

# MATLAB: A Comprehensive Introduction





# How to Use This Book

This handbook accompanies the taught sessions for the course. Each section contains a brief overview of a topic for your reference and one or more exercises.

The document is written as a PDF, with internal links as well as links to online documentation. Furthermore, it has been designed so that you can copy and paste example commands from the PDF. You should have the PDF open during the taught sessions of the course.

## The Exercises

Exercises are arranged as follows:

- A title and brief overview of the tasks to be carried out
- A numbered set of tasks, together with a brief description of each
- A numbered set of detailed steps that will achieve each task

Some exercises, particularly those within the same section, assume that you have completed earlier exercises. Your teacher will direct you to the location of files that are needed for the exercises. If you have any problems with the text or the exercises, please ask the teacher or demonstrator for help.

This book includes plenty of exercise activities. You should try them during the course, while the teacher and demonstrator(s) are around to guide you. Later, you may attend follow-up sessions at ITLC, where you can continue to work on the exercises, with some support from IT teachers. Other exercises are for you to try on your own, as a reminder or an extension of the work done during the course.

## Writing Conventions

A number of conventions are used to help you to be clear about what you need to do in each step of a task.

- In general, the word **press** indicates you need to press a key on the keyboard. **Click**, **choose** or **select** refer to using the mouse and clicking on items on the screen.
- Names of keys on the keyboard, for example the Enter (or Return) key are shown like this ENTER.
- Multiple key names linked by a + (for example, CTRL+Z) indicate that the first key should be held down while the remaining keys are pressed; all keys can then be released together.
- Words and commands typed in by the user are shown like `this`.
- Labels and titles on the screen are shown **like this**, unless there is a link to a help file in which case they are shown as follows: `help`.

- Drop-down menu options are indicated by the name of the options separated by a vertical bar, for example **File|Print**. In this example you need to select the option **Print** from the **File** menu. To do this, click with the mouse button on the **File** menu name; move the cursor to **Print**; when **Print** is highlighted, click the mouse button again.
- A button to be clicked will look like this.
- The names of software packages are identified *like this*, and the names of files to be used **like this**.

## Revision Information

Version	Date	Author	Changes Made
2.0	Nov 2010	Robert Stewart	Complete rewrite
2.1	Jan 2011	Robert Stewart	Update
2.2	May 2011	Robert Stewart	Update
2.3	Oct 2011	Robert Stewart	Update
2.4	Feb 2012	Robert Stewart	Update
2.5	May 2012	Robert Stewart	Update
3.0	Oct 2012	Robert Stewart	Major Update
3.1	Dec 2012	Robert Stewart	Update
4.0	Jan 2013	Robert Stewart	4 Session Update
4.1	May 2013	Robert Stewart	Exercise Updates
4.2	Jan 2014	Robert Stewart	Exercise Updates
4.3	Oct 2014	Robert Stewart	Version Updates
5.0	Jan 2015	Erasmia Lyka / Tasos Paspastylianou	Version Update
5.1	May 2015	Erasmia Lyka	Exercise Updates
6.0	Oct 2015	Erasmia Lyka	Exercise Updates
7.0	Oct 2016	Catherine Paverd	Version Updates
7.1	Jan 2017	Catherine Paverd	Exercise Updates
7.2	Oct 2017	Catherine Paverd	Exercise Updates
7.3	Oct 2018	Catherine Paverd	Exercise Updates

## Copyright

This document and the accompanying presentation slides are made available by Catherine Paverd, under a Creative Commons licence: Attribution, Non Commercial, No Derivatives. Individual resources are subject to their own licensing conditions as listed. Screenshots in this document are copyright of The Mathworks. The Oxford University logo and crest is copyright of Oxford University and may only be used by Oxford University members in accordance with the University's branding guidelines.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What You Should Already Know . . . . .	1
1.2	What you will learn . . . . .	1
1.3	Where can I get a copy of MATLAB? . . . . .	2
<b>2</b>	<b>Fundamentals - How to interact with computers</b>	<b>3</b>
2.1	Basic Concepts . . . . .	3
2.2	The MATLAB Environment . . . . .	3
2.2.1	Overview . . . . .	4
2.2.2	Desktop Tools and Development Environment . . . . .	5
2.2.3	The Command Window . . . . .	6
2.2.4	Understanding File Locations . . . . .	8
2.2.5	The MATLAB Path . . . . .	8
2.2.6	The MATLAB Editor . . . . .	9
<b>3</b>	<b>Data Types - How to store different types of information</b>	<b>13</b>
3.1	Numeric Classes . . . . .	13
3.2	Characters and Strings . . . . .	14
3.3	The Logical Class . . . . .	15
<b>4</b>	<b>Matrices - How stored data is organised for processing</b>	<b>17</b>
4.1	Matrix Fundamentals . . . . .	17
4.2	Matrix Creation . . . . .	18
4.3	Matrix Concatenation . . . . .	19
4.4	Matrix Indexing . . . . .	20
4.5	Matrix Information . . . . .	22
4.6	Matrix Resizing . . . . .	23
4.7	Matrix Reshaping and Shifting . . . . .	24
4.8	Matrix Sorting . . . . .	25
<b>5</b>	<b>Operators and Control - How to tell a computer what to do</b>	<b>29</b>
5.1	Arithmetic Operators . . . . .	29
5.2	Relational Operators . . . . .	30
5.3	Logical Operators . . . . .	30
5.4	Conditionals . . . . .	31
5.5	Loops . . . . .	33
5.6	Return and Keyboard . . . . .	35
<b>6</b>	<b>Programming - How to organise your code</b>	<b>39</b>
6.1	Scripts . . . . .	39
6.2	Functions . . . . .	39
6.3	Script Components . . . . .	40
6.3.1	Comments . . . . .	40
6.3.2	Housekeeping Code . . . . .	40
6.3.3	Script Body - drawing a circle . . . . .	40
6.4	Function Components . . . . .	41
6.4.1	Function Declaration . . . . .	41
6.4.2	Help Comment Block . . . . .	41
6.4.3	Function Body . . . . .	41

6.5	Modular Programming . . . . .	42
6.6	Toolboxes . . . . .	43
<b>7</b>	<b>Error Handling - How do deal with things that go wrong</b>	<b>47</b>
7.1	Errors . . . . .	47
7.1.1	Typographical Errors . . . . .	47
7.1.2	Syntax Errors . . . . .	47
7.1.3	Array Indexing and Assignment Errors . . . . .	47
7.1.4	Algorithmic Errors . . . . .	48
7.2	Debugging . . . . .	48
7.2.1	Debugging in the MATLAB editor . . . . .	48
7.2.2	Debugging in the MATLAB command prompt . . . . .	50
<b>8</b>	<b>Graphics 1: Figures, Axes and Graphs</b>	<b>52</b>
8.1	Figures, Axes and Graphs . . . . .	52
8.2	Setting up Figures . . . . .	53
8.2.1	figure . . . . .	53
8.2.2	subplot . . . . .	53
8.2.3	axes and axis . . . . .	55
8.3	GUI Tools . . . . .	56
8.4	Plotting Functions . . . . .	59
8.4.1	Line Graphs: plot . . . . .	61
8.4.2	Bar Graphs: bar and barh . . . . .	63
<b>9</b>	<b>Graphics 2: Objects and Images</b>	<b>65</b>
9.1	Objects, Handles and Properties . . . . .	65
9.2	Working with Images . . . . .	67
9.2.1	Displaying Images . . . . .	67
9.3	Printing and Exporting . . . . .	68
<b>10</b>	<b>File Handling - How to handle internal and external files and data</b>	<b>72</b>
10.1	MAT-Files . . . . .	72
10.1.1	GUI Import and Export . . . . .	73
10.2	Excel Files . . . . .	74
10.3	Text (ASCII) Files . . . . .	75
10.4	Binary Files . . . . .	76
<b>11</b>	<b>Performance - How to determine how efficient your code is</b>	<b>81</b>
11.1	Measuring Performance . . . . .	81
11.1.1	Stopwatch timing . . . . .	81
11.1.2	The Profiler . . . . .	81
11.2	Improving Performance . . . . .	83
11.2.1	Preallocation . . . . .	84
11.2.2	Vectorization . . . . .	85
<b>12</b>	<b>What Next?</b>	<b>86</b>
12.1	Computer . . . . .	86
12.2	IT Services Help Centre . . . . .	86
12.3	Books . . . . .	86

