x,y1, 'y*'); title('Subplot 2');
>> subplot(2,3,3);
>> subplot(2,3,5); plot(randn(1,30), 'bo:'); title('Subplot 5');
>> subplot(2,3,6); plot(x,y, 'bp--', x, (y1.^2), 'rh', x, log(y1.^2),'bs');
>> title('Subplot 6');
```

Matlab's response:



Comments:

Experimenting with the subplot command. The first two input arguments to the subplot function define the dimensionality of the figure window. If  $m$  is the first input arguments and  $n$  the second, then the figure window breaks to an  $m$ -by- $n$  plot matrix. The third input to the subplot function designates the current subplot to be handled. Starting from the upper left corner and moving row-wise to the lower right corner, 1 refers to the first subplot, 2 to the second and  $mn$  to the last one. After each subplot command, the plot command should follow. In the current examples, the commands instruct for the creation of a 2-by-3 subplot matrix. The 1<sup>st</sup>, 2<sup>nd</sup>, 5<sup>th</sup> and 6<sup>th</sup> subplots are used whilst the 3<sup>rd</sup> and 4<sup>th</sup> are currently unused. Note that if the subplot command is avoided for any subplot, then its space on the figure window stays empty. If only the subplot is called as with the case of the third subfigure, then a blank figure appears in the figure window.

Moreover, Matlab offers a variety of options to customize the graph. For example, a more general calling syntax for the plot is:

`plot(X, Y, '???', 'PropertyName', PropertyValue, ...)`

where “PropertyName” relates to an element of the plot and “PropertyValue” is defined by the user. Some useful properties concern the following:

- “LineWidth” - specifies the width (in points) of the line.
- “MarkerEdgeColor” - specifies the colour of the marker or the edge colour for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- “MarkerFaceColor” - specifies the colour of the face of filled markers.

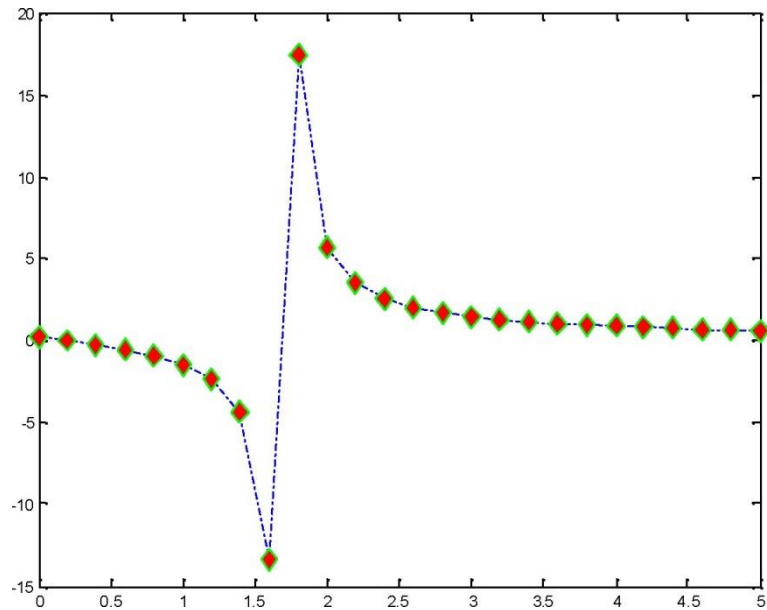
- “MarkerSize” - specifies the size of the marker in units of points.

View the following examples to conceptualize this issue:

Matlab's command:

```
>> plot(x,y,'bd-.', 'LineWidth', 2, 'MarkerSize', 11, ...  
        'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'g')
```

Matlab's response:



Comments:

An example of a plot that uses some of the plot properties.

Besides these 2D visualization facilities, Matlab can be used to create bar charts, pie charts, histograms, stems plot, staircase plots, errorbars etc. Use the online help and the *help browser* to see how you can use these visualizations facilities and learn more about the plot properties.

## 3.2 Creating 3D Graphs

Matlab offers fancy 3-dimensional line plots and surface graphs.

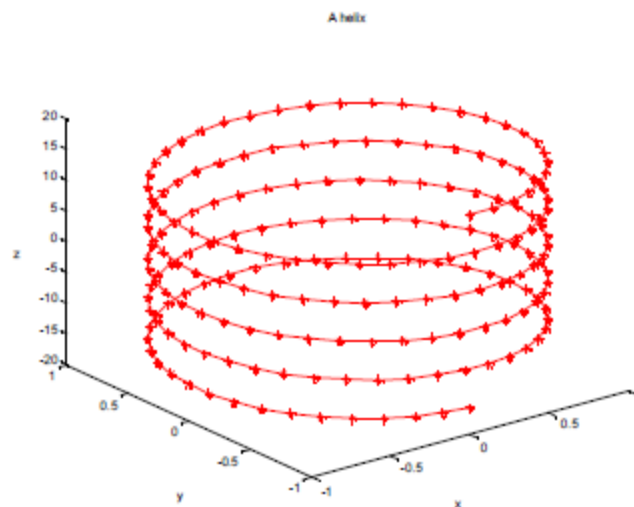
### 3.2.1 Creation of 3D Line Plots

There is the *function*: `plot3` which its use is similar as with the *function*: `plot`, with the difference that an additional input argument corresponding to the additional dimension is required (the additional dimension is symbolized as “z”). `plot3` is used to plot lines or point coordinates in the 3D space. Let use the `plot3` to create a 3D helix:

Matlab's command:

```
>> clear; g=linspace(-5*pi,5*pi,200); plot3(sin(g),cos(g),g,'r*-');  
>> xlabel('x'); ylabel('y'); zlabel('z'); title('A helix');
```

Matlab's response:



Comments:

Creating a helix with the `plot3` function.

### 3.2.2 Creation of 3D Mesh and Surface Graphs

There are also many commands that lead to the creation of 3D surfaces. Recall that to create a surface of two variables, at each pair of  $(x,y)$  a

functional expression,  $z=f(x,y)$ , is evaluated. Usually, we evaluate the  $z$  behaviour in a certain area of the  $(x,y)$ -plane in which  $z$  has some interesting properties (e.g. it presents a minimum, a maximum or a saddle point). Suppose  $x$  vector that defines the  $x$ -axis (abscissa) and a  $y$  vector that defines the  $y$ -axis (ordinate) exist and define the  $(x,y)$ -plane on which we would like to create the 3D surface of  $z=f(x,y)$ . To plot the surface, it is needed to create a grid of sample points (most preferable with high density and this is defined by the difference between the elements of  $x$  and  $y$ ) that covers the rectangular domain of the  $(x,y)$  plane in order to generate  $X$  and  $Y$  matrices consisting of repeated rows and columns of  $x$  and  $y$ , respectively, over the domain of the function. Then these matrices will be used to evaluate and graph the function.

The `meshgrid` function transforms the domain specified by two vectors,  $x$  and  $y$ , into matrices,  $X$  and  $Y$ . You then use these matrices to evaluate the  $z=f(x,y)$ . The rows of  $X$  are copies of the vector  $x$  and the columns of  $Y$  are copies of the vector  $y$ .

We will see how we can plot a 3D surface via an example. Let say that we need a graph of the well know peaks function that has the following functional form:

$$Z = f(x,y) = 3(1-x)^2 e^{-(x^2-(y+1)^2)} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-(x^2-y^2)} - \frac{e^{-(x+1)^2-y^2}}{3}$$

We want to examine this function in the rectangular space of  $(x,y)$ -plane defined as:

$$-2 < x < 2$$

$$-4 < y < 4$$

Since Matlab understands combinations of  $(x,y,z)$  pairs only, it is needed to evaluate  $Z$  at various points (the number of these points defines the surface smoothness) that lay in the above rectangular space. `meshgrid` can be used to create the desired grid of points. The following code lead to the generation of the grid:

Matlab's command:

```
>> clear; x=-2:1:2; y=-4:1:4; [X,Y]=meshgrid(x,y)
>> plot(X,Y, 'rh'); axis([-3 3 -5 5]); xlabel('x-axis'); ylabel('y-axis');
>> title('Meshgrid');
```

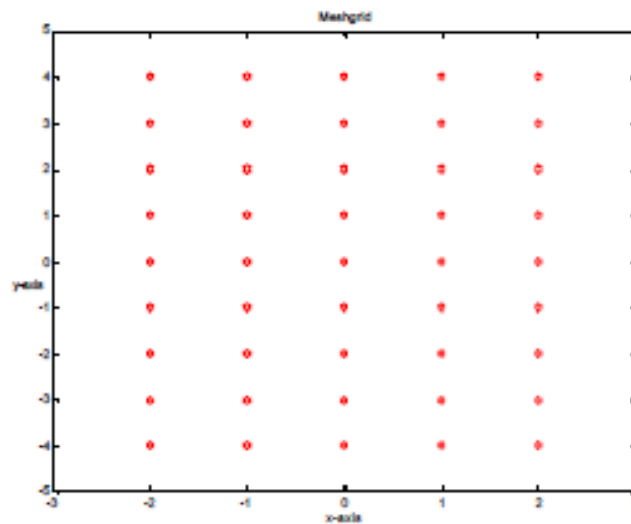
Matlab's response:

X=

-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2
-2	-1	0	1	2

Y=

-4	-4	-4	-4	-4
-3	-3	-3	-3	-3
-2	-2	-2	-2	-2
-1	-1	-1	-1	-1
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4



Comments:

Creating the grid necessary for plotting a 3D surface. `meshgrid` takes as inputs the vectors `x` and `y` that define the (x,y)-plane that should be used to evaluate `Z` and returns two matrices (`X` and `Y`) that if taken



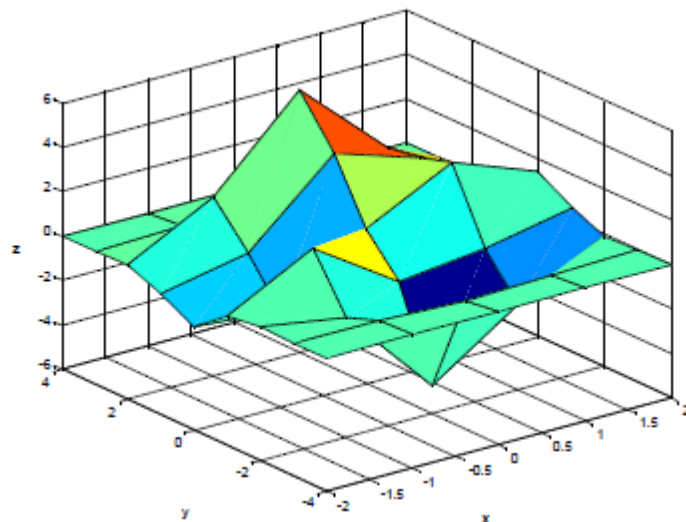
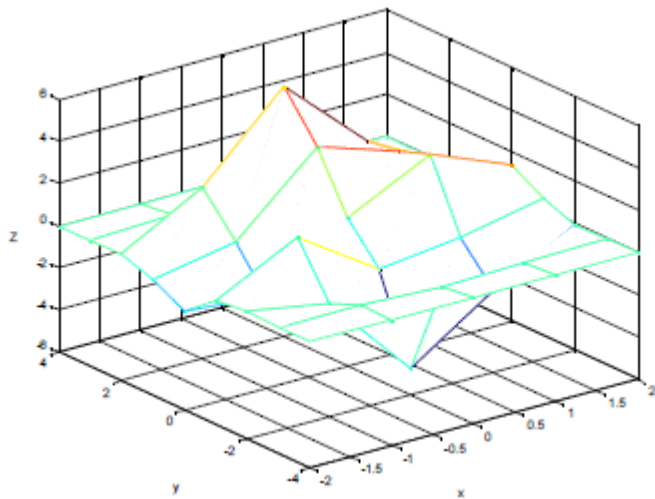
together, generate various  $(x,y)$  pairs that lay in the rectangular domain. The subsequent 2D plot of  $X$  and  $Y$  shows how the grid is sampled.

Now, we should evaluate the  $Z$  at each pair of the grid and generate the surface plot. This is done with the following code:

Matlab's command:

```
>> Z = 3*(1-X).^2.*exp(-X.^2 - (Y+1).^2) ...  
      -10*(X/5-X.^3-Y.^5).*exp(-X.^2-Y.^2) ...  
      -1/3*exp(-(X+1).^2 - Y.^2);  
>> figure(1); mesh(X,Y,Z); xlabel('x'); ylabel('y'); zlabel('Z');  
>> figure(2); surf(X,Y,Z); xlabel('x'); ylabel('y'); zlabel('Z');
```

Matlab's response:



**Comments:**

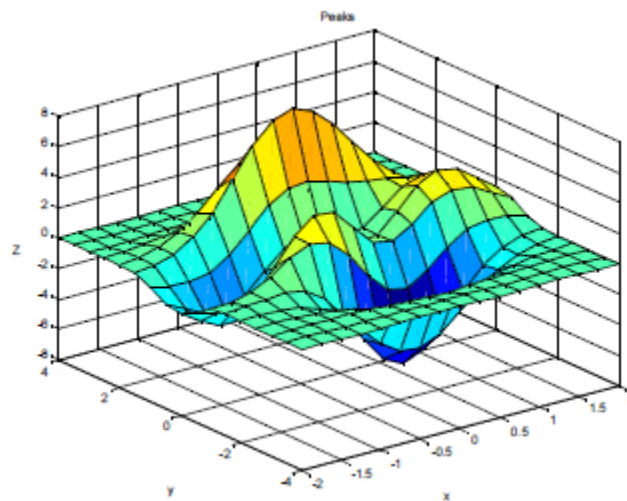
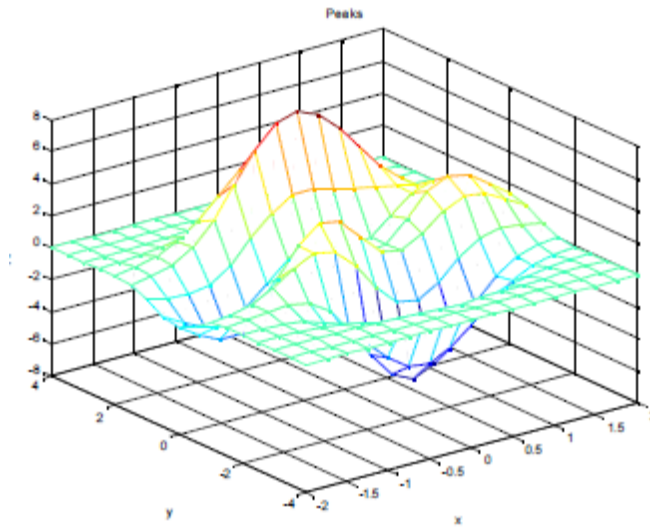
Creating the 3D mesh and a 3D surface. The function is not smooth because the dense of the x and y was too low.

With the above code, a 3D mesh and a 3D surface is created. It is apparent that the surfaces are not smooth because the original density of x and y vectors was too low. In the following, we re-create the mesh and surface with higher densities.

**Matlab's command:**

```
>> x=-2:0.25:2; y=-4:0.5:4; [X,Y]=meshgrid(x,y);  
>> Z =3*(1-X).^2.*exp(-X.^2 - (Y+1).^2) ...  
      -10*(X/5-X.^3-Y.^5).*exp(-X.^2-Y.^2) ...  
      -1/3*exp(-(X+1).^2 - Y.^2);  
>> figure(1); mesh(X,Y,Z); xlabel('x'); ylabel('y'); zlabel('Z'); title('Peaks')  
>> figure(2); surf(X,Y,Z); xlabel('x'); ylabel('y'); zlabel('Z'); title('Peaks');
```

Matlab's response:



Comments:

Creating the 3D mesh and a 3D surface with a higher density meshgrid.

To find what other useful commands can be accompanied the mesh and surf commands, use the online help.

### **3.2.2.1 Examination of a Function's Critical Points**

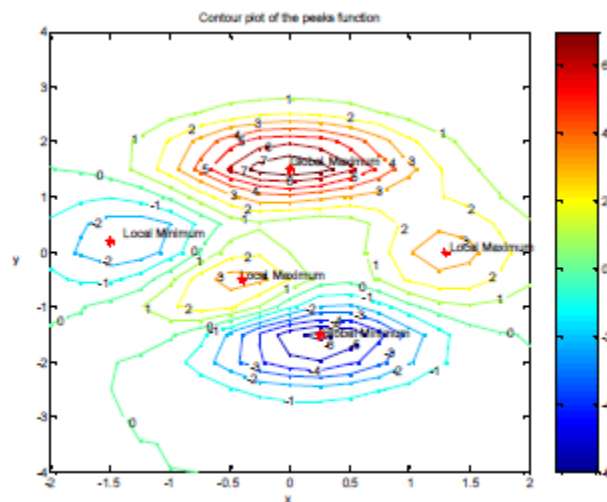
Many times, a researcher is faced with an unconstrained optimization problem. That is, it has a two variable function  $Z=f(x,y)$  and wants to examine this function in a certain area in order to locate any minimum, maximum or saddle point(s). From the previous section, it is apparent that

in the range  $[-10,10]$  the peaks function seems to have various critical points. In optimization theory, a critical point can be located via the use of a contour plot. A contour plot depicts the level curves of  $f(x,y)$  for some values  $V$ . In other words, we project the sections/incision of various heights of the  $Z=f(x,y)$  function in the  $(x,y)$ -plane. Matlab offers such *function* named as: `contour`. Experiment with the following code to see the usefulness of contour plots.

Matlab's command:

```
>> V=-10:1:10; [c,h] = contour(x,y,Z,V); clabel(c,h), colorbar; hold on;
```

Matlab's response:



Comments:

Creating the contour plot of the peaks function. Viewing this plot, it is easy to identify the critical points of the function. There is one global maximum, one global minimum, two local maxima and a local minimum. Additionally, from the color-bar that is created with the command: `colorbar`, and from the values printed next to the contour curves, it is obvious that the peaks function around its critical points exhibits a flatness (saddle points) (Note: the "\*" that indicate a critical point are plotted approximately); the text next to each critical point was placed with the `gtext` command.

Alternatively, we can approximate the maximum of the peaks function by searching exhaustively the  $Z$  matrix (to do so, we should re-define the  $x$  and  $y$  vectors to have high dense). This is done with the following code:

Matlab's command:

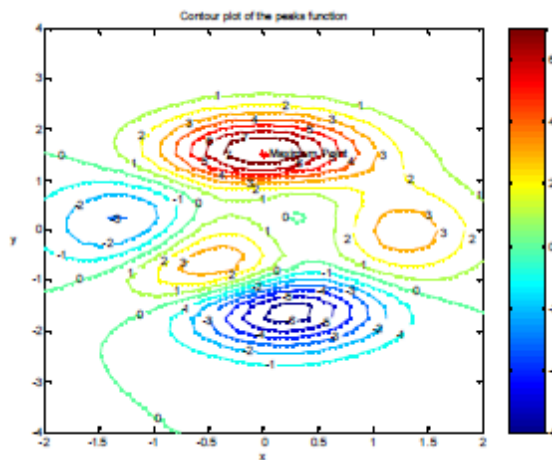
```
>> close all; x=-2:0.05:2; y=-4:0.25:4; [X,Y]=meshgrid(x,y);
```

```

>> Z = 3*(1-X).^2.*exp(-X.^2 - (Y+1).^2) ...
      -10*(X/5-X.^3-Y.^5).*exp(-X.^2-Y.^2) ...
      -1/3*exp(-(X+1).^2 - Y.^2);
>> V=-10:1:10; [c,h] = contour(x,y,Z,V); clabel(c,h), colorbar; hold on
>> xlabel('x'); ylabel('y'); title('Contour plot of the peaks function');
>> Fmax=max(max(Z)); P=find(Z==Fmax); xy=[X(P) Y(P)];
>> plot(xy(1),xy(2),'rp', 'MarkerSize',10);
>> text(xy(1)+0.05,xy(2), 'Maximum Point');

```

Matlab's response:



Comments:

Locating the maximum of the peak function via an exhaustive search. “Fmax=max(max(Z))” stores in Fmax the maximum value of the Z matrix that is equal to 7.9966. “P=find(Z==Fmax)” returns in P the element index (a single value) where Z is equal to Fmax; specifically the Z element with index number 1343 has the value of 7.9966. Finally, in xy we store the x-and-y coordinates of the maximum point. ‘MarkerSize’ in the plot command creates a larger size mark symbol.

For practice try to write your own code to find the global minimum of the peaks function (you can also try to find the local minima and maxima but this is tricky).

## 4 Control Flow

A collection of commands that include repetition of a conditional code segment is termed as *control flow* structure. In Matlab as in any programming language, *control flow* commands are presented via the execution of a conditional expression (e.g. execute a code segment if a statement is true) that is based on logical and/or relational operators.

### 4.1 Logical and Relational Operators

To evaluate a conditional code, it is needed to have a Boolean expression that takes two possible values: TRUE **or** FALSE. Matlab defines a TRUE statement with the value of 1 and a FALSE with the value of 0. In Matlab there are six relational operators: “<” (less than), “<=” (less than or equal to), “>” (greater than), “>=” (greater than or equal to), “==” (equal to), and “~=” (not equal to). These can be used to compare a variable to a scalar, a vector to a vector or a multidimensional array to another one. Always the comparisons are done element by element and the result is a scalar/vector/matrix of the same size as then ones originally used, with elements set to one where the relation is true and elements set to zero where it is not.

In addition, Matlab has (at least) 3 logical operators: “&” or “&&” (logical AND), “|” or “||” (logical OR), and “~” (logical complement - not). If we have two matrices “A” and “B” that share the same dimensionality then:

- “A” & “B” is a matrix whose elements are 1's where both “A” and “B” have non-zero elements, and 0's where either has a zero element. “A” and “B” must have the same dimensions (or one can be a scalar). For example in the case of scalars: 1 & 0, 0 & 1 and 0 & 0 will result to 0 (FALSE) while 1 & 1 is 1 (TRUE).
- “A” | “B” is a matrix whose elements are 1's where both “A” or “B” has a non-zero element and 0's where both have zero elements. “A” and “B” must have the same dimensions (or one can be a scalar). For

example and for the case of a scalar:  $0 | 1$ ,  $1 | 0$ , and  $1 | 1$  is 1 (TRUE) while  $0 | 0$  is 0 (FALSE).

"~A" is a matrix whose elements are 1's where "A" has zero elements and 0's where "A" has non-zero elements. For example and in the case of a scalar: ~1 is (FALSE) and ~0 is 1 (TRUE).

Note that for Matlab, any number (integer or real, positive or negative) different from 0 (zero) is a true statement, that is, it has the same meaning as with 1. Pay also attention that the precedence with relational and logical operators is the following (from highest to lowest):

1. Transpose ".'", power ".^", complex conjugate transpose "' ", matrix power "^";
2. Unary plus "+", unary minus "-", logical negation "~";
3. Multiplication ".\*", right division "./", left division that returns the inverse of a division ".\", matrix multiplication "\*", matrix right division "/", matrix left division "\";
4. Addition "+", subtraction "-";
5. Colon operator ":";
6. Less than "<", less than or equal to "<=", greater than ">", greater than or equal to ">=", equal to "==", not equal to "~=";
7. Element-wise logical AND "&";
8. Element-wise logical OR "|".

View the examples that follow to digest the use of logical and relational operators.

Matlab's command:

```
>> clear all; x=2; y=0; z=-2;  
>> a=(x>=1), b=(x~=1 & y), c=(1 | y & x>0), d=(~y & z | ~x == 5)
```

Matlab's response:

```
a =  
    1
```

```
b =  
    0
```

```
c =
```

```
    1
d =
    1
```

Comments:

Logical and relational operations. **a** is 1 because the statement evaluated is TRUE since **x** is 2 and greater than 1. **b** is FALSE because: “**x**~=1” returns 1, and “1 & **y**” returns 0. **c** is TRUE because: “**x**>0” is 1, and “1|**y** & 1” is “ 1 | 1” which is TRUE. **d** is TRUE because: “~**y**” is 1 and “~**x**” is 0 so we have “(1 & **z** | 0==5) that is equal to “(1 & **z** | 0) that is equal to: “1 | 0” that is TRUE.

Some examples with matrices and the logical and relational operators follow:

Matlab's command:

```
>> clear all; x=[5 0 -1; 2 4 8]; y=[2 1 -1; 1 2 4];
>> a=(x==5), b=(x~=1 & y), c=(2*y==x | ~x), d=(x>0 | y==1 | x~=x)
```

Matlab's response:

```
a=
    1    0    0
    0    0    0

b=
    1    1    1
    1    1    1

c=
    0    1    0
    1    1    1

d=
    1    1    0
    1    1    1
```

Comments:

Logical and relational operations with matrices. In **a** we have only the first element to be 1 since it the only one that is equal to 5. We get **b** to be a TRUE matrix because “**x**~=1” returns a matrix of 1’s since none element is equal to 1. By definition, all elements in **y** are taken to be TRUE statements. So TRUE & TRUE gives **b** full of 1’s (Figure out how the **c** and **d** were evaluated).



### 4.1.1 The any and all Functions

Matlab has two *build-in functions* that are work as logical operators. The first one is the *any* function that is TRUE if any element of a vector is nonzero. For vectors, *any* returns 1 if any of the elements of the vector are non-zero. Otherwise it returns 0. For matrices, *any* operates on the columns of the matrix, returning a row vector of 1's and 0's. The second one, *all*, becomes TRUE if all elements of a vector are nonzero. For vectors, *all* returns 1 if none of the elements of the vector are zero. Otherwise it returns 0. For matrices, *all* operates on the columns of the matrix, returning a row vector of 1's and 0's. View the following code to understand these tow function.

Matlab's command:

```
>> clear all; z=[-5 2 0]; x=[5 0 -1; 2 4 8]; y=[2 1 -1; 1 2 4];  
>> a1=any(z), a2=any(x), a3=any(y), a4=all(z), a5=all(x), a6=all(y)
```

Matlab's response:

```
a1 =  
    1  
  
a2 =  
    1    1    1  
  
a3 =  
    1    1    1  
  
a4 =  
    0  
  
a5 =  
    1    0    1  
  
a6 =  
    1    1    1
```

Comments:

Displaying the use of *any* and *all*.

## 4.2 Conditional Statements

The above section has been introduced since it is necessary for the evaluation of conditional statements as well as loop expressions (see the

following section). A conditional statement is a segment of programming code that evaluates a statement; if the statement is TRUE then it executes some commands, otherwise if it is FALSE it runs a bulk of different programming code. The two most important conditional statements that Matlab materialize, is the if statement and the switch statement.

### 4.2.1 The if Statement

The if statement evaluates a logical expression and executes a group of statements when the expression is TRUE. The optional elseif and else keywords can be used for the execution of alternate groups of statements. An end keyword, which matches the if terminates the last group of statements. The groups of statements are delineated by the four above keywords-no braces or brackets are involved [2]. The three possible syntax versions of this conditional expression are tabulated in Table 7.

**Table 7: The if statement syntax and examples**

Syntax	Example
<i>if logical expression</i> <i>segment of programming code executed if the logical expression is TRUE</i> <i>end</i>	<pre>flag=1; if isempty(flag)     disp('Hello') end</pre>
<i>if logical expression</i> <i>segment of programming' code executed if the logical expression is TRUE</i> <i>else</i> <i>segment of programming' code executed if the logical expression is FALSE</i> <i>end</i>	<pre>sales=5000;    if sales&lt;1000 Profit=sales*0.1; else     Profit=(sales-1000)*0.2+1000*0.1; end</pre>
<i>if logical expression #1</i> <i>segment of programming code executed if the logical expression #1 is TRUE</i> <i>elseif logical expression #2</i> <i>segment of programming code executed if the logical expression</i>	<pre>sales=5000; if (sales&gt;1000 &amp; sales&lt;=2000)     disp('Low Sales'); Profit=sale*0.1;</pre>

<pre>#2 is TRUE else     segment of programming code     executed if the previous logical     expressions are FALSE end</pre>	<pre>elseif (sales&gt;2000 &amp; sales&lt;=10000)     disp('Medium Sales ');     Profit=sales*0.15; else     disp('Satisfactory Sales ');     Profit=sales*0.17; end</pre>
---	--

Although it is possible to run these examples via the *command window*, it is better to use the *Matlab Editor/Debugger* to write a script instead. Real examples with the conditional statements as well as with the loops are postponed until the introduction of scripts in section 5.

## 4.2.2 The switch Statement

The **switch** statement executes groups of statements based on the value of a variable or variable or expression. The keywords **case** and **otherwise** delineate the groups. Only the groups. Only the first matching case is executed. There must always be an **end** to match the switch. [2]. The expression following the case should be either a scalar either a scalar or a string. The syntax of this conditional expression is tabulated in tabulated in

Table 8.

**Table 8: The switch statement syntax and example.**

Syntax	Example
<pre><b>switch</b> expression     <b>case</b> choice #1         segment of executable         programming code     <b>case</b> choice #2         segment of executable         programming code <b>otherwise</b></pre>	<pre>dice=3; <b>switch</b> (dice)     <b>case</b> 1         disp('One')     <b>case</b> 2         disp('Two')     <b>case</b> 3         disp('Three')     <b>case</b> 4</pre>

<i>segment of executable programming code</i> <b>end</b>	<pre> disp('Five') <b>case</b> 5     disp('Five') <b>otherwise</b>     disp('Six') <b>end</b> </pre>
---	--

## 4.3 Loop Expressions

As mentioned before, loop statements are useful when a segment of executable programming code is needed to be executed either a specific number of times, than or as long as an expression is TRUE. The for and while loop statements are needed commands when writing efficient programming code.

### 4.3.1 The for Loop

The for loop repeats a group of statements a fixed, predetermined number of times. A times. A matching end delineates the statements. The syntax for the for loop is exhibited in

Table 9.

**Table 9: The for statement syntax and example.**

Syntax	Example
<pre> <b>for</b> <i>index=frst_value : step: last_value</i>     <i>segment of executable programming code</i> <b>end</b> </pre>	<pre> <b>for</b> <i>i =1:10</i>     <i>disp(i)</i> <b>end</b>  <i>sumj=0;</i> <b>for</b> <i>j=25:-2:-12</i>     <i>sumj=sumj+j;</i> <b>end</b> </pre>

The colon notation is similar as in the case of the vectors. Actually, *index* in the for syntax is a vector with *n* elements with first element being the *first\_value* and last the *last\_value*. The difference between the *index* elements

is *step*. Afterwards, the for statement executes *n* times the segment of executable programming code moving from the first element of the *index* to the last one element-by-element at each time. If *step* is not displayed, then by default is set to 1.

### 4.3.2 The while Loop

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. That is, as long as an expression is TRUE, then the TRUE, then the segment of executable programming code that is included in the while the while statement is executed. A matching end delineates the statements [2]. The [2]. The syntax for the while loop is exhibited in

Table 10.

Table 10: The while statement syntax and example.

Syntax	Example
<b>while</b> <i>expression</i> <i>segment of executable programming code</i> <b>end</b>	<i>X=-3;</i> <b>while</b> <i>X&lt;=10</i> <i>disp(X)</i> <i>X=X+1;</i> <b>end</b>

### 4.3.3 Nested Conditional and Loop Expressions

All conditional and loop statements can be combined in any way that is desirable. If for example a loop expression includes an if statement that nests a switch statement and so on, then, in programming terminology, termed that we have *nested statements*. As long as the statements are nested in a logical and a non-redundant manner, the user will get answers. View the following example that is exhibited in

Table 11 and nests various statements (figure out their use).

**Table 11: Nested conditional and loop statements.**

An example of syntax of nested conditional and loop statements
<pre> clear; flag=1; x=2*pi; temp=pi; while x&lt;=4*pi     figure; hold on;     X=temp:pi/10:x;     for j=1:4         subplot(2,2,j); plot(X,sin(j*X)./X, 'b:', 'LineWidth', 3);     switch (j)         case 1             title('Plot of y=sin(x)/x');         case 2             title(' Plot of y=sin(2x)/x');         case 3             title(' Plot of y=sin(3x)/x');         otherwise             title(' Plot of y=sin(4x)/x');     end     xlabel('x'); ylabel('y'); end temp=x; x=x+pi; end </pre>

### 4.3.4 Additional Control Flow Statements

Matlab has additional *control flow* statements that can be used to write an executable *script* or *function*. These are more specialized statements that are listed in Table 12. You can learn more details via the online help.

**Table 12: Additional control flow statements.**

Statement	Comment
break	Terminates the execution of a for or while loop. Statements in the loop that appear after the break statement are not executed.
continue	Passes control to the next iteration of the for or while loop in which it appears, skipping any remaining statements in the body of the loop.

<p>try ... catch</p> <p>return</p>	<p>Changes <i>flow control</i> if an error is detected during execution. Causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.</p>
------------------------------------	--

## 5 m-files: Scripts and Functions

Up to this point, all commands passed to Matlab, were done through the *command window*. But when a segment of programming code is composed from several different commands, then it is time consuming, inefficient and quite difficult to continue using the *command window*. To alleviate this peculiarity, Matlab allows for the creation of *Function* and *scripts* by using an *Editor/Debugger* that looks like a text editor.

As said, *functions* and *scripts* can be created with the use of the Matlab's *Editor/Debugger*. You only have to write the command lines, save the file in a directory of your choice and execute it by typing its name to the *command window* and pressing **[Enter]** (you have to make sure that the Matlab's *current directory* and the directory in which you save the file are the same).

Although *m-files* represent a text file and it is possible to use any text editor to create a *function* or a *script* (as long as the extension of the file is \*.m this can work), it is preferable to use the default editor because: *i*) when you save a file it automatically takes the extension \*.m, *ii*) the reserved words like: for, while, function, end etc are blue coloured whilst comments that make the programming code more readable are green coloured, and *iii*) it automatically communicates with the *command window* and allows to run a *script* via a command found in the *editor tab*.

### 5.1 m-files: Functions

*Functions* are *m-files* that can accept input arguments and return output arguments. The name of the *m-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*. *Functions* are useful for extending the existing Matlab language for personal applications. For example a practitioner might want to write various *functions* that return the theoretical price of a call option contract according to various options pricing models. Options pricing models like the Black and Scholes, the



Merton's Jump's diffusion, the displaced diffusion model, Heston's stochastic volatility model, and other can easily be implemented via a different *function*.

A simple example of a function named as *DiagExtract.m* that takes as input a square matrix and returns in a vector the elements of its main diagonal is exhibited in Figure 9.