## List of Exercises

1	Getting Started with the MATLAB Environment . . . . .	11
2	Setting the MATLAB Path . . . . .	12
3	Exploring MATLAB Data Types . . . . .	16
4	Introduction to Matrices . . . . .	26
5	Magic Matrices . . . . .	27
6	Control Statements - Basic Concepts . . . . .	36
7	Fibonacci Numbers . . . . .	36
8	Roots of quadratic equation . . . . .	38
9	Function for roots of quadratic equation . . . . .	44
10	Noughts and Crosses . . . . .	45
11	Error Checking . . . . .	51
12	Interactive Plot Tools . . . . .	56
13	Data Exploration Tools . . . . .	57
14	Exploring Graphics Objects . . . . .	66
15	Printing and Exporting Graphics through the GUI . . . . .	70
16	Creating a Video . . . . .	71
17	Magic Files . . . . .	77
18	Demographic statistics of UK and European Union countries . . . . .	79
19	Measuring Performance . . . . .	82

# 1 Introduction

Welcome to the course MATLAB: A Comprehensive Introduction. This booklet accompanies the course delivered by Oxford University's IT Learning Centre. Although the exercises are clearly explained so that you can work through them yourselves, you will find that it will help if you attend the taught session where you can get advice from the teacher, demonstrator and each other.

If at any time you are not clear about any aspect of the course, please make sure you ask your teacher or demonstrator for some help. If you are away from the class, you can get help by email from your teacher or from [help@it.ox.ac.uk](mailto:help@it.ox.ac.uk).

## 1.1 What You Should Already Know

This course covers the fundamentals of data analysis with MATLAB. No prior knowledge of MATLAB is expected, however watching some introductory videos on Lynda.com is highly recommended. We will assume that you are familiar with opening files from particular folders and saving them, perhaps with a different name, back to the same or a different folder.

The computer network in our teaching rooms may differ slightly from that which you are used to in your College or Department; if you are confused by the differences ask for help from the teacher or demonstrator.

## 1.2 What you will learn

This series of 4 sessions will cover MATLAB core functionality as follows:

### Session 1: Getting Started

- Fundamentals - How to interact with computers
- Data Types - How to store different types of information
- Matrices - How stored data is organised for processing

### Session 2: Programming Basics

- Operators and Control - How to tell a computer what to do
- Programming - How to organise your code
- Error Handling - How to deal with things that go wrong

### Session 3: Working with Graphics

- Graphics 1 - How to work with Figures, Axes and Graphs
- Graphics 2 - How to work with Objects and Images

### Session 4: Further Programming

- File Handling - How to handle internal and external files and data
- Performance - How to measure how program efficiency



### **1.3 Where can I get a copy of MATLAB?**

Oxford University runs a software licence server that provides access to concurrent licences to the Matlab software for any University department or college. For more details please refer to <http://www.eng.ox.ac.uk/~labejp/TAH/matlabTAH.html>

## 2 Fundamentals - How to interact with computers

### Analogy: The Language of Computers

*Let's assume you only speak English, but you need to email some instructions to a colleague in French. To do this, you could write the email in English, send it to a translator to translate it into French, and then send it to your colleague. Your colleague will then be able to understand your instructions and execute them.*

*In a similar way, programmers write programs in a programming language (like MATLAB), but a computer can only understand computer language (machine code). So we need to write the program, send it to a Compiler or an Interpreter to translate it, and then give it to the computer to execute.*

### 2.1 Basic Concepts

At the fundamental level, a computer is a collection of small components that have either a high (1) or a low (0) state. That is to say, a computer can understand instructions given by a series of 1's and 0's, called 'machine code'. Now, if one were to write 'Welcome!' in machine code it would look like this: '01010111 01100101 01101100 01100011 01101111 01101101 01100101 00100001'. So you can image how long it would take to write an entire program. To avoid this we use programming languages like MATLAB, C, Python and others. These languages allow us to give a computer instructions easily, for example by writing `sum` in order to add two numbers. Languages that are relatively close to machine code are known as low-level languages, such as Assembly. On the other hand, high-level languages such as Python and MATLAB are very abstracted from machine code, and are thus more user friendly.

Normally we write programming instructions in a text editor. We then use a Compiler or an Interpreter to translate what we have written into something a computer can understand. This process is shown in figure 2.1. Usually, one is able to write code, compile/ interpret it, and execute it all within one program, or environment. Some development environments will offer more powerful functionality, such as multiple browsers (to see different parts of the code you are working on), or a debugger (to quickly identify errors in the code). The MATLAB Environment will be discussed in greater detail in the next section.

### 2.2 The MATLAB Environment

MATLAB (*MATrix LABoratory*) is a high-level programming language with an interactive environment, used for algorithm development, data visualization, data analysis, and numeric computation. Core strengths of the system include rapid development, powerful built-in functionality and extensive application-specific functionality provided by both official toolboxes and user-contributed code.

MATLAB includes extensive documentation, which can be accessed through both the program help menu and the web. For example, the official Getting Started

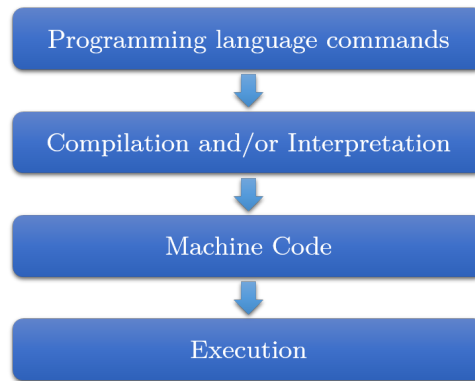


Figure 2.1: The stages of program execution.

guide is here: **Help►Matlab►Getting Started**. In this section, we will introduce the MATLAB environment and learn how to enter commands, get help, and save and load sessions.

### 2.2.1 Overview

MATLAB consists of the language itself, the development environment and multiple powerful libraries, some of which are discussed in more detail here:

- **The Language** - The MATLAB language is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both “programming in the small” to rapidly create quick programs you do not intend to reuse. You can also do “programming in the large” to create complex application programs intended for reuse.
- **Desktop Tools and Development Environment** - This part of MATLAB is the set of tools and facilities that help you use and become more productive with MATLAB functions and files. Many of these tools are Graphical User Interfaces (GUIs). It includes: the MATLAB desktop and Command Window, a text editor, a debugger, a code analyzer, and browsers for viewing help, the Workspace, and folders.
- **Mathematical Function Library** - This library is a vast collection of computational algorithms ranging from elementary functions, like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, and fast Fourier transforms.
- **Graphics Libraries** - MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete GUIs on your custom built MATLAB applications.
- **External Interfaces Library** - This library allows you to write C and

Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), for calling MATLAB as a computational engine, and for reading and writing MAT-files.