```

1 function [V]=DiagExtract(X)
2
3 % DiagExtract returns the diagonal elements of a square matrix (not single values).
4 % This function takes as input a square matrix named X and does the followings:
5 % 1. If by mistake the user does not enter a square matrix, it returns
6 % an error message
7 % 2. If a square matrix is correctly entered, then it returns the
8 % elements of its main diagonal in vector V
9
10 % Author: Panayiotis Andreou, Sept. 2003
11
12 % This function uses the error build-in function that displays a message
13 % and aborts the function's execution
14
15 [m,n]=size(X); %Saving the size of the matrix
16
17 if isempty(X) %Checking if it is an empty matrix
18     error('You have not passed a square matrix');
19 elseif (m~=n) %Checking if matrix is square
20     error('You have enter a non square matrix');
21 elseif (m==n && m==1) %Checking if it is a single value
22     error('You have enter a single value');
23 else
24     Ind=(1:n:n^2)+(0:n-1); %Finding the indeces of the diagonal elements
25     V=X(Ind); %Extracting to V the diagonal elements
26 end
  
```

**Figure 9: An example of a Function**

For the function *DiagExtract* note the followings: *i)* if after executing your *function* you get an error, then read the error message that it is returned to the *command window* (this is very helpful in finding where to look for the error); you have to return to the *Editor* to *debug* the error, re-save the *function* and try to run it again. If you *debug* your *function* or add something, then you have to re-save the file in order to take place the changes (note that if unsaved changes are made in a function file, an asterisk (\*) appears next to the function name in the *Editor's* title bar; *ii)* although the input argument of the *function* is a matrix named X and the output is a vector named V,

when you call this *function* from a *script* or from the *command window*, you can give any names you like. For example in the *command window* enter:

```
“[DiagElements]=DiagExtract(magic(4))”
```

to see the result; *iii*) After the *function* is executed, none of the variables that were used are saved in the *workspace*; *iv*) If after each command line you do not place the semi-colon “;” then when the *function* is executed, you can observe in the *command window* the result of these command lines (this is useful to be done when you try to debug a *function*); *v*) observe on Figure 9 that on the bottom of the *Editor/Debugger* there is a predefined caption area that indicates that the current *m-file* is a *function* with the name `DiagExtract`.

### 5.1.1 Basic Parts of a Function

A *function* is composed from the following parts (most of this section is taken/copied from [2])

#### 5.1.1.1 Function Definition Line

The *function* definition line informs Matlab that the *m-file* contains a *function*, and specifies the argument calling sequence of the *function*. For example, the function definition line for the `average` function (that computes the average of a vector) could be:

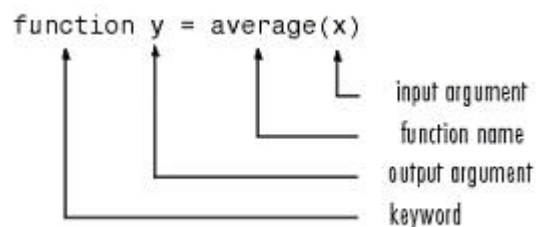


Figure 10: Function Definition Line

Matlab *function names* have the same constraints as *variable names*. The name must begin with a letter, which may be followed by any combination of

letters, digits, and underscores. Making all letters in the name lowercase is recommended as it makes your *m-files* portable between platforms (of course the use of upper case assures that the function names you create do not resemble any other Matlab *function* and/or reserved word). The name of the text file that contains a Matlab *function* consists of the *function name* with the extension *.m* appended. For example, “DiagExtract.m”.

If the *filename* and the *function definition line name* are different, the internal (function) name is ignored. Thus, if “DiagExtract.m” is the file that defines a *function* named: `diagonal_extraction`, you would invoke the function by typing in the *command window*:

```
“DiagExtract”
```

Concerning *function arguments*, if the function has multiple output values, enclose the output argument list in square brackets “[ ]”. Input arguments, if present, are enclosed in parentheses “( )”. Use commas to separate multiple inputs or output arguments. Here's a more complicated example:

```
“function [Abs, Mean, Std] = StatistiS(X, Y, Z)”
```

If there is no output, leave the output blank:

```
“function print_results(x)”
```

or use empty square brackets:

```
“function [] = print_results (x)”
```

The variables that you pass to the *function* do not need to have the same name as those in the *function definition line*.

### **5.1.1.2 The H1 Line**

The *H1 line*, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the *H1 line* begins with a percent sign, “%.” For the `DiagExtract`, the *H1 line* is:

```
“% DiagExtract returns the diagonal elements of a square matrix (not single values).”
```

This is the first line of text that appears when a user types `help function_name` at the Matlab prompt. Further, the `lookfor` searches and displays only the *H1 line* of the function. Because this line provides important summary information about the *m-file*, it is important to make it as descriptive as possible.

### 5.1.1.3 The Help Text

You can create online help for your *m-files* by entering text on one or more comment lines beginning with the line immediately following the *H1 line*. The help text for the `DiagExtract` function is:

```
"% This function takes as input a square matrix named X and does the followings:
%     1. If by mistake the user does not enter a square matrix, it returns
% an error message
%     2. If a square matrix is correctly entered, then it returns the
% elements of its main diagonal in vector V "
```

When you type `help function_name`, Matlab displays the comment lines that appear between the *function definition line* and the first non-comment (executable or blank) line. The help system ignores any comment lines that appear after this help block.

### 5.1.1.4 The Body Text

The *function body* contains all the Matlab code that performs computations and assigns values to output arguments. The statements in the *function body* can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the `DiagExtract` function contains a number of simple programming statements:

```
"% Author: Panayiotis Andreou, Sept. 2003
```

```

% This function uses the error build-in function that displays a message
% and aborts the function's execution

[m, n]=size(X); %Saving the size of the matrix

if isempty(X) %Checking if it is an empty matrix
    error('You have not passed a square matrix');
elseif (m~=n) %Checking if matrix is square
    error('You have enter a non square matrix');
elseif (m==n & m==1) %Checking if it is a single value
    error('You have enter a single value');
else
    Ind=(1:n:n^2)+(0:n-1); %Finding the indices of the diagonal elements
    V=X(Ind); %Extracting to V the diagonal elements
end ”

```

## 5.2 Scripts

On the other hand, *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 *function*. *Scripts* are useful for automating a series of steps that are needed to be performed many times (e.g. to create a *script* that calls various options pricing *function* in order to price a call option and compare the pricing accuracy of each model).

Unlike *function*, a *script* has no a specific structure. It includes a number of commands that are serially executed. As long as the command series has a logical interpretation, the *script* will result to the desire output. Remember that a *script* does not take and does not return input and output arguments respectively. The vectors and matrices (variables or scalars) are stored in the Matlab's *workspace*. An example of a script follows.

Let's say that we need to create a *script* with the name "CompuStat.m" that

should store in a vector named `Data`, the mean, standard deviation, median, covariance, minimum and maximum of the main diagonal of a magic square. Figure 11, shows how a version of such a *script* might look like for an 8-by-8 magic square.

```

1  % This script returns various descriptive statistics for the elements of
2  % the main diagonal of a magic square
3
4  clear all %You should always clear the workspace before operating on it
5  clc %This is optional if you want to clear the screen and home the cursor
6
7  Xmatrix=magic(8); %Creating the magic square
8
9  diagEle=DiagExtract(Xmatrix); %Calling the appropriate function
10 % Storing the desired statistics in the Data vector
11 Data=[mean(diagEle), std(diagEle), median(diagEle),...
12       cov(diagEle), min(diagEle), max(diagEle)];
13
14 % Printing the results in the screen
15 disp('The diagonal of the matrix is:')
16 disp(diagEle)
17 fprintf('Mean is: %g, standard deviation is: %g, median is: %g, \n',Data(1:3))
18 fprintf('covariance is: %g, minimum is: %g, and maximum is: %g.',Data(4:6))
19
20 %Plotting the diagonal elements
21 plot(diagEle,'mh','markersize',10);
22 xlabel('Obs'); ylabel('Value'); title('Plot of diagonal elements');
23
24 % To run this script go: Debug>Save and Run. Otherwise, save it to the desired
25 % directory, go to command window and type its name at ">>" (make sure that yours
26 % and Matlab's directory are the same)

```

**Figure 11: An example of a script**

Try to see what this *script* does. After executing the *script*, figure out what is the difference between the command `disp` and the `fprintf`. Use the command `who` to view the *script*'s variables that are saved in the workspace. Notice that a *script* does not have the same structure as a *function*. Moreover, observe on Figure 11 that on the bottom of the *Editor/Debugger* there is a predefined caption area that indicates that the current *m-file* is a *script*.

Important Note: when you start writing *scripts* and *function* that are stored in a certain directory, you occasionally need to see what files are included in this directory. To do so, from the *command window* you can use the `dir` command (the same one as used in the MS-DOS) to view the contents of Matlab's *current directory* or what to view all *m-files* in the directory. Additionally, you can use `cd directory_name` to set the current directory to

the one specified and `cd..` to move to the directory above the current one. Moreover, `delete file_name` deletes the specified file, and `pwd` prints on the screen the *current directory*.

## 5.3 Testing and Debugging

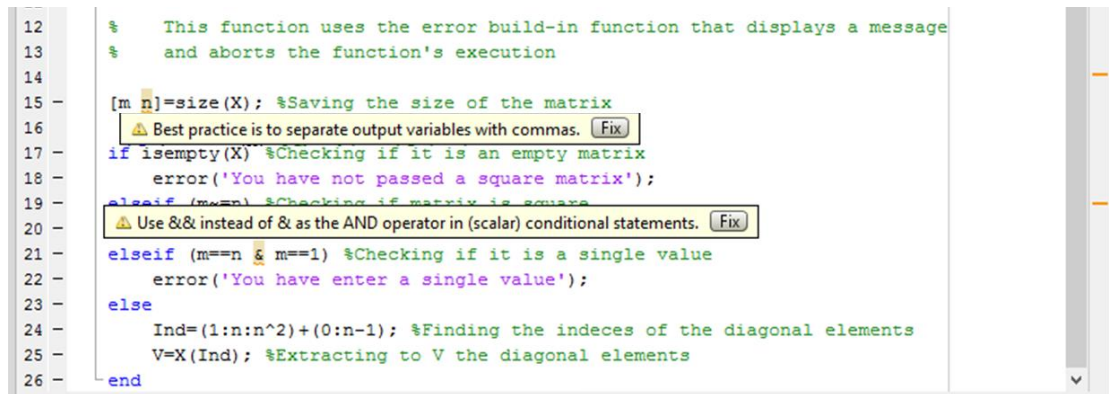
Matlab provides several tools for testing and debugging programs (either *functions* or *scripts*) to ensure they produced the expected results. A (non-trivial) program hardly ever works correctly for all possible cases straightaway. It requires thorough testing and debugging.

Potential errors or pieces of code that could cause unwanted effects (warning) can be identified by using *Matlab Editor*. If the code in the Matlab editor looks different from what you expect there is probably a syntax error. E.g.:

- A string (text) must be displayed in purple. If it is red, you probably forgot the second ‘ to end the text.
- When key words like `while`, `for` or `if` are underlined in red, the corresponding end is missing.
- When the indentation looks weird, first try to mark the piece of code and on Editor tab, in the Edit section, click on smart indent (or press `[ctrl]+[i]`). If this does not help, look for a forgotten end.

*Matlab Editor* underlines suspicious commands in red for errors and in orange for warnings. When you move the mouse pointer over the mark, a pop-up box tells you about the potential problem. When you click into the box, you will get more information, and frequently a suggested replacement for the particular code as shown in Figure 12. (Note that not always the suggested replacement is the answer for correcting the error or warning.) In addition, Matlab considers it to be good style to suppress all command line outputs from within functions and scripts by using “;” and will mark all lines without semicolon. If you want to have the command line output, just ignore the red line or click with a right click on the marked “=” and choose “suppress this instance” from the pop-up menu. Finally when writing a

*function*, Matlab marks variables that do not contribute to the calculations of the output(s) of the function i.e. are defined but not further used in the *function*. Consider if you really need them or if you can get rid of the command to keep your program as simple and clear as possible and to avoid using more memory capacity than necessary.



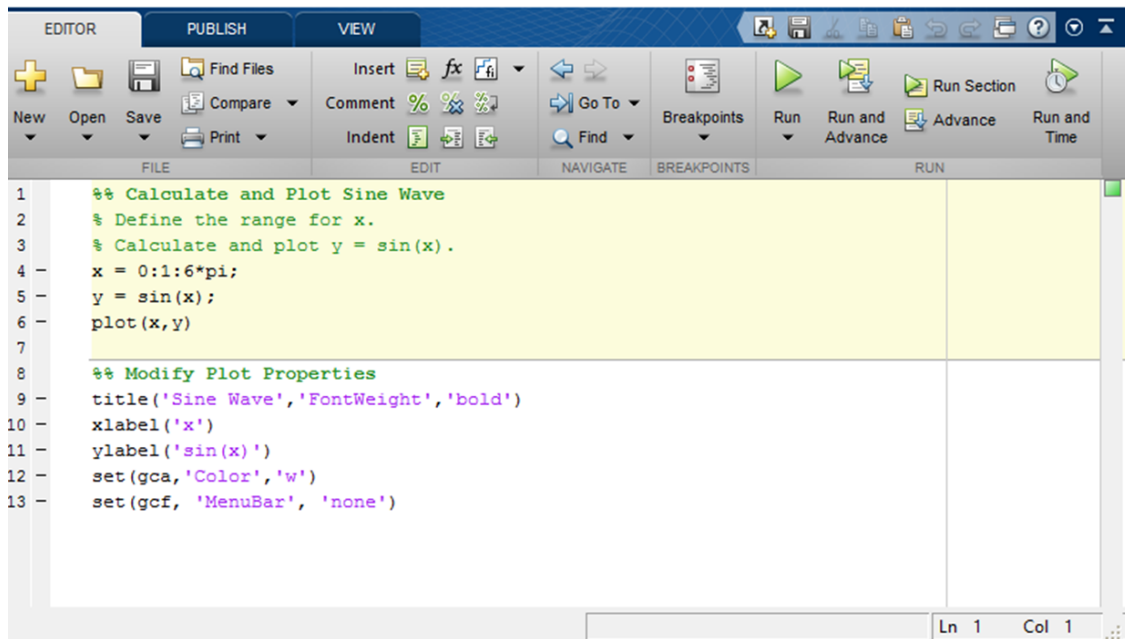
```
12 % This function uses the error build-in function that displays a message
13 % and aborts the function's execution
14
15 [m n]=size(X); %Saving the size of the matrix
16 % Best practice is to separate output variables with commas. [Fix]
17 if isempty(X) %Checking if it is an empty matrix
18     error('You have not passed a square matrix');
19 elseif (m==n) %Checking if matrix is square
20     % Use && instead of & as the AND operator in (scalar) conditional statements. [Fix]
21 elseif (m==n & m==1) %Checking if it is a single value
22     error('You have enter a single value');
23 else
24     Ind=(1:n:n^2)+(0:n-1); %Finding the indeces of the diagonal elements
25     V=X(Ind); %Extracting to V the diagonal elements
26 end
```

**Figure 12: Warnings indication**

### 5.3.1 Code Sections

When testing a program it usually helps to look only at a sub-problem at a time. The *Matlab editor* provides a convenient way to do so, using *code sections* also known as *code cells* or *cell mode*. A code section contains contiguous lines of code that you want to evaluate as a group. Sections are marked by two percent signs (“%%”) in the line before the program lines start. “%%” can be followed by comments. These comments are displayed in bold, a very convenient way to have titles for programming blocks. Figure 13 shows a script named *sine\_wave* that uses the code sections. You can run individual sections without running the rest of the code by clicking on the section (the current section is marked in yellow) and on the Editor tab, in the Run section, click *Run Section*. You can also run the code in the current section, and then move to the next section by clicking on Run and Advance.

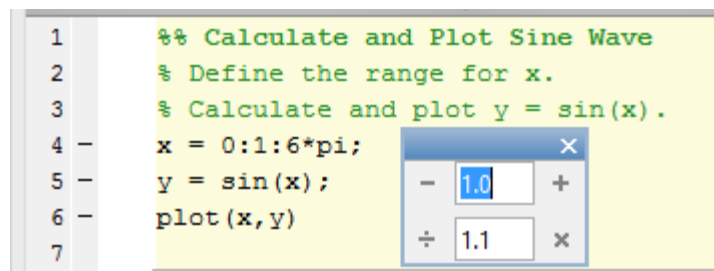




**Figure 13: Script with code sections**

Finally, you can increment numbers within a section, rerunning that section after every change. This helps you fine-tune and experiment with your code. To increment or decrement a number in a section:

- Highlight or place your cursor next to the number.
- Right-click to open the context menu.
- Select Increment Value and Run Section. A small dialog box appears as shown in Figure 14.



**Figure 14: Increment Value dialog box**

- Input appropriate values in the  $+$  /  $-$  text box or the  $\times$  /  $\div$  text box.
- Click the  $+$ ,  $-$ ,  $\times$ , or  $\div$  button to add to, subtract from, multiply, or divide the selected number in your section. Matlab runs the section after every click.

### 5.3.2 Debugging

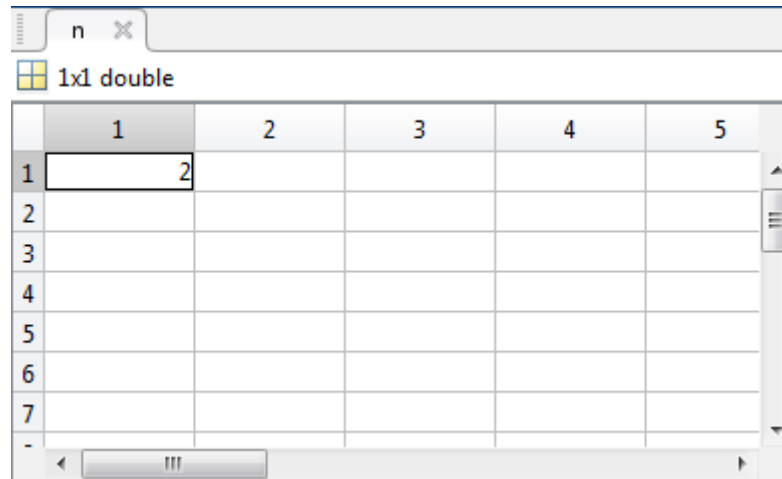
If a program does not produce the expected results, you will have to find out if the error is in the algorithm or in the implementation. Therefore, you need to follow the program step by step and compare the results of the program with your expectations. To assist you in this procedure, in addition to the section code described above, *Matlab editor* also provides the *debugger*.

In order to pause the execution of the m-file, so you can examine values where you think the problem can be, you need to set a *breakpoint*. *Breakpoints* are introduced by clicking on the “-“ between the line number and the desired line in the Matlab editor as shown in Figure 7. A red dot appears, showing that a breakpoint is set. If you make changes to the file or there is a syntax error in the file, the dot will turn grey until you correct the syntax errors, if any, and save the file. Note that you can only set valid breakpoints at executable lines in saved files that are in the current folder or in folders on the search path. When you add a breakpoint in a file that is not in a folder on the search path or in the current folder, a dialog box appears. This dialog box presents options that allow you to add or remove the breakpoint. You can either change the current folder to the folder containing the file, or you can add the folder containing the file to the search path.

After setting breakpoints, you can run the file from the *command window* or the *Matlab editor*. If you are working on a *function* with input arguments, you need to start it from the *command window* (otherwise Matlab would not know the input arguments). If no input arguments are needed, you can also click on Run in the Run Section of the Editor tab. Matlab will execute the commands until the first breakpoint is reached and stop there. In the *command window* you will see K>> to indicate that Matlab is in debugging mode. The current m-file line is marked with a green arrow in the editor.

While the program is paused, you can view the list of current variables in the workspace window (if you need to change to another workspace and view its variables, use Function Call Stack drop-down list on the Editor tab, in the Debug section). Examine values when you want to see whether a line of code has produced the expected result or not. You can access these variables by:

- Double clicking in the workspace window to open the array editor Figure 15. The *Variables Editor* opens, displaying the content for that variable. You can even change the value in the *Variables Editor*, but you should usually restrain from doing so if you want to do your operations reproducibly.




**Figure 15: Variables Editor**

- Typing the name of the variable in the command window and you will get the value displayed as text message.
- Placing the tip of your mouse pointer on a variable name in the editor window, a small window will pop up, showing type and value of the variable. (This of course only works for variables introduced in program lines, which were already processed.)

If the result is not as you expect, then that line, or a previous line, contains an error make the necessary changes, save and rerun the program. If the result is as expected, continue running or step to the next line. To continue, you have the following options, in the Debug section on the Editor tap:

- Continue (🟢): continue the program until the next breakpoint or the end of the file is reached.
- Step (📄): process one line of program code and stop again. The displayed variables will change their values according to the operations performed.
- Step in (📄): step into (sub-) function or (sub-) script (if the next line is no call of a function or script, it does the same as step).
- Step out (📄): go back from a (sub-) function or script to the program

from where it was called.

- Exit debug mode (

To delete a breakpoint click again on the red (or grey) dot. If you want to clear all breakpoints, on the Editor tab, in the Breakpoints section, click on breakpoints and then clear all.

Finally you can set *error breakpoints* to stop program execution and enter debug mode when Matlab encounters a problem. Unlike standard breakpoints, you do not set these breakpoints at a specific line in a specific file. Rather, once set, Matlab stops at any line in any file when the error condition specified by using the *error breakpoint* occurs. Matlab then enters debug mode and opens the file containing the error, with the pause indicator at the line containing the error. To introduced such *error breakpoints*, in the Editor tab, in the Breakpoints section, click on breakpoints and then select *Stop on Errors* to stop on all errors and/or *Stop on Warnings* to stop on all warnings.

## 6 Cells and Structures

*Structures* are collections of different kinds of data organized by named fields. *Cell arrays* are a special class of Matlab arrays whose elements consist of cells that themselves contain Matlab arrays. Both *structures* and *cell arrays* provide a hierarchical storage mechanism for dissimilar kinds of data. They differ from each other primarily in the way they organize data. You access data in *structures* using named fields, while in *cell arrays* data is accessed through matrix indexing operations [2].

### 6.1 Cells

A *cell array* is a Matlab array for which the elements are *cells*, containers that can hold other Matlab arrays. For example, one cell of a cell array might contain a real matrix, another an array of text strings, and another a vector of complex values [2]. Moreover, you can build *cell arrays* of any valid size or shape, including multidimensional *structure arrays*. Figure 16 gives an illustration on how a 2-by-2 cell array might look.

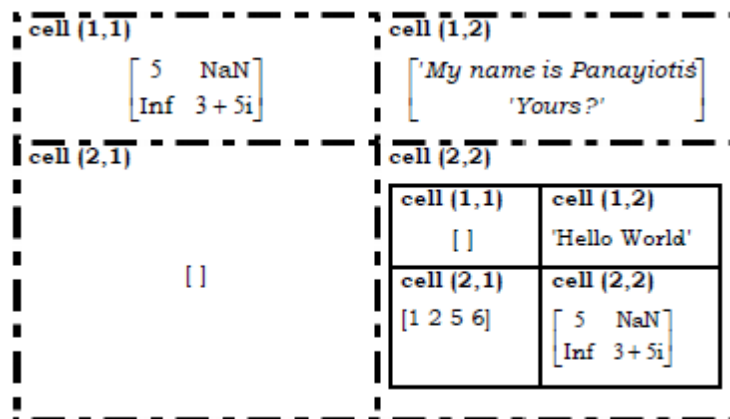


Figure 16: A 2-by-2 cell array

To create a *cell array* use the command: `cell`. For example if we want A to be a 3-by-3 *cell array*, then you should use the following syntax:

```
"A=cell(3,3)"
```

to get:

```

“A =
    []    []    []
    []    []    []
    []    []    [] “

```

That is, the A has been created to be a *3-by-3 empty cell array*. The *cells* identification is the same as with matrices. The upper left *cell* location/position is (1,1), the next *cell* in the same row is (1,2) and so on (single value indexing is allowed also). You can fill a *cell array* by assigning data to individual *cells* one at a time always supporting the correct *cell* location/index. Matlab has two default ways to store data (scalars, vectors, matrices, strings, cells, etc) to cells:

### 6.1.1 Cell indexing

Enclose the *cell* subscripts in parentheses using standard array notation. Enclose the *cell* contents on the right side of the assignment statement in curly braces, "{}" [2]. For example:

```

“A(1,1) = {[1 4 3; 0 5 8; 7 2 9]}; A(1,2) = {'Anne Smith'};
A(2,1) = {3+7i}; A(2,2) = {-pi:pi/10:pi}; A(2,3)={5};
G=cell(2,1); G(1,1)= {[5 6 9]}; G(2,1)={'Hello World'};
A(3,1) = {G}; A(3,3)={ones(2,2)}; ”

```

After assigning the cells, in the *command window* type: “A” to get:

```

“A =
    [3x3 double]    'Anne Smith'    []
    [3.0000+ 7.0000i]  [1x21 double]    [    5]
    {2x1 cell }      []    [2x2 double] ”

```

## 6.1.2 Content indexing

Enclose the *cell* subscripts in curly braces using standard array notation. Specify the *cell* contents on the right side of the assignment statement [2]. For example:

```
"A{1,1} = [1 4 3; 0 5 8; 7 2 9]; A{1,2} = 'Anne Smith';  
A{2,1} = 3+7i; A{2,2} = -pi:pi/10:pi; A{2,3}=5;  
G=cell(2,1); G{1,1}= [5 6 9]; G{2,1}='Hello World';  
A{3,1} = G; A{3,3}=ones(2,2); "
```

produces the same *cell array* as before.

If you assign data to a *cell* that is outside the dimensions of the current array, Matlab automatically expands the array to include the subscripts you specify. It fills any intervening *cells* with empty matrices. For example, the assignment below turns the 3-by-3 cell array *A* into a 3-by-4 cell array [2] (note that the same holds for vectors and matrices with the only difference that empty elements are assigned with zero values).

```
"A{3,4}='This expands the cell array';" to get:
```

```
"A =  
      [3x3 double]      'Anne Smith'      []  
      [3.0000+ 7.0000i]  [1x21 double]  [ 5]  
      {2x1 cell }      []      [2x2 double] "
```

The only difference between the *cell* command and the *cell* and *content indexing* is that by using *cell* you pre-allocating the *cell's* dimensions.

You can use content indexing on the right side of an assignment to access some or all of the data in a single *cell*. Specify the variable to receive the *cell* contents on the left side of the assignment. Enclose the *cell* index expression on the right side of the assignment in curly braces. This indicates that you are assigning cell contents, not the *cells* themselves. For example, to store to

B the 3-by-3 matrix that is located in A(1,1), you should do the following:

```
"B=cell{1,1};"
```

to get:

```
"B =  
    1    4    3  
    0    5    8  
    7    2    9"
```

and to save in C the 2<sup>nd</sup> and 3<sup>rd</sup> elements of the vector that is saved in the (1,1) cell of G which is saved in the (3,1) cell of A you do the following:

```
"C= A{3,1}{1,1}(2:3)"
```

to get:

```
"C =  
    6    9"
```

If you like to save in D the cells (1,2) to (2,3) of A (that is to create a 2-by-2 cell array) you do the following:

```
"D=A(1:2,2:3)"
```

to get:

```
"D =  
    'Anne Smith' []  
    [1x21 double] [5]"
```

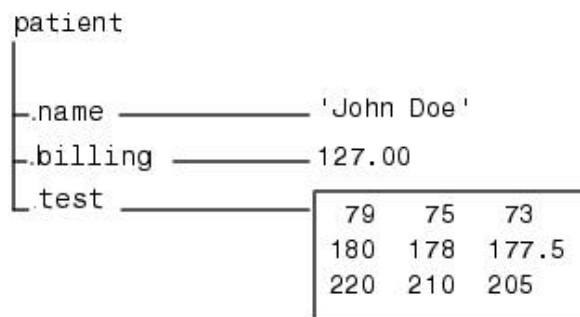
So, if "{" are used after a *cell array* then Matlab returns the array's contents but if "(" are used instead, then it returns *cells*. To access sub-sets of information stored in a *cell* use first "{" to access the *cell* and after used "(" to access the info you want.



Before leaving this section, notice the existence of two very useful *cellarrays Functions*. These are: `celldisp` that recursively displays the contents of a *cell array*, and `cellplot` displays the structure of a *cell array* as nested coloured boxes. Further info for *cell arrays* can be obtained from the Matlab's online help facilities.

## 6.2 Structures

*Structures* are Matlab arrays with named *data container*' called *fields*. The *fields* of a structure can contain any kind of data. For example, one *field* might contain a text string representing a name, another might contain a scalar representing a billing amount, a third might hold a matrix of medical test results, and so on. Like standard arrays, *structures* are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build *structure arrays* with any valid size or shape, including multidimensional structure arrays (see figure below) [2].



**Figure 17: Example of structure (Figure copied from [2])**

Structures can be created either by using assignment statements or by using the `struct` function.

### 6.2.1 Building structure arrays using assignment statements

You can build a simple 1-by-1 *structure array* by assigning data to individual fields. Matlab automatically builds the *structure* as you go along. For example, create the 1-by-1 *patient structure array* shown at the beginning of this section [2]:

```
“patient.name = 'John Doe';  
patient.billing = 127.00;  
patient.test = [79 75 73; 180 178 177.5; 220 210 205]; “
```

and if you type: “patient” in the *command window* you get:

```
“patient =  
    name: 'John Doe'  billing: 127  
    test: [3x3 double] ”
```

that is an array containing a *structure* with three *fields*. To expand the *structure array*, add subscripts after the *structure name*:

```
“patient(2).name = 'Ann Lane'; patient(2).billing = 28.50;
```

```
patient(2).test = [68 70 68; 118 118 119; 172 170 169];” and if you
```

again type: “patient” you will get:

```
“patient =  
    1x2 struct array with fields: name  
        billing  
        test ”
```

The patient *structure array* now has size [1 2]. Note that once a *structure array* contains more than a single element, Matlab does not display individual *field* contents when you type the array name. Instead, it shows a summary of the kind of information the *structure* contains.

You can also use the `fieldnames` function to obtain this information. `fieldnames` returns a cell array of strings containing field names (see the online help).

As you expand the *structure*, Matlab fills in unspecified *fields* with empty matrices so that:

- All *structures* in the array have the same number of *fields*
- All *fields* have the same *field names*

For example, entering `patient(3).name = 'Alan Johnson'` expands the patient array to size [1 3]. Now both `patient(3).billing` and `patient(3).test` contain empty matrices. To see this, type: `patient(3)` to get:

```
“ans =  
  
    name: 'Alan Johnson' billing: []  
  
    test: [] ”
```

## 6.2.2 Building structure arrays using the struct function

You can pre-allocate an array of *structures* with the `struct` function. Its basic form is [2]:

```
“str_array = struct (field1', val1, 'field2',val2, ...)”
```

where the arguments are *field names* and their corresponding values. A *field value* can be a single value, represented by any Matlab data construct, or a *cell array* of values. All *field values* in the argument list must be of the same scale (single value or *cell array*). To create the *structure* with the patient info shown before you should have the following syntax (the first command line instructs for the creation of a *1-by-3 structure* will all *fields* set to empty):

```
“patient(3)= struct ('name',[],'billing',[],'test',[]);  
patient(1)=struct('name', 'John Doe', 'billing', 127.00, 'test', ...  
                [79 75 73; 180 178 177.5; 220 210 205]);  
patient(2)=struct('name', 'Ann Lane', 'billing', 28.50, 'test', ...  
                [68 70 68; 118 118 119; 172 170 169]);  
patient(3)=struct('name', 'Alan Johnson', 'billing', [], 'test', []);”
```

The view an entire *field* this can be done with the following example syntax:

```
“structure_name.field_name”
```

For example type: `patient.name` in the *command window* to get:

```
“patient.name
```

```
ans =  
John Doe
```

```
ans =  
Ann Lane
```

```
ans =  
Alan Johnson ”
```

To view the *fields data* associated with a certain *structure entry*, the calling syntax should be of the form:

```
“structure_name(index_of_entry)”
```

For example type: “patient(2)” in the *command window* to get:

```
“ans =  
name: 'Ann Lane' billing: 28.5000 test: [3x3 double] ”
```

To manipulate sub-data on data stored under a *fields name* for a specific *entry*, the calling syntax should be of the form:

```
“structure_name(index_of_entry).field_name( ) ”
```

For example type: “patient(2).test(2:end,1:end-1)” in the *command window* to get:

```
“ans =  
118 118  
172 170 ”
```

Useful *build-infunction* that can make the *structures* handling much easier include:

- `fieldnames` to get *structure field names*
- `getfield` that can be used to get *structures' fields*
- `isfield` that returns a TRUE if *field* is in *structure array*

- `rmfield` that can be used to delete a *structure's field*
- `setfield` that can be used to assign a *structure field*

*Structures* and *cell arrays* are very useful when you need to write a program that handles various *data types*. It is a challenge to work with such *data types*. Before attempting writing a segment of programming code based on these make sure that you have spent a lot of time reading the Matlab's online help.

## 7 Entering and Saving Data Files

Matlab can be used to both load a file of data from an external source and to save to your PC data-sets that are located in the *workspace* or that are created from a *function*. When you have to deal with small amount of data, then you can enter them by hand either in the *command window* or in the *Editor/Debugger*. But when you have to work with a large amount of data retrieved by an external source (e.g. you need to process the historical daily prices for S&P 500 from 1985 to 1999, saved in a file that was downloaded from the Internet), then it is an imperative need to import those data either invoking a graphical user interface i.e. interactively or using some *build-in function* that can read files of data.

### 7.1 Import Data Interactively

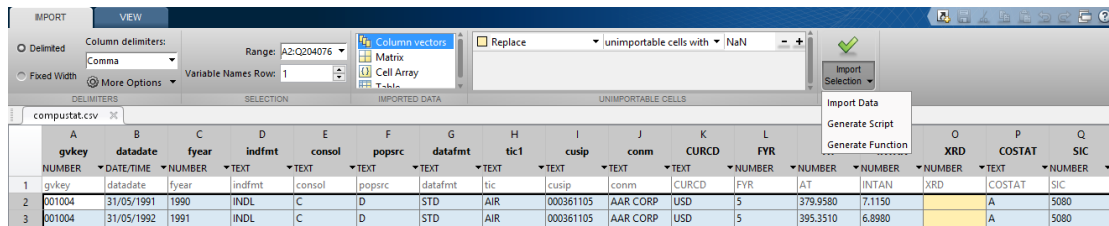
To retrieve data form an external source interactively you can either double-click a file name in the *Current Folder browser* or on the *Home tab*, in the *Variable section*, select *Import Data*. The imported file can be a text file, a data file or Excel file. Once the file is open the import tool as shown in Figure 18 will appear providing a view to the data in the file. Note that for large files only a beginning subset of the data is loaded in the preview, therefore you can create a custom import to load in part of the large data file without having to use the memory required to load in the entire data file. The initial setting for formatting, variables names and type of the imported data are educated choices by the tool based on file structure and type. However all of the choices can be modified by the user.

NUMBER	gkey	datadate	year	indfmt	consol	popsrc	datafmt	tic	cusip	comm	CURCD	FYR	AT	INTAN	XRD	COSTAT	SIC
2	001004	31/05/1991	1990	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	379.9580	7.1150		A	5080
3	001004	31/05/1992	1991	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	395.3510	6.8980		A	5080
4	001004	31/05/1993	1992	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	365.1510	6.5710		A	5080
5	001004	31/05/1994	1993	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	417.6260	6.3130		A	5080
6	001004	31/05/1995	1994	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	425.8140	6.1010		A	5080
7	001004	31/05/1996	1995	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	437.8460	5.8420		A	5080
8	001004	31/05/1997	1996	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	529.5840	5.6530		A	5080
9	001004	31/05/1998	1997	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	670.5590	26.5650		A	5080
10	001004	31/05/1999	1998	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	726.6300	40.0930		A	5080
11	001004	31/05/2000	1999	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	740.9980	38.8400		A	5080
12	001004	31/05/2001	2000	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	701.8540	45.3750		A	5080
13	001004	31/05/2002	2001	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	710.1990	50.0420		A	5080
14	001004	31/05/2003	2002	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	686.6210	49.9240		A	5080
15	001004	31/05/2004	2003	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	709.2920	47.7700		A	5080
16	001004	31/05/2005	2004	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	732.2300	47.2530		A	5080
17	001004	31/05/2006	2005	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	978.8190	46.7890		A	5080
18	001004	31/05/2007	2006	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	1067.6330	76.1380		A	5080
19	001004	31/05/2008	2007	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	1362.0100	131.1660		A	5080
20	001004	31/05/2009	2008	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	1377.5110	150.9950		A	5080
21	001004	31/05/2010	2009	INCL	C	D	STD	AIR	000361105	AAR CORP	USD	5	1501.0420	169.4280		A	5080
22	001009	31/10/1991	1991	INCL	C	D	STD	ABSI	000781104	ABS INDUS...	USD	10	35.5590	0.0000		I	3460
23	001009	31/10/1992	1992	INCL	C	D	STD	ABSI	000781104	ABS INDUS...	USD	10	41.9760	0.0000		I	3460
24	001009	31/10/1993	1993	INCL	C	D	STD	ABSI	000781104	ABS INDUS...	USD	10	63.9970	0.0000		I	3460
25	001009	31/10/1994	1994	INCL	C	D	STD	ABSI	000781104	ABS INDUS...	USD	10	93.8110	0.0000		I	3460
26	001010	31/12/1991	1991	INCL	C	D	STD	4165A	00099V004	ACF INDUS...	USD	12	1756.2620	5.0000	0.9000	I	3743
27	001010	31/12/1992	1992	INCL	C	D	STD	4165A	00099V004	ACF INDUS...	USD	12	1706.4540		0.8000	I	3743

Figure 18: The import tool

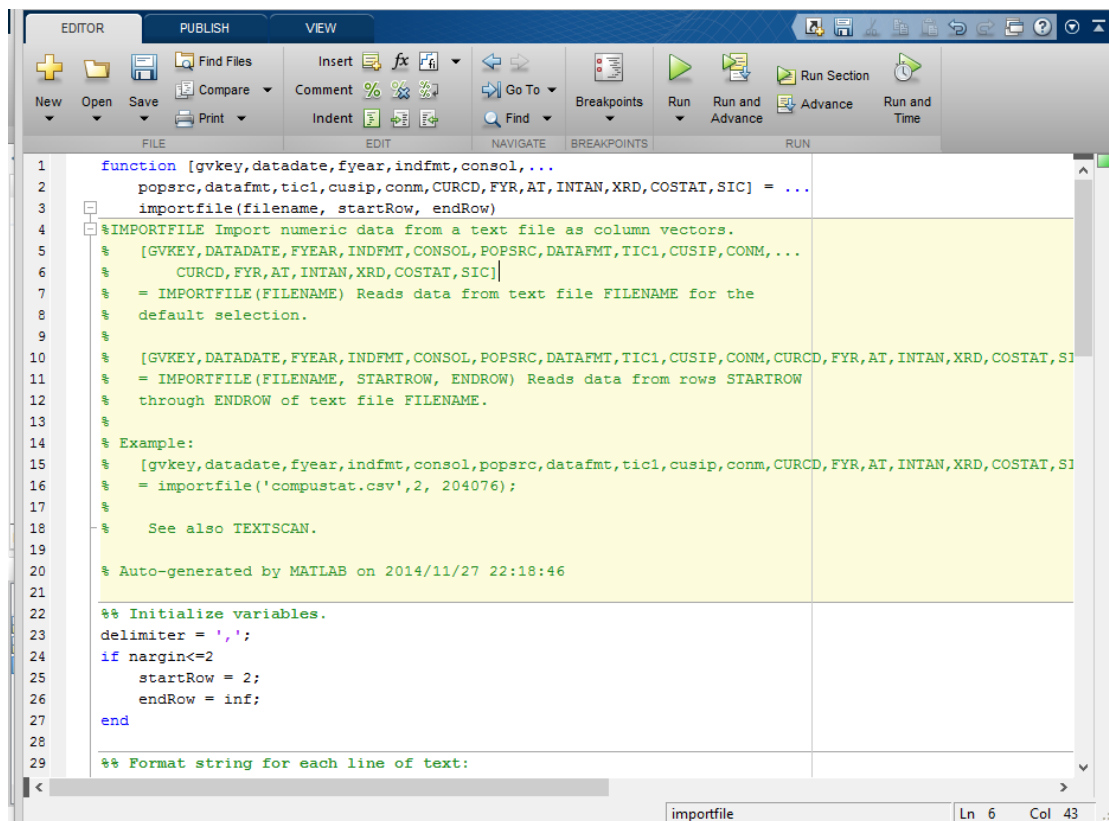
In the example shown in Figure 18 the variable in column B is date but it is read as text. This can be modified by clicking on the corresponding Type and select the correct one. Then if you hover over a cell you can see how the data will be imported. In the case of column I (named Cusip) since the majority of the data are number, the tool chooses as a type for this column number. However this is wrong since some of the entries have characters in them. You can alter the type to text by clicking on the type on column I. Similarly, the column O (named XRD) the majority of data are missing thus the tool chooses the default type that is text. Empty cells are automatically replaced by a NaN value, but this is also adjustable in the *Unimportable Cells* section.

By default the data will be import as a column vectors however you could choose to bring them in either as a numeric matrix or cell array. Moreover you can adjust the columns and rows of the data you wish to bring in by either highlighting the section or by entering the range in the corresponding text box.



**Figure 19: Importing choices**

Once you have established the way you wish to import the data, you either import the data directly into Matlab workspace, by selecting Import Data, in the Import section on the Import tab, or you can generate a script or a function by selecting the corresponding option in the same menu (Figure 19). By generating the code you can make it easier to repeat the data import process. If the generate function option is used, then the function in Figure 20 is automatically produced. Therefore, saving this function can be used to import the same or similar data.



**Figure 20: Automatically generated function for importing data**



## 7.2 Import Data using build-in functions


Matlab provides a variety of functions that can be used to read data files. Using this approach you could read more complex and non-standard format files, have more control over importing options and automate the importing of one for many files.

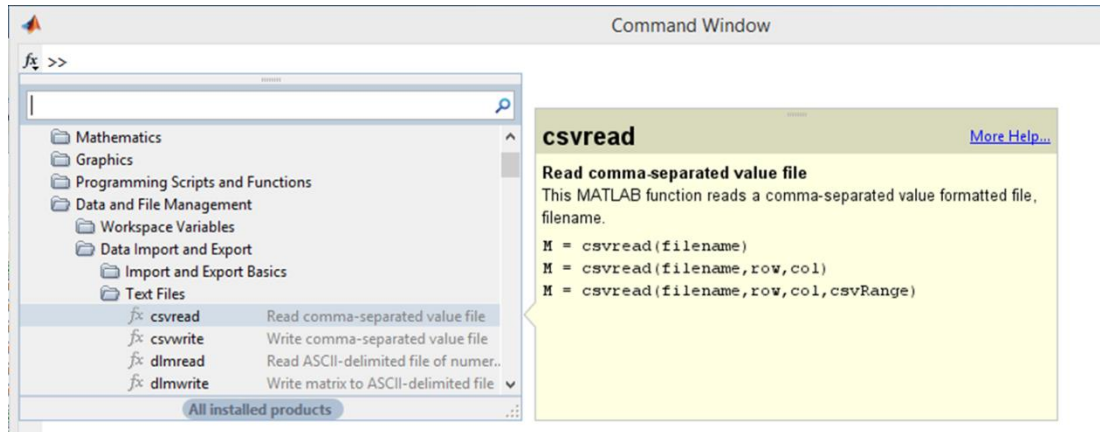
The Table 13 shows the supported file types and the associated function to read them.

**Table 13: File formats that Matlab can handle**

File Content	Extension	Description	Import Function	Export Function
MATLAB formatted data	MAT	Saved MATLAB workspace	load	save
		Partial access of variables in MATLAB workspace	matfile	matfile
		Comma delimited numbers	csvread	csvwrite
		Delimited numbers	dlmread	dlmwrite
Text	any, including: CSV TXT	Delimited numbers, or a mix of strings and numbers	textscan	none
		Column-oriented delimited numbers or a mix of strings and numbers	readtable	writetable
Spreadsheet	XLS XLSX XLSM	Worksheet or range of spreadsheet	xlsread	xlswrite
	XLSB (Systems with Microsoft®Excel® for Windows® only)	Column-oriented data in worksheet or range of spreadsheet	readtable	writetable

Use Matlab help documentation for the syntax of each of the function.

Moreover, you can browse for them by clicking on  in the *command window* and then selecting *Data Import and Export*. As shown in the Figure 21 the functions are categorised based on the file type that they are associated with. When choosing each of the *function* Matlab provides a quick help about the syntax of each of the function.



**Figure 21: Browsing through data import functions in the command window.**

## 8 Data Analysis

Matlab has many capabilities for data analysis. In addition to the build-in functions, Matlab also provides functions for data analysis by the Statistical Toolbox and the Econometric toolbox. Therefore using Matlab you can perform from simple statistical analysis like describing the features of a data sample and correlation analysis to advance analysis like linear and non-linear regression. Moreover, you can also write your own functions to implement particular algorithm required from the analysis. This section will introduced basic functions used for univariate analysis of your data and then provide same basics for linear and logistic regression analysis.

### 8.1 Descriptive statistics

Matlab provides functions for describing the features of a data sample. These descriptive statistics include measures of location and spread, percentile estimates and functions for dealing with data having missing values. The Table 14 shows the most important ones with a brief description of their use.

**Table 14: Functions for descriptive statistics**

<b>Function Name</b>	<b>Description</b>
<code>corr</code>	Linear or Rank Correlation Coefficient
<code>corrcoef</code>	Linear Correlation Coefficient with Confidence intervals
<code>cov</code>	Covariance Matrix
<code>geomean</code>	Geometric Mean
<code>grpstats</code>	Summary Statistics by Group
<code>kurtosis</code>	Kurtosis
<code>mad</code>	Median Absolute Deviation
<code>median</code>	50th Percentile of a Sample
<code>moment</code>	Moments of a Sample
<code>nanstat</code>	Descriptive Statistic ignoring NaNs (missing data) Replace the particle <b>stat</b> by one of the following names: <i>max</i> , <i>min</i> , <i>std</i> , <i>mean</i> , <i>median</i> , <i>sum</i> or <i>var</i> .

prctile	Percentiles
quantile	Quantiles
range	Range
skewness	Skewness
std	Standard deviation
var	Variance

As an example, let's find some sample statistics of the time series variables  $x_t$ ,  $w_t$  and  $a_t$  generated in the first code section of the script shown in Figure 22. Before doing it, we must know that in a data matrix, MATLAB automatically interprets columns as samples and rows as observations. Therefore, the output of all these functions applied to the data matrix is a row vector with its  $i^{th}$  element being the value of the statistics for the  $i^{th}$  column. Let's now calculate the mean, standard deviation, median, skewness, kurtosis, and the 25<sup>th</sup> and 75<sup>th</sup> Percentile of the selected series by running the second section of the script. Note that to display the output all the results can be written in a cell table as shown in the final section of the script.