See also **Product Overview**

## 2.2.2 Desktop Tools and Development Environment

The first time you start MATLAB, the desktop appears with the following layout (I have run a few commands for illustrative purposes):

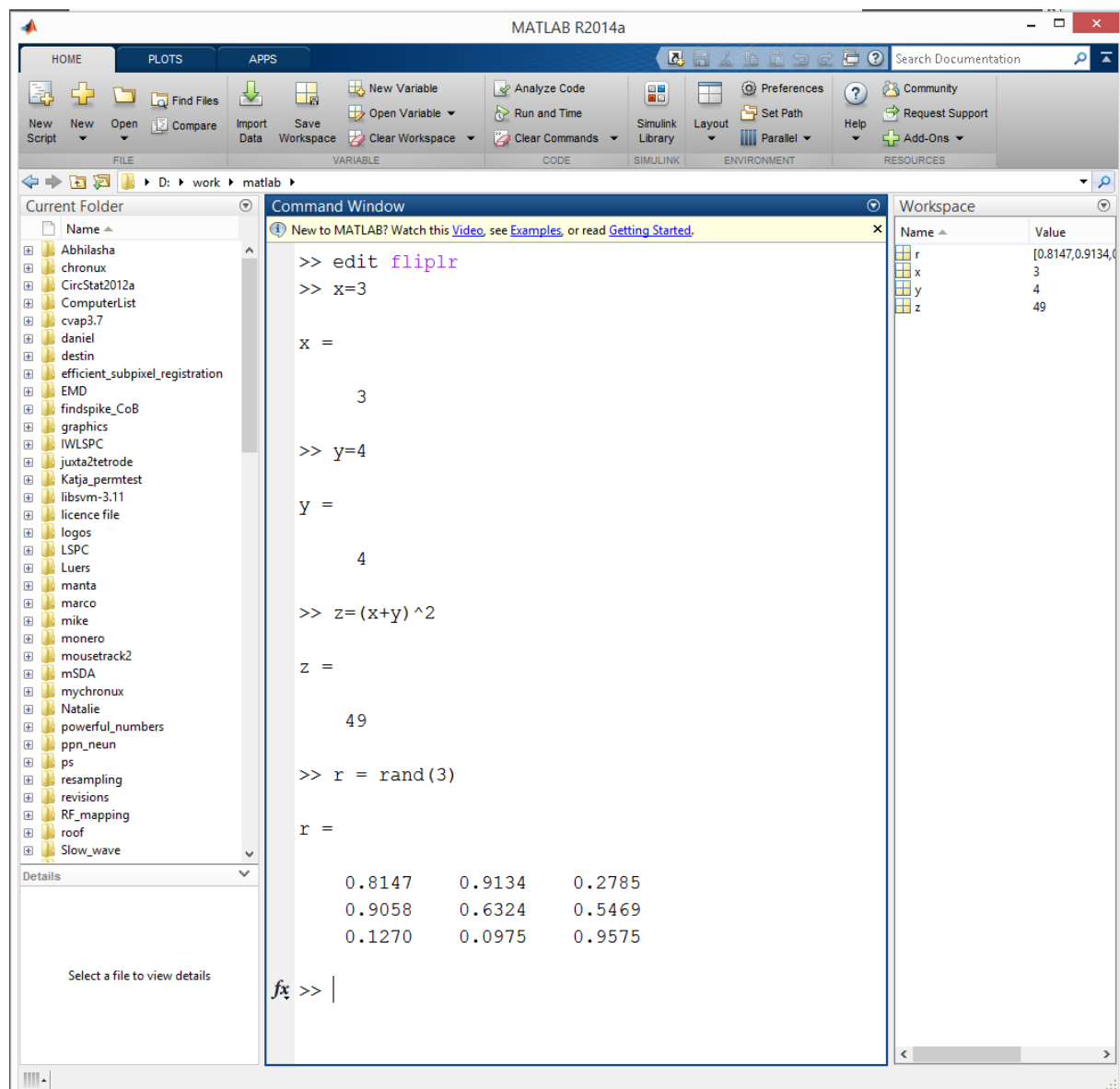


Figure 2.2: The MATLAB Desktop: a Java GUI with several components.

The layout can be altered using the desktop Home tab Layout button:



See also the **Desktop Overview**

The MATLAB Desktop is a Java GUI and, as such, should appear and behave in a similar fashion on different operating systems. The two most important Desktop Tools are the Command Window, which usually sits in the centre of the Desktop, and the Editor, which usually occupies a separate window (though it may be docked). In this session, we will introduce both.

### 2.2.3 The Command Window

The command window is used to create variables, enter commands and run programs interactively. A command entered here is immediately processed by MATLAB to produce a response, before the system returns a new prompt for further commands. For example, if we enter 3 at the command prompt (type 3 and press Enter):

```
3
```

then the system returns:

```
ans =  
3
```

Our command has created a variable called **ans** and assigned it the value 3. **ans** is a special, generic variable short for ‘answer’. It is used and overwritten whenever a statement returns a value that is not explicitly assigned to a variable.

MATLAB can perform basic arithmetic like a calculator. Entering

```
3+4
```

at the command prompt returns the value 7 in **ans**, overwriting the previous value. In MATLAB, + is one of a family of *arithmetic operators* (see section 5).

#### Getting Help

For help on commands and operators you can enter **help** followed by the command, or operator name. e.g.

```
help fliplr
```

Further information is sometimes available using **doc** in the same way.

```
doc fliplr
```

Each time MATLAB outputs to **ans**, the old value is lost. For the system to remember more than one thing at a time, we need to define named variables:

```
x = 3+4
```

```
y = 3*4
```

### Getting Workspace Information

To see the current value of a single variable, you can enter the variable name at the command prompt. e.g.

```
x
```

For information on all currently defined variables, you can use `who` and `whos`

```
who
```

```
whos
```

Alternatively, look at the Workspace Browser in the GUI (top right by default).

### Command Window Output

As noted above, variable values are displayed in the Command Window when the variable name is entered. However, adding a semi-colon after a variable name, or expression, suppresses the usual output. For example, typing `x` and pressing ENTER will cause MATLAB to display the variable name and value. However, adding a semicolon after the variable (for example typing `x;`) will suppress the output.

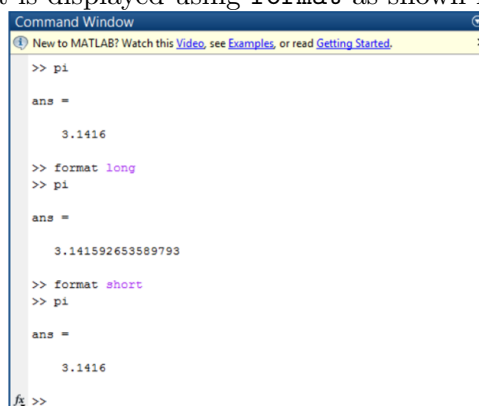
To save space in program output, we can use `disp` to display variable values without printing the array name. A semi-colon does not suppress the output from `disp`.

### Mathematical Display Format

The `format` command can be used to control the output format of numeric values displayed in the Command Window. This does not affect computation. To view the options, use:

```
help format
```

For example, MATLAB has `pi` as a pre-defined as a mathematical constant, but we can change how it is displayed using `format` as shown in the figure below:



```

Command Window
New to MATLAB? Watch this Video, see Examples, or read Getting Started.
>> pi
ans =
    3.1416

>> format long
>> pi
ans =
    3.141592653589793

>> format short
>> pi
ans =
    3.1416

fx >>
  
```

The `clc` command clears the Command Windows, while the `home` function sends the cursor home (to the top left of the Command Window). These look superficially similar but the Command Window history is still visible by scrolling up for `home` but not `clc`. Paged output can be obtained using `more`.

## Accessing the Command History

Previous commands can be accessed in the Command Window by dragging and dropping a line from the GUI Command History Browser (or double-clicking to run immediately). Note, the Command History Browser may not be docked in the Desktop default layout, but this can be changed using the Layout button.

Alternatively, you can press the up arrow key to move backwards through entered commands. In addition, partial matches can be used to find a particular line by typing in the first few letters before pressing the up arrow key. Try typing `x` at the prompt (don't press enter), and then press the up arrow several times. Note that a previous command selected in this way can be edited before entering again.