```

1  %% Generate Data
2  clear all % Clearing the workspace
3  T =1000; % Sample size
4  a = normrnd(0,1,T,1); % Generating the white noise process
5  yt = zeros(T,1); xt = 3*ones(T,1); wt = zeros(T,1); % Generating the empty vectors
6  for t = 2:T, % Starting the loop at t=2
7      yt(t) = yt(t-1) + a(t); % Generating yt
8      xt(t) = 3 + 0.5 * a(t-1) + a(t); % Generating xt
9      wt(t) = -0.8 * wt(t-1) + 0.5 * a(t-1) + a(t); % Generating wt
10 end % Finishing the loop
11
12 %% Generate statistics
13 avg = mean([yt xt wt a]); % Calculating the mean for each series)
14 med = median([yt xt wt a]); % Calculating the median for each series
15 sd = std([yt xt wt a]); % Calculating the standard deviation for each series
16 skew = skewness([yt xt wt a]); % Calculating the Skewness
17 kurt = kurtosis([yt xt wt a]); % Calculating the Kurtosis
18 prct25 = prctile([yt xt wt a],25); % Calculating the 25th Percentile
19 prct75 = prctile([yt xt wt a],75); % Calculating the 75th Percentile
20
21 %% Write Stats in a cell Table
22 Table = cell(5, 8); %Create an empty table
23 Header = {'Variables Names','mean', 'standard deviation', 'median', ...
24 'skewness', 'kurtosis', '25th Percentile', '75th Percentile'};
25 VariablesNames = {'yt', 'xt', 'wt', 'a'};
26 Table(1,:) = Header; % Set the Titles of each column
27 Table(2:end,1) = VariablesNames; % set the names of the variables
28 Table(2:end,2:end) = num2cell([avg' sd' med' skew'...
29 kurt' prct25' prct75]); %Put the results into cells. Note that
30 % the results are converted into columns.

```

Figure 22: Script for generating descriptive statistics

## 8.2 Statistical plots

The graphical methods of data analysis are as important as the descriptive statistics to collect information from the data. There is an extended support for box plots, normal probability plots, Weibull probability plots, control charts, quantile-quantile plots and polynomial curve fitting and prediction. There are also functions to create scatter plots or matrices of scatter plots for grouped data, and to identify points interactively on such plots. It is true that the best way of plotting some data strongly depends on their own nature. Thus, for instance, qualitative data information might be summarized in pie charts or bars, while quantitative data in scatter plots. The Table 15 shows some of the most known specialized graphs.

**Table 15: Functions for Statistical Plotting**

<i>Statistical Plotting</i>	<i>Description</i>
bar	Data plotting in vertical bars
barh	Data plotting in hertical bars
boxplot	Boxplots of a data matrix (one per column)
cdfplot	Plot of empirical cumulative distribution function
ecdfhist	Histogram calculated from empirical cdf
gline	Point, drag and click line drawing on figures
gplotmatrix	Matrix of scatter plots grouped by a common variable
gscatter	Scatter plot of two variables grouped by a third
hist	Histogram
hist3	Three-dimensional histogram of bivariate data
ksdensity	Kernel smoothing density estimation
lsline	Add least-square fit line to scatter plot
normplot	Normal probability plot
parallelcoords	Parallel coordinates plot for multivariate data
Pie	Pie chart of data
scatter	Scatter plot of two variables
stem	Plot of discrete data as lines (stems) with a marker
probplot	Probability plot
qqplot	Quantile-Quantile plot

Obviously, the histogram is the most popular in describing a sample. Let's

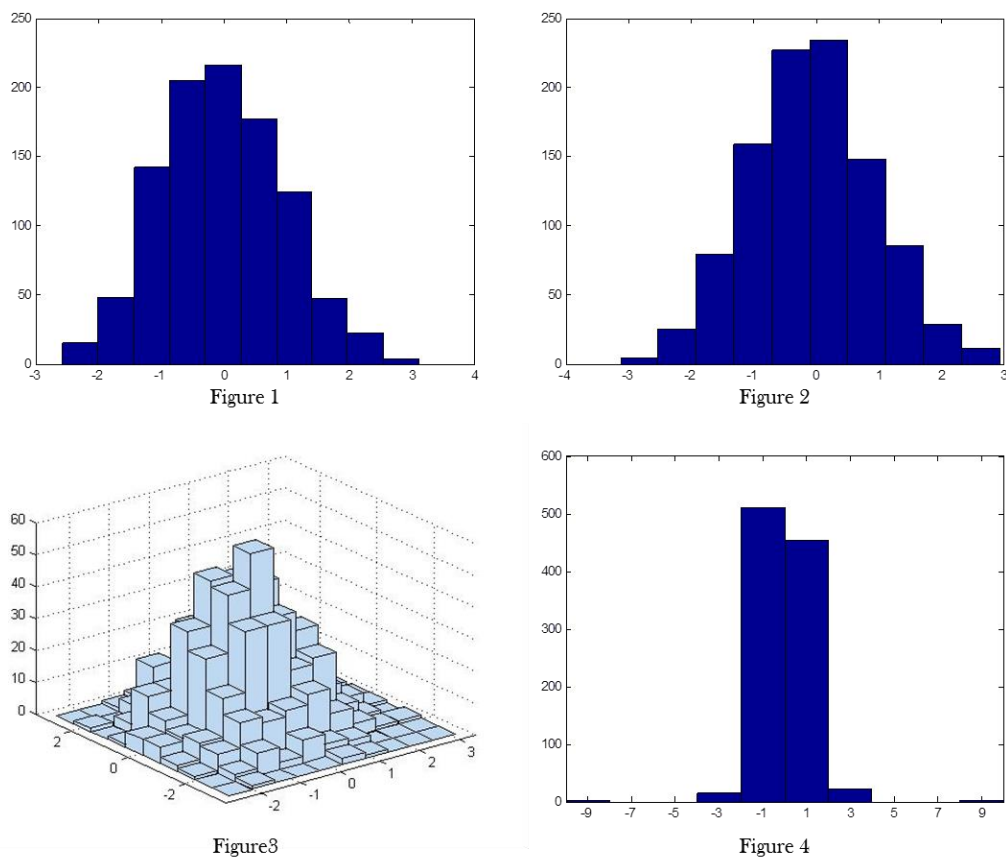
find out how to use it. Extending the above script variable X is a vector of 1000 standard normal random numbers. We will obtain the histogram for each variable and a 3-D version of the histogram for both at the same time by running the following code:

```

%% Continue from the previous sections to plot histograms
X = normrnd(0,1,1000,1); % generates 1000 random numbers from the
    % normal distribution with mean 0 and standard
    % deviation 1
figure(1),hist(X) % Histogram of X
figure(2),hist(a) % Histogram of a
figure(3),hist3([X a]) % Histogram of the bivariate sample [X,a]
figure(4),hist([X;10;-10]) % Histogram of X contaminated with two outliers”

```

The resulting figures are as shown in Figure 23.



**Figure 23: Plotting Histograms output**

The histograms serve mainly for giving a first view of the shape of the distribution, and thus infer whether the data come from a given known distribution. Moreover, they also serve to detect outliers, as shown in the bottom-right figure above. This also shows the histogram of  $X$ , but after adding two “large” (in absolute value) observations. It is clear that these two outliers significantly change the shape of the histogram.

### **8.3 Statistical Tests**

Matlab provides functions for performing a variety of statistical tests. In deciding which test is appropriate to use, it is important to consider the type of variables that you have (i.e., whether your variables are categorical, ordinal or interval and whether they are normally distributed). Table 16 summarise some of the most commonly used tests:

**Table 16: List of Statistical tests when to be used and the corresponding Matlab function**

<b>Nature of Independent Variable</b>	<b>Nature of Dependent Variable(s)</b>	<b>Test(s)</b>	<b>Matlab function</b>
0 IVs (1 population)	interval & normal ordinal or interval categorical (2 categories) categorical	one-sample t-test one-sample median binomial test Chi-square goodness-of-fit	ttest signrank binofit chi2gof
1 IV with 2 levels (independent groups)	interval & normal ordinal or interval categorical	2 independent sample t-test Wilcoxon-Mann Whitney test Chi- square test Fisher's exact test	ttest2 ranksum crosstab fishertest
1 IV with 2 or more levels (independent groups)	interval & normal ordinal or interval categorical	one-way ANOVA Kruskal Wallis Chi- square test	anova kruskalwallis crosstab
1 IV with 2 levels (dependent/matched groups)	interval & normal ordinal or interval categorical	paired t-test Wilcoxon signed ranks test McNemar	ttest signrank mcnemar

The last function in the Table 16, `mcnemar`, can be retrieved from Matlab File Exchange website.

As an example, let's perform one sample t-test that allows us to test whether a sample mean (of a normally distributed interval variable) significantly



differs from a hypothesized value. Therefore to check if the mean of the variable  $X$ , as previously generate, differs from zero type the following command:

```
“[h,p,ci,stats] = ttest(X,0);”
```

The function `ttest` returns four variables. The binary variable `h`, that is the hypothesis test result, that takes the value of 1 if the null hypothesis is rejected and otherwise the value of zero. In this case as expected the variable `h` is zero this indicates indicates a failure to reject the null hypothesis (i.e. the mean of Variable  $X = 0$ ). Note since we have not specified the level of significance for accepting or rejecting the null hypothesis Matlab uses the default that is 0.05. The second output of the function is the variable `p`, which is the p-value of the test. The third output is the two-element vector `ci`, that contains the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval. Finally, the fourth output is a structure, `stats`, that contains the following fields:

- `tstat`, Value of the test statistic.
- `df`, Degrees of freedom of the test.
- `Sd`, Estimated population standard deviation.

## 8.4 Regression Analysis

Matlab offers a variety of functions to perform different type of regressions like Linear Regression, Generalized Linear Models and Nonlinear Regression. Those functions are mostly located in the Statistical Toolbox, however the Econometric Toolbox has also a number of function to perform a particular regression analysis. In this section only linear and logistic models are presented.

### 8.4.1 Linear Regression - ordinary least squares (OLS)

Linear regression is used to predict the value of a dependent variable ( $Y$ ) from the value(s) of independent variable(s) ( $X_i$ ). The relationship is given by the equation:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 \dots + \beta_i X_i + \epsilon$$

where  $\alpha$  is the constant term,  $\beta_i$  is coefficient corresponding to the  $i^{\text{th}}$  independent variable, and  $\epsilon$  is an error term that is normally distributed with zero mean and constant variance.

Matlab has two basic functions for performing this linear regression: `regress` and `regstats`. Both of them, in addition to the regression coefficient, can return statistics that are useful in assessing the model (like r-square). The main inputs of both of those functions are two numeric variables  $Y$  that is a vector variable corresponding to the depended variable and the  $X$  variable that can be either a vector or matrix that have each of the independent variables in columns, therefore each column represents an explanatory variable and each row represents an observation. Missing data are allowed in both  $Y$  and  $X$ , and those observations are automatically removed. Note that in the case of the `regress` function a column of all ones is needed to include the constant term in the model.

The example in Figure 24 shows the use of both of those functions. In this example data are loaded from the file “RegressionAnalysisData.mat” that contains several vectors variables: `Year`, `gvkey`, `FF48`, `log_Salary`, `lag1_ROA`, `lag1_Return`, `lag1_log_at`, `lag1_mb` and `Duality`. All those variables are either directly taken or calculated from data from Compustat and Execucomp. After loading the data we are running regression model a using as depended variable the `log_Salary` and independent variables: `lag1_ROA`, `lag1_Return`, `lag1_log_at` and `lag1_MB`.

```

1  %% Load Data;
2  - clear all; %Clear the workspace
3  - load('RegressionAnalysisData.mat'); % Load data from the file
4
5  %% Run the Linear regression
6  % Using as:
7  % depended variable: log_Salary
8  % independent variables: lag1_ROA, lag1_Return lag1_log_at lag1_mb
9
10 - X = [lag1_ROA lag1_Return lag1_log_at lag1_mb]; % Create a matrix will all the
11          % independent variables
12 - Y = log_Salary; % Variable with the dependent
13
14 - [b,bint,r,rint,stats] = regress(Y,[ones(size(X,1),1) X]); %Run the regression
15 % using regress function. Note that to compute the regression coefficients
16 % for a linear model with an interaction term, need to add a first column of 1s to X
17 - StatsReg = regstats(Y, X, 'linear'); %Run the regression
18 % using regstats function. By default, regstats adds a first column of 1s to X
19

```

**Figure 24: Linear Regression example script using regress and regstats functions**

Executing the code in the section produced the results from both functions. The results are identical but are given in different form. The regress function returns five outputs:

- the vector **b** of coefficient estimates
- the matrix **bint** of 95% confidence intervals for the coefficient estimates
- the vector **r** of residuals
- the matrix **rint** of intervals that can be used to diagnose outliers. If the interval **rint(i,:)** for observation *i* does not contain zero, the corresponding residual is larger than expected in 95% of new observations, suggesting an outlier
- the vector **stats** that contains, in order, the  $R^2$  statistic, the F statistic and its p value, and an estimate of the error variance.

The function `regstats` returns a structure `StatsReg` that contains significantly more information about the regression model. Its fields are shown in the Table 17

**Table 17: Fields of the output structure of function regstats**

<b>Field</b>	<b>Description</b>
Q	Q from the QR decomposition of the design matrix
R	R from the QR decomposition of the design matrix
beta	Regression coefficients
covb	Covariance of regression coefficients
yhat	Fitted values of the response data
r	Residuals
mse	Mean squared error
rsquare	R2 statistic
adjrsquare	Adjusted R2 statistic
leverage	Leverage
hatmat	Hat matrix
s2_i	Delete-1 variance
beta_i	Delete-1 coefficients
standres	Standardized residuals
studres	Studentized residuals
dfbetas	Scaled change in regression coefficients
dffit	Change in fitted values
dffits	Scaled change in fitted values
covratio	Change in covariance
cookd	Cook's distance
tstat	t statistics and p-values for coefficients
fstat	F statistic and p-value
dwstat	Durbin-Watson statistic and p-value

Recently a new function has been added for performing linear regression, it is called `fitlm`. The next section in the code, Figure 25, shows the syntax of this function (as always using Matlab help, you can obtain more details

about using the function with given examples). One advantage of this function is the way that the results are output. Note that by allowing Matlab to write in the Command Window (i.e. no “;” at the end of the command) a summary of the results is displayed. The output of this function is also an object (structure) that contains various fields, including a table with coefficient estimates with the corresponding standard errors, t-statistics and p-values, several measurements helpful in finding outliers and influential observations, and different measurements of residuals. The complete description of the output can be found in Matlab help.

```

20 %% Regression Using fitlm
21 % Run the regression using the fitlm function
22 - OutModel = fitlm(X,Y,'linear', 'VarNames', ...
23     {'lag1_ROA', 'lag1_Return', 'lag1_log_at', 'lag1_mb', 'log_Salary'})
24
25

```

**Figure 25: Code section for performing linear regression using fitlm**

Working with panel data fixed effects regressions are very important, as data often fall into categories such as industries, states, families, etc. When you have data that fall into such categories, you will normally want to control for characteristics of those categories that might affect the dependent variable. Unfortunately, you can never be certain that you have all the relevant control variables, so you will have to worry about unobservable factors that are correlated with the variables that you included in the regression. Omitted variable bias would result. If you believe that these unobservable factors are time-invariant, then fixed effects regression will eliminate omitted variable bias. In estimating fixed effect model a dummy variable for each group is included, leaving out one group (that will be the reference group)

Continuing on the previous example, the code section shown in the figure () illustrate an easy way to construct those dummy variables for both Year and FF48 variables using the functions `categorical` and `dummyvar`. Then those variables are included in the model, remembering to omit one of them.

```

26 %% Include fixed effects
27 - DummyYear = dummyvar(categorical(Year));
28 - DummyFF48 = dummyvar(categorical(FF48));
29
30 %% Regress with Fixed effects
31 - [b,bint,r,rint,stats] = regress(Y,[ones(size(X,1),1) X DummyYear(:,1:end-1)...
32     DummyFF48(:,1:end-1)]);
33

```

**Figure 26: Code section to create Fixed Effects Variables**

Moreover, using the `fitlm` function this operation can be even simpler. Note that using `fitlm` function random effects can also be included in the model. The code section as in Figure 27 shows how to use this function and include fixed effects. In this example the vectors are converted to Matlab data structure called `Table`. Then variables `Year` and `FF48` of the table `InputTable` are converted to categorical ones and the regression model is specified by using variables names.

```

33
34 %% Fixed Effects using fitlm
35 % Converts the Vectors to a table, InputTable.
36 % Each vector becomes a variable in Table.
37 VarNames = {'Year' 'gvkey' 'FF48' 'log_Salary' 'lag1_ROA' ...
38             'lag1_Return' 'lag1_log_at' 'lag1_mb' 'Duality'};
39 InputTable = array2table([Year gvkey FF48 log_Salary lag1_ROA ...
40                         lag1_Return lag1_log_at lag1_mb Duality], 'VariableNames', VarNames);
41 % Covert Year and FF48 to categorical
42 InputTable.FF48 = categorical(InputTable.FF48);
43 InputTable.Year = categorical(InputTable.Year);
44 % Specify the regression model
45 modelspec = ['log_Salary ~ lag1_ROA + lag1_Return '...
46             '+ lag1_log_at + lag1_mb + Year + FF48'];
47 % Run the regression
48 OutModelFF = fitlm(InputTable, modelspec);
49

```

**Figure 27: Code section for performing linear regression using fixed effects with `fitlm` function**

In regression analysis, basic forms of models make use of the assumption that the errors have the same variance across all observation points. When this is not the case, the errors are said to be heteroscedastic. If there exists heteroscedasticity then estimators of parameters for linear regression are still unbiased and consistent, but the standard errors are not efficient. For this case a function named `hac` from the econometric toolbox can be used to perform the regression.

## 8.4.2 Logistic Regression

Binary logistic regression models are based on a dependent variable that can take on only one of two values. In this setting, the independent (sometimes called explanatory or predictor) variables are used for predicting the probability of occurrence of an outcome. The independent variables can be either continuous or categorical.

Logistic analysis is used to create an equation that can be used to predict

the probability of occurrence of the outcome of interest, to assess the relative importance of independent variables, and to calculate odds ratios that measure the importance of an independent variable relative to the response. The odd of an event measures the expected number of times an event will occur relative to the number of times it will not occur. Thus, if the odds of an event is 5, this indicates that we expect five times as many occurrences as non -occurrences.

The logistic equation is given by:

$$p(Y) = \frac{e^Y}{1 + e^Y}$$

where Y

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 \dots + \beta_n X_n + \epsilon$$

The above equation can be modeled in Matlab using `mnrfit` function. The code in the section in Figure 28 demonstrates the use of the function. The dependent variable is a binary variable `Duality` and the independent variables are the same as the previous example. Note that since this function requires the dependent variable to contain positive integer category numbers we add one to the binary variable. Moreover, it important to notice, that this function models the probability that the dependent variable equals to its minimum value. Therefore in this case the probability modeled is that `Duality` variable equals zero. If you need to model the probability that `Duality` equals one then the variable needs to be inverted.

```

51 %% Run the Logistic regression
52 % Using as:
53 % depended variable: Duality
54 % independent variables: lag1_ROA, lag1_Return lag1_log_at lag1_mb
55
56 - X = [lag1_ROA lag1_Return lag1_log_at lag1_mb]; % Create a matrix will all the
57 % independent variables
58 - Y = Duality; % Variable with the dependent
59 - [B,dev,LogicStat] = mnrfit(X,Y + 1); % Perform Multinomial logistic regression
60

```

**Figure 28: Code Section for performing logistic regression**

The function `mnrfit` returns three variables:

- The vector **B** that contains the coefficient estimates for a multinomial logistic regression.

- The variable `dev` that is the deviance of the fit. It is twice the difference between the maximum achievable log likelihood and that attained under the fitted model.
- The structure `LogicStat` that contains the fields as show in

**Table 18: Fields of the structure `LogicStat`**

<b>Field</b>	<b>Description</b>
<code>beta</code>	The coefficient estimates. These are the same as <code>B</code> .
<code>dfe</code>	Degrees of freedom for error
<code>sfit</code>	Estimated dispersion parameter.
<code>s</code>	Theoretical or estimated dispersion parameter.
<code>estdisp</code>	Indicator for a theoretical or estimated dispersion parameter.
<code>se</code>	Standard errors of coefficient estimates, <code>B</code> .
<code>coeffcorr</code>	Estimated correlation matrix for <code>B</code> .
<code>covb</code>	Estimated covariance matrix for <code>B</code> .
<code>t</code>	$t$ statistics for <code>B</code> .
<code>p</code>	$p$ -values for <code>B</code> .
<code>resid</code>	Raw residuals. Observed minus fitted values,
<code>residp</code>	Pearson residuals
<code>residd</code>	Deviance residuals



## 9 Case Studies

In the following sub-section, some very useful examples with *functions* and *scripts* are illustrated. Read, implement and execute the codes that are given. Try to change the code. Experiment by adding your own command lines. You can find all needed *function*, *scripts* and datasets in the folder named as:

“Matlab Examples”

on my personal website. Depending on the place you are, this folder may be in different locations.

### 9.1 The Black - Scholes- Merton Options Pricing Formula

The Black Scholes Merton (BSM) model expresses the value of an option as a function of the current value of a stock,  $S$ , the option's strike price,  $X$ , the option's time to maturity,  $T$ , the volatility,  $s$ , of the stock price returns (this is the standard deviation of the log-relative returns of the stock for the past  $n$  days, where  $n$  is usually set equal to 60), the risk free rate,  $r$ , that prevails for a maturity  $T$  and the stock's dividend yield,  $d$ . The analytic formula for the a European call option derived by BSM is:

$$c^{BSM} = Se^{-dT} N(d_1) - Xe^{-rT} N(d_2)$$

where,

$$d_1 = \frac{\ln(S/X) + (r - d + s^2/2)T}{s\sqrt{T}},$$

$$d_2 = d_1 - s\sqrt{T},$$

and,

$$N(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^z e^{-\frac{1}{2}z^2} dz$$

represents the cumulative normal distribution function with mean zero and unity standard deviation (the Matlab build in function is named as: normcdf).

The value of a European put option can also be defined via the BSM as:

$$p^{BSM} = Xe^{-rT} N(-d_2) - Se^{-dT} N(-d_1)$$

### 9.1.1 Implementation of the BSM Formula

We want to create a *script* named as *OptionsBSM.m* that calls the user created *function* BSMprice to price European call, c, and European put, p, options using the Black, Scholes and Merton model (BSM). The function BSMprice should take seven (at least six) input parameters (as single values or in the form of a vector) and should return the value (or the vector) of a European call or put. Its *calling syntax* should look like:

“[Price] = BSMprice(S, X, T, vol, r, Index, dv)”

“Index” is a string input argument that is set as an additional input argument that takes the values: 'CALL' (or any other spelling of the 'CALL' such as 'Call', 'CaLL', etc), or 'PUT' (or any other spelling of the 'PUT' such as 'Put', 'PuT', etc) and should be used to define the kind of option type (use the build in function lower to convert the “Index” to lower case). Note the “Index” precedes “dv” because “dv” should be an optional input (will come back to this in a while).

To validate the formula, via the *script* create a table that tabulates the call and put values against the moneyness ratio, S, and maturity, T. The data should be loaded from a text file (that the user creates either by hand or via a *function*) named as “options\_data.txt” in which the columns should contain S, X, T, s, r, d. The data values are given below:

$$S = \{80,85,90,95,100,105,110,115,120\} ,$$

$$T = \{0.1,0.15,0.20,0.25\}$$

$$s = \{0.1,0.2,0.3\}$$

and,

$$X = 100, r = 0.05, d = 0.02$$

After creating the text file, you should have a *108-by-6* table since for each value of  $T$  and  $s$  you have nine different possible values for  $S$ .

Note that the function:

- Should check if the correct number of input arguments is supplied and display an error message otherwise (to do so, use the build-in function `nargin`; see the help facilities for understanding its use).
- Set dividends equal to zero automatically if no dividend input is supplied or if the dividend input argument is set to be an empty array (use the build in function `exist`).
- In order to prevent an error, the function should check whether the input parameters are all positive or non-negative (elaborate on the BSM formula to see which inputs should be  $\geq 0$  and which only  $> 0$ ;  $d$  and especially  $r$  with negative values can exist but do not have any economical implication).
- If “Index” takes a different value other than those explained before, an error message should be returned and the program execution should break. Additionally, an error should be returned and program execution should stop if “Index” is a string matrix and not a string vector (“Index” should be either *1-by-4* string vector in the case of a 'CALL' or a *1-by-3* in the case of a 'PUT').

### 9.1.2 BSM Derivatives

From the same *script* as above, call a *function* with the name: `BSMderivatives` that returns the partial derivatives of the BSM formula (known as the Greek letters) for both the call and the put value. The partial derivatives are:

$$\begin{aligned}
\text{delta: } \partial c / \partial S &= N(d_1)e^{-dT}, \quad \partial p / \partial S = -N(-d_1)e^{-dT} \\
\text{theta: } \partial c / \partial T &= -\frac{n(d_1)Ss}{2\sqrt{p}} - rXe^{-rT}N(d_2) \\
\partial p / \partial T &= -\frac{n(d_1)Ss}{2\sqrt{p}} + rXe^{-rT}N(-d_2) \\
\text{gamma: } \partial^2 c / \partial S^2 &= \frac{n(d_1)}{Ss\sqrt{T}}, \quad \partial^2 p / \partial S^2 = \frac{n(d_1)}{Ss\sqrt{T}} \\
\text{vega: } \partial c / \partial s &= S\sqrt{T}n(d_1), \quad \partial p / \partial s = S\sqrt{T}n(d_1) \\
\text{rho: } \partial c / \partial r &= XTe^{-rT}N(d_2), \quad \partial p / \partial r = -XTe^{-rT}N(-d_2)
\end{aligned}$$

where,

$$n(z) = \frac{1}{\sqrt{2p}} e^{-z^2/2}$$

represents the normal probability density function with mean zero and unity standard deviation (the built in function is named as `normpdf`). The BSMderivatives *calling syntax* should be:

“[Derivatives] = BSMderivatives(S, X, T, vol, r, Index, dv)”

The *function* should make similar checks as before. The output of the *function* BSMderivatives should be a row vector if the input arguments are single values or a two dimensional array if the input arguments are vectors with the following format:

“Derivatives = [Delta, Theta, Gamma, Vega, Rho]”

### 9.1.3 BSM Plots and Surfaces

Make the following plots:

- Two subplot figures, one for call options with  $T=0.15$  and  $s=0.20$ , and one for put options with  $T=0.25$  and  $s=0.20$  that plot S against all partial derivatives. Give titles and axis names.
- A 3D surface of the  $c^{BSM}$  and  $p^{BSM}$  versus the  $S/X$  and  $T$  (note that in order to create a smooth surface you should create a dense mesh-grid) for the ranges:

$$80 \leq S \leq 120, \quad 0.10 \leq T \leq 0.25$$

for:

$$X = 100, s = 0.20, r = 0.05, d = 0.02$$

### 9.1.4 BSM Implied Volatility

The only unobservable parameter related with the BSM formula when we deal with real world problems is the volatility measure,  $s$  (although as said before it can be estimated via log-relative reruns of the underlying asset). It is accustomed by options traders to observe a market value for a call or put option and given the observed values of  $S$ ,  $X$ ,  $T$ ,  $r$ , and  $d$ , they derive the value of  $s$  that equates the BSM estimate with the market quote. Such  $s$  value is known as *implied volatility*. For instance, if  $c^{mrk}$  is the market price of a call option and  $S$ ,  $X$ ,  $T$ ,  $r$  and  $d$  are the observable parameters related with the call option, then the BSM implied volatility,  $s_{imp}$ , is that value of  $s$  that minimizes the absolute error between  $c^{mrk}$  and  $c^{BSM}$ . Analytically this problem is:

$$\min_{s_{imp}} |g(e)| = |c^{mrk} - c^{BSM}(S, X, T, s_{imp}, r, d)|$$

Alternatively, think this as solving the following equation:

$$c^{mrk} = c^{BSM}(S, X, T, s_{imp}, r, d)$$

for  $s_{imp}$ . It is known that such equation has no analytic solution, so to solve it someone has to implement a numerical root-finding algorithm. The most well-known root finding algorithm for the implied volatility problem is the Newton-Raphson method.

The Newton method is very simple and quite fast method. It can be summarized in three steps:

- **Step #1:** give an initial guess for  $s_{imp}$  (e.g.  $s_{imp}^0 = 0$ )
- **Step #2:** perform an algorithm iteration,  $k$ , based on the following formula:

$$s_{imp}^k = s_{imp}^{k-1} - \frac{g(e)}{g'(e)} = s_{imp}^{k-1} - \frac{c^{mrk} - c^{BSM}}{-\frac{\partial c^{BSM}}{\partial S}} \Rightarrow$$

$$s_{imp}^k = s_{imp}^{k-1} + \frac{c^{mrk} - c^{BSM}}{\frac{\partial c^{BSM}}{\partial S}}$$

for  $k=1,2,3,\dots$

- **Step #3:** If  $|g(e)| < e$  then stop and return  $s_{imp}^k$  **otherwise** go to Step #2 and perform one more iteration,  $k$ , of the algorithm (the  $e$  is the desire accuracy, usually set to a very small quantity)

Write a *function* with the following *calling syntax*:

“[ImpliedVol] = BSMimpliedVol(Qmrk, S, X, T, r, Index, dv, maxNumIter, tol)”

where “Qmrk” is the market quote of a call or put option. “maxNumIter” represents the number of iterations and “tol” the desire accuracy (it is a stopping criterion related with the absolute difference of market option value and BSM estimate). “Index” is a single string with values “CALL” and “PUT” (or any other combination that delivers the required word meaning) that is used to distinguish between calls and puts as before. Note that all input arguments precede the optional ones (the optional are “dv”, “maxNumIter” and “tol”).

Use as inputs the initial data from the loaded *txt* file (without considering  $s$ ). The function should make similar checking like previous *functions* with the additional that:

- If not specified, “maxNumIter” should be equal to 100.
- If *not* specified, “tol” should be  $1e-5$ .
- At the beginning of the function, it should be tested whether with a small value of  $s$  (e.g.  $s=0.001$ ) the  $c^{BSM}$  and  $p^{BSM}$  are greater than  $c^{mrk}$  and  $p^{mrk}$  respectively; if such condition holds (it can be observed in practice as an arbitrage result), an implied volatility value that minimizes  $|g(e)|$  does not exist, so the function should return a “NaN” value.
- To help the algorithm convergence rate and to save computing time, the

starting values for the algorithm should be given from the following approximation suggested by Bharadia, Christofides and Salkin [3]:

$$\text{for call options: } s_{imp}^0 = \sqrt{\frac{2p}{?} \left( \frac{c^{mk} - d}{Xe^{-rT} + d} \right)}, \text{ and,}$$

$$\text{for put options: } s_{imp}^0 = \sqrt{\frac{2p}{?} \left( \frac{p^{mk} + d}{Xe^{-rT} + d} \right)}$$

with,

$$d = 0.5(S - Xe^{-rT})$$

To use the implied volatility *function*, replace the volatility column of the options data that were previously loaded from the text file with random values taken from a uniform distribution on the interval [0,1] (use the build in function: rand). Afterwards, use the `BSMprice` to find the theoretical values for calls and puts. In the following, use the `BSMimpliedVol` to verify that the implied volatility is quite close with the ones that you have initially used (these are the random values) to price calls and puts with the BSM. Note that the implied volatility of calls and puts for should match for similar input arguments. Plot the difference to visualize that always the dollar difference is less than “tol” (given that the algorithm iterations were adequate to secure convergence).

## 9.2 Function Minimization and Plots

There is a variety of optimization algorithms that can be used to minimize (or to maximize) a univariate or a multivariate function (note that the maximization of a function,  $f(x)$ , is the same as the minimization of  $-f(x)$ ). Two well known and widely used algorithms are the *gradient descent*, that minimizes a function by relying solely on a function’s first partial derivatives (gradient vector), and the *Newton’s descent* algorithm that utilizes a function’s second order partial derivatives (Hessian matrix).

The gradient descent algorithm can be summarized as follows:

- **Step #1:** give an initial guess for the coordinates of the minimum point  $x^0$  (e.g. if the function under consideration has two unknowns  $x_1$  and  $x_2$ , then  $x$  should be a two element row vector:  $x^0 = [x_1^0, x_2^0]$ )

- **Step #2:** perform an algorithm iteration,  $k$ , based on the following formula:

$$x^{k+1} = x^k - \alpha f'(x^k)$$

for  $k=1,2,3,\dots$

where  $f(x^k)$  is the value of the function at  $k^{\text{th}}$  iteration at point  $x$  and  $f'(x^k)$  is a row vector that includes the first order partial derivatives of  $f(\cdot)$  w.r.t  $x$ 's at the  $k^{\text{th}}$  iteration. The parameter  $\alpha$  is a positive scale function that lies between 0 and 1 and is a learning rate that sets the step size. Large values of  $\alpha$  force the algorithm to oscillate around the minimum point or even diverge, whereas small values of  $\alpha$  lead to more stable convergence but with extremely slow rate. Typical values of  $\alpha$  range between 0.01 and 0.15 depending on the faced problem.

Specifically,  $f'(x)$  for a function with  $n$  variables is:

$$f'(x) = \left[ \frac{\partial f(x)}{\partial x_1} \quad \frac{\partial f(x)}{\partial x_2} \quad \dots \quad \frac{\partial f(x)}{\partial x_n} \right]$$

- **Step #3:** Increase:  $k \leftarrow k+1$  and if the current iteration index  $k$  is larger than the maximum number of iterations or if  $\|\alpha f'(x^k)\| < \epsilon$  then stop and return  $x^{k+1}$ , **otherwise** go to Step #2 and perform one more iteration of the algorithm (the  $\epsilon$  is the desired accuracy, usually set to a small quantity such as  $1e-6$ , whereas the maximum number of iterations depends solely by the experience of the researcher).

The Newton Descent algorithm can be summarized as follows:

- **Step #1:** give an initial guess for the coordinates of the minimum point  $x^0$  (e.g. if the function under consideration has two unknowns  $x_1$  and  $x_2$ , then  $x$  should be a two element row vector:  $x^0 = [x_1^0, x_2^0]$ )
- **Step #2:** perform an algorithm iteration,  $k$ , based on the following formula:

$$x^{k+1} = x^k - f'(x^k) [f''(x^k)]^{-1}$$

for  $k=1,2,3,\dots$



where  $f(x^k)$  is the value of the function at  $k^{\text{th}}$  iteration at point  $x$ ,  $f'(x^k)$  is a row vector that includes the first order partial derivatives of  $f(\cdot)$  w.r.t  $x$ 's at the  $k^{\text{th}}$  iteration and  $[f''(x^k)]^{-1}$  is a square matrix that represents the inverse of the second order partial derivatives of  $f(\cdot)$  w.r.t  $x$ 's (Hessian matrix) at the  $k^{\text{th}}$  iteration.

Specifically,  $f''(x)$  for a Function with  $n$  variables is:

$$f''(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}$$

- **Step #3:** Increase  $k \leftarrow k+1$  and if the current iteration index  $k$  is larger than the maximum number of iterations or if  $\|f'(x^k)[f'(x^k)]^{-1}\| < e$

Then stop and return  $x^{k+1}$ , **otherwise** go to Step#2 and perform one more iteration of the algorithm (the  $e$  is the desire accuracy, usually set to a small quantity such as  $1e-6$ , whereas the maximum number of iterations depends solely by the experience of the researcher).