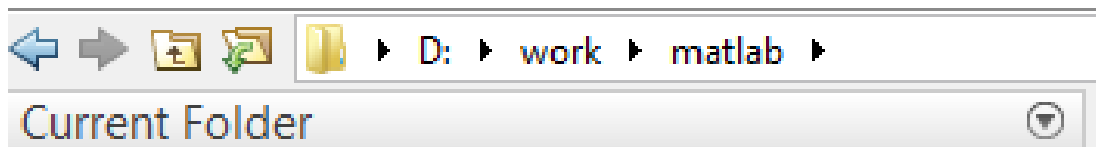
### 2.2.4 Understanding File Locations

There are several important locations defined in the MATLAB system.

`matlabroot` is the folder where MATLAB is installed. Important, modifiable locations include the **current** and **startup** folders.

The **current folder** is a reference location that MATLAB uses to find files. This folder is sometimes referred to as the current directory, current working folder, or present working directory. It is not the same location as the operating system current folder. You can view and change the current folder in two main ways:

- In the Command Window, use the `cd` or `pwd` commands
- On the Desktop, use the tools just above the Current Folder Browser



The **startup folder** is the current folder in the MATLAB application when it starts. On Windows and Apple Macintosh platforms, a folder called `userpath` is added automatically to the search path (see below) upon startup, and is the default startup folder. On Linux platforms, you can set the `userpath` as the startup folder. You can view and edit the current user path by running `userpath`.

### 2.2.5 The MATLAB Path

For performance reasons, MATLAB limits where it looks for files. Files are only accessible if they are in either the current folder, or the search path. The **path** is a list of folder locations where MATLAB will search for program and data files. To see the current path:

```
path
```

In Windows, with a typical MATLAB installation, this returns something like:

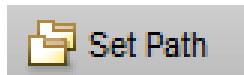
## MATLABPATH

```

H:\My Documents\MATLAB
H:\Program Files\MATLAB\R200nn\toolbox\matlab\general
H:\Program Files\MATLAB\R200nn\toolbox\matlab\ops
H:\Program Files\MATLAB\R200nn\toolbox\matlab\lang
H:\Program Files\MATLAB\R200nn\toolbox\matlab\elmat
H:\Program Files\MATLAB\R200nn\toolbox\matlab\elfun
...

```

To edit the path, you can use `addpath`, `rmpath`, `genpath`, `pathtool` and `savepath`. Alternatively, use the Set Path button on the Desktop Home tab (next to Layout):



## 2.2.6 The MATLAB Editor

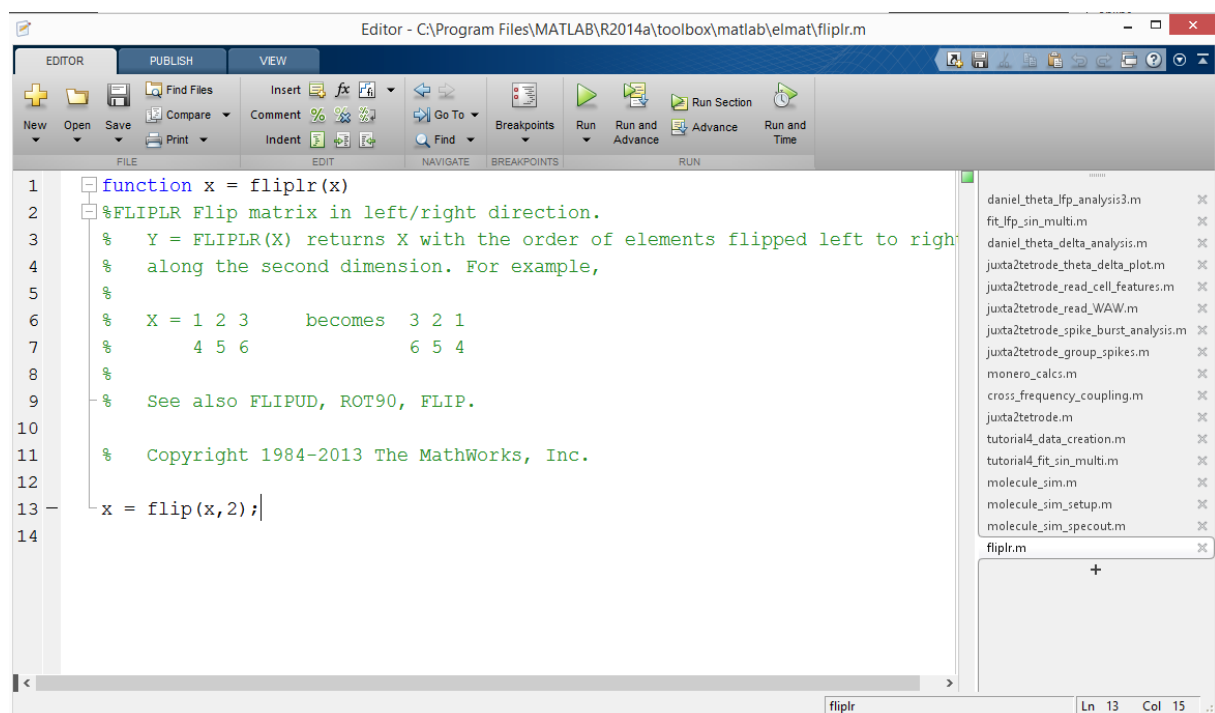


Figure 2.3: The MATLAB Editor

The MATLAB Editor is used to view and edit MATLAB program files. Normal MATLAB programs are text files with a `.m` extension and are often called *M-files*. The Editor starts automatically when you create or open an M-file and provides *code highlighting* and *debugging* features.

The simplest kind of MATLAB program is called a **script**. Scripts are simply a list of commands stored in an M-file. When you run a script, the result is exactly the same as if the commands were entered one-by-one in the command window.



To open a new (blank) script, you can use the key combination `CTRL+N`, or click the New button in the editor, or the New Script button on the desktop Home tab.



To open an existing file, you can use the `edit` or `open` commands, the key combination `CTRL+O`, or click on the Open button in the editor or desktop Home tab.



**Running scripts** There are two main ways to run a script in MATLAB.

1. From the Command Window, you can enter the script name.
2. From the Editor, you can press F5, or click the Run button (which features a green 'play' triangle).

<b>Exercise 1. Getting Started with the MATLAB Environment</b> <ul style="list-style-type: none"> <li>• Start MATLAB</li> <li>• Change the Current Folder</li> <li>• Open the PDF Handbook</li> <li>• Explore the Help System</li> </ul>	
Key Functions	<code>help</code> , <code>doc</code> , <code>pwd</code> , <code>format</code>
<b>Task 1</b> Start MATLAB	<b>Step 1</b> Find MATLAB in <b>Start Menu All Programs</b> , and click on the program file (Matlab.exe).
	<b>Step 2</b> Note each of the components introduced above in the MATLAB desktop. Test docking and undocking the Command History, using the Layout button.
<b>Task 2</b> Change the Current Folder	<b>Step 1</b> To view the current folder enter <code>pwd</code> in the Command Window.
	<b>Step 2</b> The course files will be distributed to your computer. For MATLAB to operate on these files, change the Current Folder to the folder in which you find the files (for example, H:\MATLAB), as described in section 2.2.4.
<b>Task 3</b> Open the PDF Handbook	<b>Step 1</b> The PDF Handbook features internal links and links to online documentation. It has also been designed so that you can copy and paste some example commands.
	<b>Step 2</b> On your system (outside MATLAB), open the course PDF now.
	<b>Step 3</b> Browse to section 2.2.3 in the PDF and write the code example from the Mathematical Display Format box into the MATLAB Command Window. Press Enter in the Command Window after each line to see the result.
<b>Task 4</b> Explore the Help System	<b>Step 1</b> Read the Getting Help box in section 2.2.3, and run the example commands.
	<b>Step 2</b> Run <code>help</code> and <code>doc</code> on several of the commands introduced in this section: e.g. <code>help</code> , <code>doc</code> , <code>whos</code> , <code>disp</code> , <code>format</code> , <code>clc</code> , <code>home</code> , <code>cd</code> , <code>pwd</code> .
	<b>Step 3</b> In the Help Browser (opened using <code>doc</code> ), Enter <code>path</code> in the Help Browser search box (top). Look at a few of the entries listed, navigating using the back button (top left).

<b>Exercise 2. Setting the MATLAB Path</b> <ul style="list-style-type: none"> <li>• Update the Path</li> <li>• Create a startup file</li> </ul>	
Key Functions	<code>help</code> , <code>doc</code> , <code>path</code> , <code>addpath</code> , <code>genpath</code> , <code>userpath</code>
<b>Task 1</b> Update the Path	<b>Step 1</b> To view the path, enter <code>path</code> in the Command Window. Hint: scroll up to view all the entries.
	<b>Step 2</b> To add the <b>matlab</b> folder to the MATLAB path, run <code>addpath('insert the name of the path where your Matlab files are stored here')</code> , for example <code>addpath('H:\matlab')</code> . Enter <code>path</code> to view the changes.
	<b>Step 3</b> To generate path locations for all subfolders within the <b>matlab</b> folder, run <code>genpath('H:\matlab')</code> . Use <code>doc addpath</code> and <code>doc genpath</code> to find out more.
	<b>Step 4</b> Combining steps 2 and 3, add the whole matlab folder tree to the path by running <code>addpath(genpath('H:\matlab'))</code> . You can use the up arrow key to modify your previous command (see section 2.2.3). Run <code>path</code> to view the changes.
<b>Task 2</b> Create a startup file	<b>Step 1</b> A <code>startup.m</code> file is a user-created script that runs when MATLAB starts. It should normally go in the <code>userpath</code> location. Run <code>doc startup</code> for details.
	<b>Step 2</b> To begin creating a startup file, open a new script in the Editor by clicking the New Script button on the Desktop Home tab,  , and undock if docked: 
	<b>Step 3</b> Add the path-updating command from Task 1, Step 4 into the script (you can use copy and paste).
	<b>Step 4</b> Add a second line to your script to update the current folder, using <code>cd('H:\matlab')</code> .
	<b>Step 5</b> In the Command Window, run <code>userpath</code> . Note the path.
	<b>Step 6</b> In the Editor, click the save button and save your script in the <code>userpath</code> location, with file name <code>startup.m</code> .
	<b>Step 7</b> Exit MATLAB and restart. Note the Current Folder and updated path after startup.

### 3 Data Types - How to store different types of information

#### Analogy: Data Types

*Humans tend to focus on different things at different times, for example, on numbers in a maths exam and letters in a language exam. In the same way, computers also have different data types for storing information, but you need to specify which data type you want the computer to use.*

There are many different data types, or classes, supported in MATLAB. For an overview, see ► **Fundamental MATLAB Classes**. Each data type has specific characteristics that make it useful for storing specific information. For example, to store numbers you would use a numeric class, but to store letters or words you would use the character (char) class.

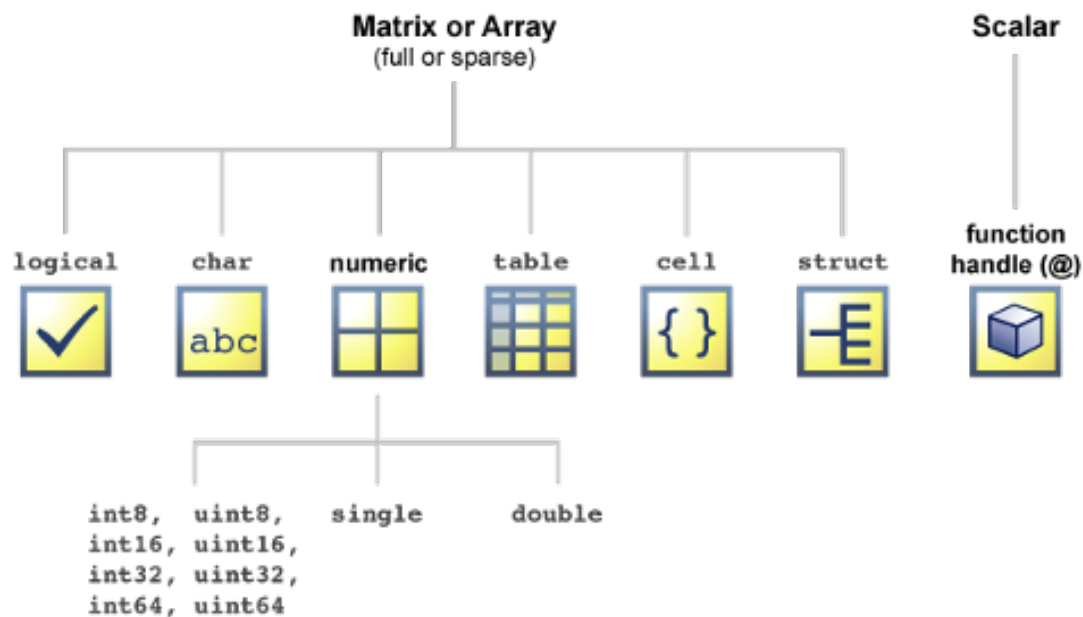


Figure 3.1: Fundamental MATLAB Data Types

#### 3.1 Numeric Classes

The default type for **numerical data** is double precision floating point (or double). A double occupies 64-bits of memory. Single precision floats (singles) are supported in more recent versions of MATLAB, and they occupy 32-bits at the cost of lower precision. Signed and unsigned integer types are also defined.

**Converting between types**

You can convert between data types using the functions described in:

```
doc datatypes
```

For example:

```
format long
```

```
x = pi
```

```
y = single(x)
```

```
z = uint8(x)
```

```
whos
```

As mentioned before, MATLAB also has a powerful set of functions that perform operations on stored data. However, not all functions work for all data types. For example, `sin` is a built in MATLAB trigonometric function which is not defined for integer types.

## 3.2 Characters and Strings

Text data are represented in MATLAB using the `char` or **character** data type. A character in MATLAB is actually an integer value converted to its Unicode UTF-16 character equivalent (see `char` for details).

A string is a vector of characters. String variables are normally surrounded by single quotes, and highlighted in pink.

```
name = ['Alan ' 'M. ' 'Turing']
```

Strings are useful when working with and displaying text data, displaying warning and error messages during program execution, and naming variables and files.

Strings are also commonly used to provide *categorical* inputs to MATLAB commands (functions). See the Function and Command Syntax box in section 10.1 for further details.

`num2str` is a function that converts numbers to character or string values. This can be useful for labelling figures, and working with numbered files. Here is an example of how `num2str` can be combined with matrix concatenation (recall section 4.3) to reference a numbered file:

```
filename = ['DataFile',num2str(400)]
```

```
filename =  
DataFile400
```

`str2num` (or `str2double`) converts strings to numeric values, which can be useful when data has been imported from an ASCII file for example.

### 3.3 The Logical Class

The `logical` data type represents a logical true or false state as 1 and 0, respectively.

A **logical expression** is a piece of matlab code that returns a logical value.