Both the gradient descent and the Newton descent algorithms have inherent problems with their implementation. For example, the user should “magically” define the correct value of the learning parameter  $a$  in order to succeed a robust convergence of the algorithm when it comes to implement the gradient descent algorithm, or concerning the Newton descent one, when the Hessian matrix is singular and its inverse does not exist then it is not applicable. Moreover, there is no way to secure that the initial point to set up the algorithm will eventually help in reaching the minimum point. Most of the times and when the user does not have a detail overview of the faced problem, the initial value of  $x$  might be too far from the minimum point, preventing in this way the algorithm to converge.

There are many suggestions to suppress these problems (i.e [4], [5]). In here, we

will implement these two algorithms just for getting the feeling on the way that a minimization algorithm can be used. The user can later improve the *function* that will be illustrated according to his/her needs.

To implement the previous algorithms, the user should be in a position to find the gradient vector and the Hessian matrix. Although it is better to work with the exact first and second order partial derivatives of the function via their functional form, a more flexible, practical and quite accurate way is to create two function in Matlab that calculate these with two sided finite differencing methods. That is, instead of evaluating a function at a point  $x$  by using the functional form of the gradient vector and the Hessian matrix, it is easier to find these information via numerical differentiation (two sided finite differencing).

The centred and evenly spaced finite difference approximation of the first order partial derivatives (gradient vector) of a function  $f$  at point  $x$  is (for simplicity assume a two variable function with  $x_1$  and  $x_2$  to be the unknown variable - the extension to the  $n$  dimensional case is trivial):

$$\frac{\partial f(x)}{\partial x_1} \approx \frac{f(x_1 + h, x_2) - f(x_1 - h, x_2)}{2h}$$

$$\frac{\partial f(x)}{\partial x_2} \approx \frac{f(x_1, x_2 + h) - f(x_1, x_2 - h)}{2h}$$

where  $h$  is given from a rule of thumb (see [4] page 103),

$$h = \max(|x_i|) \sqrt[3]{\epsilon}$$

with  $\epsilon$  to represent machine precision (this is the *build-in function eps*).

The centred and evenly spaced finite difference approximation of the second order partial derivatives (Hessian matrix) of a function  $f$  at point  $x$  is (for simplicity assume a two variable function with  $x_1$  and  $x_2$  to be the unknown variable - the extension to the  $n$  dimensional case is trivial):

$$\frac{\partial^2 f(x)}{\partial x_1^2} \approx \frac{f(x_1 + 2h, x_2) + f(x_1 - 2h, x_2) - f(x_1 - h, x_2) - f(x_1 + h, x_2)}{4h^2}$$

$$\frac{\partial^2 f(x)}{\partial x_1 \partial x_2} \approx \frac{f(x_1 + h, x_2 + h) + f(x_1 - h, x_2 - h) - f(x_1 - h, x_2 + h) - f(x_1 + h, x_2 - h)}{4h^2}$$

$$\frac{\partial^2 f(x)}{\partial x_1 \partial x_2} \equiv \frac{\partial^2 f(x)}{\partial x_2 \partial x_1}$$

$$\frac{\partial^2 f(x)}{\partial x_2^2} \approx \frac{f(x_1, x_2 + 2h) + f(x_1, x_2 - 2h) - f(x_1, x_2 + h) - f(x_1, x_2 - h)}{4h^2}$$

where  $h$  is given from a rule of thumb (see [4] page 103),

$$h = \max(|x|, 1) \sqrt[4]{\epsilon}$$

Having in mind the above, write a script with the name: *FunctionMin.m* that will do the following:

- Make the 3D - surface and the contour plots of the function:

$$f(x_1, x_2) = e^x (4x^2 + 2y^2 + 4xy + 2y + 1)$$

in the area -  $2 \leq x, y \leq 1.5$ . The function should be saved in an *m-file* with the name: *funToMin*. Given that you have created the *m-file* with the functional form of the function, to evaluate it at a point use the build in function: *feval*. The calling syntax is:

“*feval (@funToMin,x)*”

where  $x$  is a two element row vector (or *m-by-2* array) with “*x(1)*” and “*x(2)*” to represent  $x_1$  and  $x_2$  respectively.

- Do a new figure that creates the contour plot of this function, with the value of the contours to range between 0 and 80 (for example use: “*V=[0:0.25:2, 3:1:10, 20:10:80]*”).
- Call a function with the name: *GradDescent* that implements the gradient descent algorithm for minimizing the aforementioned function. The calling syntax of this function should be:

“*[MinPoints, xVals] = GradDescent(fun, x, lr, maxIterNum, tol)*”

*fun* is the function name “*@funToMin*”,  $x$  is a row vector with the user’s guess (starting value) about the minimum of the function. The input argument “*lr*” is the learning parameter  $\alpha$  that should be set to 0.1 if not specified by the user. “*maxIterNum*” is as before the maximum number of iterations that should be set to 100 if not

specified and “tol” is the tolerance concerning the norm of the updating component of the point  $x$  (stopping criterion) and should be set to  $1e-6$  if not specified. The function returns “MinPoints” that if the algorithm has converged is the point where the function is minimized, and “xVals” that is an  $m$ -by-2 array with the set of the values of  $x$  at each iteration of the algorithm (up to either equal to: maxIterNum” or a smaller number if the algorithm converges).

This function should call another function with the name: NumerDers and with the following calling syntax:

“GradsVals = NumerDers(fun,x)”

that takes as input a function and a single coordinate point and returns the gradient vector at that point. Use the two sided finite difference method explained before. For the above, use as initial point:  $x^0=[0 \ 0]$ .

After you have called the function, plot on the contour figure the algorithm’s trajectory to the minimum found in “xVals”. Call once more the function with  $x^0=[-1 \ -2]$  and add the new trajectory to the contour figure. Use “lr”=0.25 and “maxIterNum” =200. Use gtext to enter a text that indicates the trajectory.

- Call a function with the name: Newton that implements the Newton descent algorithm for minimizing the aforementioned function. The *calling syntax* of this function should be:

“[MinPoints,xVals] = Newton(fun, x, maxIterNum, tol)”

“fun” is the *function* name “@funToMin”,  $x$  is a row vector with the user’s guess (starting value) about the minimum of the function. “maxIterNum” is as before the maximum number of iterations that should be set to 25 if not specified and *tol* is the tolerance concerning the norm of the updating component of the point  $x$  (stopping criterion) and should be set to  $1e-6$  if not specified. The *function*

returns “MinPoints” that given that the algorithm has converged, it is the point where the minimum is located, and “xVals” that is an  $m$ -by-2 array with the set of the values of  $x$  at each iteration of the algorithm ( $m$  is either equal to “maxIterNum” or a smaller number if the algorithm converges earlier).

This *function* should call another function with the name: *NumerHessian* and with the following *calling syntax*:

“HessVals = NumerHessian(fun,x)”

that takes as input a function and a single coordinate point and returns the Hessian at that point. Use the finite two sided difference method explained before. To avoid the calculation of a singular Hessian that cannot be inverted, after saving the Hessian to a square matrix, set equal to zero all its off diagonal elements. This is a very useful rule of thumb that helps the algorithm convergence and robustness. Use  $x^0=[0 \ 0]$  as initial estimate for the minimum.

After you have called the *function*, create a new contour figure of the function and plot on it the algorithm’s trajectory to the minimum found in “xVals”. Call once more the function with  $x^0=[-1 \ -2]$  and add the new trajectory to the contour figure. Use *gtext* to enter a text that indicates the trajectory.

### 9.3 Portfolio Optimization

Many times, a researcher needs to create a portfolio of securities that for a desire level of return/profit, it bears the least/minimum risk. The portfolio optimization problem has been defined since Markowitz (1952) who assumed that the risk associated with a portfolio can be measured with variance.

The researcher objective is to define via an optimization programming methodology the weights that each of the alternative assets should have in the portfolio. By construction, the portfolio return is linear w.r.t to the weights of the assets and it is given from the following relation:

$$E(r_p) = W^T e$$

where  $r_p$  is the weighted average return of the portfolio and  $E(.)$  is the expectation operation. The weight representation of each asset in the portfolio is given by  $W$  and the expected return of each asset is given by  $e$  as follows:

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad \text{and} \quad e = E \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}$$

where  $n$  is the number of available assets. Contrary, the volatility of the portfolio w.r.t the weights is a high nonlinear function given by the following relation:

$$s_p^2 = W^T O W$$

where  $s_p$  represents the weighted average volatility (standard deviation) of the portfolio, and  $O$  is the assets' variance-covariance matrix:

$$O = \begin{bmatrix} s_{11} & s_{12} & \cdots & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ s_{n1} & s_{n2} & \cdots & s_{nn} \end{bmatrix}$$

with  $s_{ij} = cov(r_i, r_j)$ . Additionally, for the portfolio optimization problem, it is required that the sum of the assets weights should be added to one:

$$W^T N_1 = 1$$

with  $N_1$  to be a column vector with unity elements.

From the above it is obvious that the portfolio optimization problem can be solved via the minimization of a quadratic volatility function subjected to linear constraints, one related with the portfolio expected return and another that assures that all available funds are invested. There are several convenient mathematical methods for solving this problem. In here, in first place, it is illustrated the one that is called the method of the *Langrangean multiplier* that allows the solution of the general portfolio problem with short sales with the requirement that the constraints must be in equality form.

Lets consider the case of three risky assets without a risk-free rate (it is trivial to generalize it to  $n$  risky assets). The inclusion of the risk free rate is easy; by treating the risk-free rate as a “risky asset” with zero volatility and zero covariance with all the other assets. The problem formulation that achieves an expected return equal to  $R$  follows:

$$\min_{w_i} W^T O W$$

s.t

$$W^T e = R$$

$$W^T N_1 = 1$$

and for the case of the three assets in an analytic form:

$$\min_{w_i} w_A^2 s_A^2 + w_B^2 s_B^2 + w_C^2 s_C^2 + 2w_A w_B s_{AB} + 2w_A w_C s_{AC} + 2w_B w_C s_{BC}$$

s.t.

$$w_A E(r_A) + w_B E(r_B) + w_C E(r_C) = R$$

$$w_A + w_B + w_C = 1$$

Note: If we do not use the first constraint and in the absence of a risk-free rate we will get the minimum variance portfolio. It is actually advisable to first get the minimum variance portfolio before we proceed in a real problem. Then, by changing the level of desire return, we trace points on the efficient frontier.

To solve this problem, we make two simplifications to the problem. First we replace everywhere  $w_C$  with  $1 - w_A - w_B$ . Second, we rearrange the constraint:

$$w_A E(r_A) + w_B E(r_B) + w_C E(r_C) = R$$

to

$$R - w_A E(r_A) + w_B E(r_B) + w_C E(r_C) = 0$$

and since it equals zero, we multiply it with a constant  $\lambda$  and we get:

$$\lambda(R - w_A E(r_A) + w_B E(r_B) + w_C E(r_C)) = 0$$

Since this equals zero, we add it to the objective function that we want to minimize without affecting the results. Finally, we minimize the quadratic function without any constraints. Thus, we have to minimize a function (so

it can also be solved with methods that we have seen before). So, after substitution we have:

$$f = w_A^2 s_A^2 + w_B^2 s_B^2 + (1 - w_A + w_B)^2 s_C^2 + 2w_A w_B s_{AB} + 2w_A(1 - w_A + w_B)s_{AC} + 2w_B(1 - w_A + w_B)s_{BC} + \lambda(R - w_A E(r_A) - w_B E(r_B) - (1 - w_A + w_B)E(r_C))$$

We need to solve this problem to find the weights of the three assets in the portfolio. We thus first find the solution to  $w_A$ ,  $w_B$  and  $\lambda$ . The way to solve the above problem is now easy. We take the partial derivatives each time in respect to one of the three unknowns and set them equal to zero, and we will derive a set of three equations with three unknowns. The first is the partial derivative of  $f$  w.r.t  $w_A$ , the second w.r.t  $w_B$  and the third w.r.t the Lagrangean multiplier  $\lambda$ .

The first equation gives:

$$2w_A s_A^2 + (2w_A + 2w_B - 2)s_C^2 + 2w_B s_{AB} + (2 - 4w_A - 2w_B)s_{AC} - 2w_B s_{BC} - \lambda E(r_A) + \lambda E(r_C) = 0$$

The second equation gives:

$$2w_B s_B^2 + (2w_A + 2w_B - 2)s_C^2 + 2w_A s_{AB} - 2w_A s_{AC} + (2 - 2w_A - 4w_B)s_{BC} - \lambda E(r_B) + \lambda E(r_C) = 0$$

The third equation gives:

$$R - w_A E(r_A) - w_B E(r_B) - (1 - w_A - w_B)E(r_C) = 0$$

To work with this case study, assume the following data:

$$e = E \begin{bmatrix} r_A \\ r_B \\ r_C \end{bmatrix} = \begin{bmatrix} 0.05 \\ 0.10 \\ 0.15 \end{bmatrix}$$

$$O = \begin{bmatrix} s_A^2 & s_{AB} & s_{AC} \\ s_{BA} & s_B^2 & s_{BC} \\ s_{CA} & s_{CB} & s_C^2 \end{bmatrix} = \begin{bmatrix} 0.25 & 0.15 & 0.17 \\ 0.15 & 0.21 & 0.09 \\ 0.17 & 0.09 & 0.28 \end{bmatrix}$$



Write a script with the name: *MeanVarMin.m* that will do the following:

- Saves in a column vector the weights for the solution of the above mean-variance minimization problem given by:

$$0.38w_A + 0.34w_B + 0.10w_C - 0.22 = 0$$

$$0.34w_A + 0.62w_B + 0.05w_C - 0.38 = 0$$

$$0.10w_A + 0.05w_B + 0w_C - 0.05 = 0$$

and

$$w_C = 1 - w_A - w_B$$

Find its expected return and its standard deviation.

- Create an m-file with the analytic expression of the function /with the name: *portFun* with the following syntax:

“f=portFun(x,r,V,R)”

with  $x$  to represent a three element vector,  $x = [w_A, w_B, w_C]$ ,  $r$  to be the vector of the assets expected return,  $V$  the variance covariance matrix and  $R$  the desire portfolio expected return.

- Assuming the inexistence of a risk-free rate, create a plot with the minimum variance opportunity set (current efficient frontier) for values of  $R$  that range between 1% and 40%. The figure should plot the standard deviation against the portfolio expected return. To do so, change slightly the *Newton* function so that its calling syntax becomes:

“[MinPoints] = NewtonPort(fun, x, r, V, R)”

with  $r$  to be the column vector of the assets’ expected returns,  $V$  the assets’ variance-covariance matrix and  $R$  the desire expected return. Similar changes in order to pass the additional arguments should be done also to *NumerDers.m* (the new should be named as: *NumerDersPort*) and *NumerHessian.m* (the new should be named as: *NumerHessianPort*). Because the portfolio optimization problem is actually the minimization of a quadratic function, the Newton descent algorithm can find its solution with a single iteration. So, alleviate also the rule of thumb according to which the elements of the

Hessian matrix except the ones of the main diagonal are set to zero. Use the origin (0, 0, 0) as an initial point.

- Add a risk free asset with expected return 1%, find and plot the efficient frontier. Create a new *m-file* with the name: *portFunRiskFree.m* that includes the new expression of  $f$  after the inclusion of risk-free rate (*hint*: find which components are zerovalued and ignore them). Create a third figure that combines the two previous ones. If you are familiar with investment theory, you can view various interesting components (e.g. two fund separation theorem, capital market line, etc).

## References

- [1] Ela Pekalska, Marjolein van der Glas, (2001), "*Introduction to MATLAB*", Pattern Recognition Group, Faculty of Applied Sciences, Delft University Technology.
- [2] MATLAB® Release 2014a Online Help.
- [3] Bharadia M. A. J., Christofides N, and Salkin G. R., (1995), "*Computing the Black and Scholes Implied Volatility*", *Advances in Futures and Options Research*, Vol. 8, pp. 15-29.
- [4] Miranda M. and Fackler P., (2002), *Applied Computational Economics and Finance*, Massachusetts Institute of Technology.
- [5] Bertsekas D., (1999), *Nonlinear Programming*, 2<sup>nd</sup> Edition, Athena Scientific.
- [6] Griffiths D., (2001), "*An Introduction to Matlab: Version 2.2*", Department of Mathematics, The University Dundee DD1 4HN.
- [7] Silvestrov D., and Malyarenko A., (2001), "*An Introduction to Financial Mathematics With MATLAB*", Malardalen University.
- [8] Chandler G., (2000), "*Introduction to Matlab*", Mathematics Department, The University of Queensland.
- [9] Ferrari S., (2000), "*Tutorials in Robotics and Intelligent Systems: Part IV. Introduction to MATLAB Optimization Toolbox*", Department of Mechanical and Aerospace Engineering, Princeton University.