#### Using Logical Data

As introduced in section 4.4, logical values can be used in MATLAB for *matrix indexing*. Logical values are also used in *conditional statements*. We will look at these in more detail in section 5, but here is a simple example:

```
r = '2'

if ischar(r)
    circumference = 2*pi*str2num(r)
else
    circumference = 2*pi*r
end
```

<b>Exercise 3. Exploring MATLAB Data Types</b> <ul style="list-style-type: none"> <li>• Create a random whole number between 1 and 10</li> <li>• Convert the number to integer and string types</li> <li>• Form a file name string and save workspace to file</li> </ul>	
Key Functions	<code>rand</code> , <code>*</code> , <code>round</code> , <code>fix</code> , <code>floor</code> , <code>ceil</code> , <code>num2str</code> , <code>save</code>
<b>Task 1</b> Get help on the <code>rand</code> function	<b>Step 1</b> Enter <code>doc rand</code> in the Command Window. Note the Description for the scalar form with no inputs.
<b>Task 2</b> Create a random whole number between 1 and 10	<b>Step 1</b> Open a new script and add a line to run <code>rand</code> and assign the output to variable <code>x</code> (i.e. <code>x=rand</code> ). Save and run the script (don't call it <code>rand</code> !).
	<b>Step 2</b> <code>rand</code> produces floating point numbers between 0 and 1. To get a value between 1 and 10, you need to multiply by 10 (enter <code>help *</code> for details). Modify the previous command and run the script again.
	<b>Step 3</b> To get an integer (whole number) value, you need to apply rounding. There are four rounding functions in MATLAB: <code>round</code> , <code>fix</code> , <code>floor</code> and <code>ceil</code> . Remember that a zero value is not allowed (this should help you choose the rounding function). Modify the previous command and run the script a few times to confirm that <code>x</code> is a whole number between 1 and 10.
<b>Task 3</b> Convert the number to integer and string types	<b>Step 1</b> In the Workspace Browser (top right of desktop), right click Name and ensure Bytes and Class are selected.
	<b>Step 2</b> In your script, use the integer conversion functions detailed in <code>doc datatypes</code> to convert the random number into different integer data types. Each time use an expression like <code>x = func(x)</code> (e.g. <code>x = int8(x)</code> ) to overwrite <code>x</code> . View the changes in the Workspace Browser.
	<b>Step 3</b> use <code>num2str</code> to convert <code>x</code> into a string, assigning the response to a new variable <code>s</code> ( <code>s = num2str(x)</code> ).
<b>Task 4</b> Form a file name string and save workspace to file	<b>Step 1</b> Type <code>help strcat</code> and read the documentation. Then create a string to store the word 'DataFile' and use <code>strcat</code> together with <code>num2str</code> to form a file name of the form 'DataFileN', where N is the random number. The variable name should be <code>filename</code> .
	<b>Step 2</b> Use <code>save(filename)</code> to save your workspace to the named file. The saved file should appear in your Current Folder browser. Double click to view the contents.

## 4 Matrices - How stored data is organised for processing

### 4.1 Matrix Fundamentals

- All of the datatypes mentioned in 3 can be stored in an ordered configuration known as a matrix. All data in MATLAB is stored in matrices of different sizes for ease of accessibility.
- The elements of a matrix must all be of the same data type (see section 3) (numeric, character, logical etc.), but here we will focus on numeric matrices.
- MATLAB uses two-dimensional matrices (where the data is stored in rows and columns) to store both single elements and series of elements. If a matrix has  $N$  rows and  $M$  columns, it is called an  $N \times M$  matrix. For numeric data, special meaning is sometimes attached to 1-by-1 matrices, which are called *scalars*, and to matrices with only one row or column, which are *vectors*.
- MATLAB also supports data structures that have more than two dimensions. These data structures are referred to as multidimensional *arrays* (for example, 2D, 3D or 4D arrays) in the MATLAB documentation, and indeed the matrix is really a special case of an n-dimension array, with  $n=2$  (see section 3). In this book, we will focus on two-dimensional matrices.

#### Analogy: The Matrix Structure

*You could think of a two-dimensional matrix in MATLAB like a large table of data, similar to what you see in a Microsoft Excel spreadsheet. Essentially, each individual piece of information has a unique position, and can be accessed by using specific row and column values.*



## 4.2 Matrix Creation

There are many ways to create a matrix in MATLAB. In section 2.2.3, the code examples with defined variables created simple matrices to store scalar numeric data. A simple way to create two-dimensional matrices is to use the matrix constructor operator, `[ ]`. For example,

```
A = [6 3 2 8; 5 1 3 7; 1, 6, 7, 2;4,5,4,1]
```

```
A =
     6     3     2     8
     5     1     3     7
     1     6     7     2
     4     5     4     1
```

Each row of the matrix is terminated with a semi-colon, while the elements within each row are separated by either commas or spaces. Spaces matter when dealing with signed numeric data, and commas are less error-prone. Compare:

```
[7 -2 +5],    [7 - 2 + 5],    [7, - 2, +5]
```

Because matrices are rectangular, all rows in a matrix must have the same number of elements and MATLAB will return an error if this condition is violated.

```
B = [6 3 2 8; 5 1 3]
```

In addition to the matrix constructor operator, MATLAB also has a number of built-in functions (commands) for creating particular kinds of matrix. Frequently-used examples include the following.

<code>ones</code>	Create a matrix (or array) of all ones
<code>zeros</code>	Create a matrix of all zeros
<code>rand</code>	Create a matrix of uniformly distributed random numbers
<code>randn</code>	Create a matrix of normally distributed random numbers

**Generating Numeric Sequences: The Colon Operator**

Numeric sequences are useful for indexing and they can be created in MATLAB using the `colon` operator. The colon operator (`first:last`) generates a vector of sequential numbers from the `first` value to the `last`. The default sequence is made up of incremental values, each 1 greater than the previous one, even if the end value is not an integral distance from the start:

```
D = 4:7, D = 4:7.5
```

```
D =  
    4 5 6 7
```

The numeric sequence does not have to be made up of positive integers. It can include negative numbers and fractional numbers as well:

```
D = -1.5:1.5
```

```
D =  
-1.5000 -0.5000 0.5000 1.5000
```

If the `last` value is lower than the `first`, MATLAB will return an empty matrix:

```
D = 7:4
```

```
D =  
Empty matrix: 1-by-0
```

For a sequence with non-default stepping, use (`first:step:last`), where `step` can be any real value (positive or negative):

```
D = 4:.5:7
```

```
D =  
    4.0000  4.5000  5.0000  5.5000  6.0000  6.5000  7.0000
```

or

```
D = 7:-.5:4
```

```
D =  
    7.0000  6.5000  6.0000  5.5000  5.0000  4.5000  4.0000
```

### 4.3 Matrix Concatenation

Matrix concatenation is the process of joining one or more matrices to make a new matrix. The brackets `[ ]` operator discussed above serves not only as a matrix constructor, but also as the concatenation operator. The expressions `C = [A B]` or `C = [A,B]` horizontally concatenates matrices A and B. The expression `C = [A;B]` vertically concatenates them. E.g.

```
E = [A;C]
```

Caution: Horizontal concatenation requires an equal number of rows, and vertical concatenation requires an equal number of columns.

An alternative to using the `[ ]` operator for concatenation are the three functions `cat`, `horzcat`, and `vertcat`. For example, the following commands give the same result as `E = [A;C]`:

```
E = cat(1,A,C)
```

```
E = vertcat(A,C)
```

Another useful concatenation function is `repmat`, which creates a matrix composed of tiled copies of a smaller matrix. For example,

```
F = [1,2,3;4,5,6];
```

```
G = repmat(F,2,3)
```

replicates the matrix `F` two times vertically and 3 times horizontally.

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

## 4.4 Matrix Indexing

To reference a single element in a matrix, you can use either *row-column* indexing, `A(row,column)`, or *linear* indexing `A(ind)`, where matrix elements are counted downward through successive columns.

```
A =
    6  3  2  8
    5  1  3  7
    1  6  7  2
    4  5  4  1
```

```
A(3,2), A(7)
```

```
ans =
    6
```

To convert between index styles, use `sub2ind` and `ind2sub`, where *sub* refers to row-column indexing. E.g.

```
linearindex = sub2ind(size(A),3,2)
```

```
linearindex =
    7
```

**Accessing Sequential Elements: The Colon Operator Returns**

With either row-column or linear indexing, a single index can be replaced by an *integer-valued* numeric sequence to specify multiple elements.

```
A(2,1:3)
```

```
ans =  
    5 1 3
```

```
A(1:3:16)
```

```
ans =  
    6 4 6 3 8 1
```

```
A(2:3,1:3)
```

```
ans =  
    5 1 3  
    1 6 7
```

In indexing, the keyword **end** designates the last element in a particular dimension.

```
A(2,2:end)
```

```
ans =  
    1 3 7
```

The colon by itself refers to *all* elements.

```
A(2,:)
```

```
ans =  
    5 1 3 7
```

```
A(:)
```

```
ans =  
    6  
    5  
    1  
    4  
    3  
    ...
```

For non-sequential access, an integer-valued matrix can also be used for indexing.

```
A([1,4,9])
```

```
ans =  
    6 4 2
```

Logical true and false states are represented in MATLAB as 1 and 0, respectively (see section 3). Logical values can be used for matrix indexing in MATLAB and this is an efficient and general method for selecting arbitrary matrix elements. A logical indexing matrix is normally the same size as the matrix being accessed and the indexing operation here is based on the position of true values in the indexing matrix. In the following example, we use logical indexing to blank out the smaller values in A.

```

L = A<5

L =
    0 1 1 0
    0 1 1 0
    1 0 0 1
    1 0 1 1

A(L) = 0

A =
    6 0 0 8
    5 0 0 7
    0 6 7 0
    0 5 0 0

```

## 4.5 Matrix Information

Various built-in functions in MATLAB reveal information about matrices. The following functions return information regarding matrix size and shape:

<code>size</code>	The length of each dimension.
<code>length</code>	The length of the longest dimension.
<code>numel</code>	The total number of elements.

Here are a few examples:

```

size(F)

ans =
    2 3

length(F)

ans =
    3

numel(F)

ans =
    6

```

Size and length are particularly useful and frequently used functions.

Certain functions test the data type of a matrix, returning a logical value to indicate the result (see section 3). These include the generic `isa` function and specific functions such as `isfloat`, `isinteger`, `isnumeric`, and `islogical`.

Other functions test the structure of a matrix, such as `isvector`, `isscalar` and `isempty`. These function are useful for avoiding errors caused by special cases. For example, say you have a program designed to perform an operation on paired data stored in a 2-by- $N$  matrix, where  $N$  may vary. In the general case, with  $N > 1$ , we

could use `length` to determine the number of data pairs, but this approach may fail for  $N = 1$  (where `length` is 2), and  $N = 0$  (where `length` is 0). Calls to `isvector` and `isempty` could check for these special cases.

## 4.6 Matrix Resizing

Matrices in MATLAB can be dynamically resized provided the resulting matrix remains rectangular.

Matrix expansion can be achieved via concatenation as introduced above. Alternatively, it is possible to write to a location outside the current bounds of a matrix. In this case, MATLAB automatically pads it with zeros where a row or column is not completely specified. For example:

```
F = [1,2,3;4,5,6];
```

```
F(1,5) = 3
```

```
F =
    1  2  3  0  3
    4  5  6  0  0
```

You can add a block of numbers in the same way:

```
F = [1,2,3;4,5,6];
```

```
F(1:2,5:6) = 8
```

```
F =
    1  2  3  0  8  8
    4  5  6  0  8  8
```

Conversely, you can delete rows and columns from a matrix by assigning the empty array `[]` to those rows or columns.

```
F(:,6) = []
```

```
F =
    1  2  3  0  8
    4  5  6  0  8
```

## 4.7 Matrix Reshaping and Shifting

The following functions modify matrix shape or ordering

<code>flipud</code>	Flip matrix in up/down direction.
<code>fliplr</code>	Flip matrix in left/right direction.
<code>flipdim</code>	Flip matrix in the specified direction (1 for up/down).
<code>rot90</code>	Rotate matrix anti-clockwise by 90 degrees.
<code>transpose</code>	Flip matrix about its main diagonal, turning row vectors into column vectors and vice versa. This is equivalent to the <code>'</code> operator (for more on operators see section 5).
<code>reshape</code>	Modify the shape of a matrix. <code>B = reshape(A,m,n)</code> returns the m-by-n matrix B whose elements are taken column-wise from A. An error results if A does not have m*n elements.
<code>circshift</code>	Circularly shift matrix contents.

Here are a few illustrative examples:

```
F = [1,2,3;4,5,6]
```

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

```
fliplr(F)
```

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

```
transpose(F)
```

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

```
reshape(F,3,2)
```

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

```
circshift(F,[0,1])
```

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

## 4.8 Matrix Sorting

The MATLAB `sort` function sorts matrix elements along a specified dimension, using `sort(A,1)` to sort along columns and `sort(A,2)` to sort along rows. As with many built-in functions, omitting the specified dimension causes the function to operate column-wise, i.e. `sort(A)` is equivalent to `sort(A,1)`.

```
A = [6 3 2 8; 5 1 3 7; 1, 6, 7, 2;4,5,4,1]
```

```
A =
     6     3     2     8
     5     1     3     7
     1     6     7     2
     4     5     4     1
```

```
sort(A)
```

```
ans =
     1     1     2     1
     4     3     3     2
     5     5     4     7
     6     6     7     8
```

```
sort(A,2)
```

```
ans =
     2     3     6     8
     1     3     5     7
     1     2     6     7
     1     4     4     5
```

By default, `sort` sorts in ascending order, but an additional argument can be used to specify descending-order sorting:

```
sort(A,2,'descend')
```

```
ans =
     8     6     3     2
     7     5     3     1
     7     6     2     1
     5     4     4     1
```

`sortrows` keeps elements of each row in its original order, but sorts the rows according to the order of the elements in a specified column (the first by default).

```
sortrows(A)
```

```
A =
     1     6     7     2
     4     5     4     1
     5     1     3     7
     6     3     2     8
```



<b>Exercise 4. Introduction to Matrices</b> <ul style="list-style-type: none"> <li>• Create a matrix</li> <li>• Matrix concatenation</li> <li>• Matrix information</li> <li>• Matrix indexing</li> <li>• Matrix resizing and sorting</li> </ul>	
Key Functions	[ ], zeros, ones, cat, horzcat, vertcat, length, size, numel, clc, transpose, reshape, sort
<b>Task 1</b> Create a matrix	<b>Step 1</b> Create a matrix A of 3 rows and 3 columns using the matrix constructor operator [ ], and assign random numbers to each element.
	<b>Step 2</b> Create a matrix B of 3 rows and 2 columns, in which all elements are all ones.
	<b>Step 3</b> Create a matrix C of 2 rows and 5 columns, in which all elements are all zeros.
<b>Task 2</b> Matrix concatenation	<b>Step 1</b> Concatenate matrices A and B horizontally into a new matrix D. Display the resulting matrix in the command window.
	<b>Step 2</b> Concatenate matrices C and D vertically into a new matrix E. Display the resulting matrix in the command window.
<b>Task 3</b> Matrix information	<b>Step 1</b> Using already available MATLAB functions identify the size of matrix E and make sure that the output is the expected one. Display the number of rows, columns and the total number of elements in the matrix.
<b>Task 4</b> Matrix indexing	<b>Step 1</b> Clear your command window using clc command. Display matrix E on your command window.
	<b>Step 2</b> Display the element in the 2 <sup>nd</sup> row and 3 <sup>rd</sup> column of matrix E. Make sure that the result is the expected one.
	<b>Step 3</b> Display all the elements in the 2 <sup>nd</sup> row.
	<b>Step 4</b> Display all the elements in the 3 <sup>rd</sup> column.
	<b>Step 5</b> Display the elements in the 2 <sup>nd</sup> row from the 3 <sup>rd</sup> column until the last column of matrix E.

<b>Task 5</b> Matrix resizing and sorting	<b>Step 1</b> Create a matrix <b>F</b> that is actually part of matrix <b>E</b> . More specifically <b>F</b> should consist of the rows 2 until 4 and columns 1 until 4 of matrix <b>E</b> .
	<b>Step 2</b> Identify the size of each dimension of the new matrix <b>F</b> and display them on the command window.
	<b>Step 3</b> Assuming now that matrix <b>F</b> has <i>m</i> rows and <i>n</i> columns, reshape matrix <b>F</b> in order to have <i>n</i> rows and <i>m</i> columns.
	<b>Step 4</b> Create a matrix <b>G</b> that will be matrix <b>F</b> sorted along column in a descending order.

<b>Exercise 5. Magic Matrices</b> <ul style="list-style-type: none"> <li>• Create a magic square</li> <li>• Replicate the magic square</li> <li>• Restore the original square</li> </ul>	
Key Functions	<code>magic</code> , <code>sum</code> , <code>diag</code> , <code>repmat</code> , <code>isequal</code> , <code>reshape</code>
<b>Task 1</b> Create a magic square	<b>Step 1</b> A magic square is a square matrix which has the special properties that if you take the sum along any row, column or either main diagonal, you get the same result.
	<b>Step 2</b> Open a new script in the Editor. The first line should create a magic square of size 3, using the <code>magic</code> function as follows: <code>A = magic(3)</code> . Save the script as <code>magic_matrices.m</code> on the H drive, and run the script.
	<b>Step 3</b> Add lines to the script to verify that the rows, columns and main diagonal of the square add up to the same number using <code>sum</code> and <code>diag</code> . Use help and doc to understand these functions. You can nest commands as in Exercise 2.
<b>Task 2</b> Replicate the magic square	<b>Step 1</b> Add lines using <code>repmat</code> to tile <b>A</b> four times vertically and two times horizontally, assigning the output to <b>B</b> ( <code>B = repmat(A, ...)</code> ). See section 4.3 for guidance.
	<b>Step 2</b> Check the row, column and diagonal sums of <b>B</b> .
	<b>Step 3</b> Replicate <b>B</b> , one time vertically and two times horizontally, assigning the output to <b>C</b> . Is <b>C</b> magic?

<b>Task 3</b> Restore the original square	<b>Step 1</b> After running the script created so far, look at the elements of A, B and C in the Command Window (type in the variable name and press Enter).
	<b>Step 2</b> The original magic square (A) can be extracted from C using row-column sequential indexing. Try it, assigning the output to (D). See section 4.4 for guidance.
	<b>Step 3</b> Add a line to verify that D is equal to A using <code>isequal</code> .
	<b>Step 4</b> The elements of A can also be extracted from C using linear indexing, taking every 4th value starting from 1 and ending at a suitable number to extract 9 elements in total. Try this and assign the output to E. Consider how you could convert E into the original magic square form and test this in your script.

## 5 Operators and Control - How to tell a computer what to do

**Operators** and **control statements** add power and flexibility to MATLAB programs. Operators are used to perform arithmetic, numerical and logical operations, using program statements that are concise and easy for a person to read.

**Conditional Statements** and **Loops** control program **flow**.

### Analogy: Baking a Cake

*You may have all the ingredients to make a cake. However, to make the cake you need to mix the ingredients together in certain quantities, in a certain order, and bake it for a specified amount of time. From a programming perspective, 'Operators' would be used to specify how you add the ingredients together. These include:*

*Arithmetic: **Add** the sugar and flour*

*Relational: Use an amount of milk **equal to** the amount of water*

*Logical: Use either chocolate **or** caramel in the sauce*

*On the other hand, 'Control Statements' would specify the flow of ingredients and the time of cooking. These include:*

*Conditionals: **If** the mixture has risen, remove the cake*

*Loops: **While** the cake is cooling, prepare the icing*

### 5.1 Arithmetic Operators

The general form of an arithmetic operator statement is:

```
operand1 operator operand2
```

There are two major types of arithmetic operations. *Array* operations are carried out element by element and the operand matrices must be of the same size, unless one is scalar. *Matrix* operations are defined by the rules of **linear algebra**. The full stop character (.) distinguishes the array operations from the matrix operations. For addition and subtraction, the matrix and array operations are the same, and here the full stop is not used. **Tip:** It is a common error to confuse matrix and array operators. Always consider which you should be using in a given situation. Below is a brief summary. For further details, look **here**.

<code>+</code> and <code>-</code>	Addition and subtraction. <code>A+B</code> adds <code>A</code> and <code>B</code> . <code>A-B</code> subtracts <code>B</code> from <code>A</code> .
<code>.*</code>	Array multiplication. <code>A.*B</code> is the element-by-element product of the arrays <code>A</code> and <code>B</code> .
<code>.^</code>	Array power. <code>A.^B</code> is the matrix with elements <code>A(i,j)</code> to the <code>B(i,j)</code> power.
<code>./</code>	Array right division. <code>A./B</code> is the matrix with elements <code>A(i,j)/B(i,j)</code> .
<code>.\</code>	Array left division. <code>A.\B</code> is the matrix with elements <code>B(i,j)/A(i,j)</code> .
<code>*</code>	Matrix multiplication. <code>A*B</code> is the linear algebraic product of <code>A</code> and <code>B</code> . For nonscalar <code>A</code> and <code>B</code> , the number of columns of <code>A</code> must equal the number of rows of <code>B</code> .
<code>^</code>	Matrix power. In the non-scalar power case, the calculation here involves eigenvalues and eigenvectors.
<code>/</code>	Matrix right division. <code>B/A = (A'\B')'</code> .
<code>\</code>	Matrix left division. <code>X = A\B</code> is the solution to the equation <code>AX = B</code> .

## 5.2 Relational Operators

**Relational operators** share the same general form as arithmetic operators, and are used to compare operands quantitatively, using operators like “less than”. Like array arithmetic operators, relational operators always operate element-by-element. If one operand is scalar, it is expanded to the size of the other operand. These operators return a logical array reflecting the outcome of the relational test. Here is a summary:

<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>==</code>	Equal to
<code>~=</code>	Not equal to

**Tip:** One equals symbol is used for assignment, while a pair is used for testing equality. Be careful not to confuse the two.

## 5.3 Logical Operators

**Logical operators** perform logical calculations and return logical values. When the operands are non-logical numerical data types, zeros are treated as false while all non-zeros are true. Logical operators come in two main types: *element-wise* for arrays, and *short-circuit* for scalars. Here is a summary:

<code>&amp;</code>	Logical AND. <code>A&amp;B</code> returns true (1) for every element location that is true (nonzero) in both arrays, and false (0) for all other elements.
<code> </code>	Logical OR. <code>A B</code> Returns true (1) for every element location that is true (nonzero) in either one or the other, or both arrays, and false (0) for all other elements.
<code>~</code>	Logical NOT. Complements each element of the input array. Nonzeros become false while zeros become true.
<code>&amp;&amp;</code>	Short-circuit AND
<code>  </code>	Short-circuit OR

For the short-circuit operators, if the outcome of the operation can be determined by the value of the first operand, the second is not evaluated. This can be more efficient in control statements (see below), and can also be used to avoid error conditions, such as division by zero, by making the first operand a test for the error condition. For example:

```
x = (b ~= 0) && (a/b > 2)
```

## 5.4 Conditionals

Conditional statements control program flow by selecting which blocks of code to execute based on the value of test expressions. They allow a MATLAB program to perform multiple tasks based on its inputs.

### Analogy: If Statement

*Let's assume you are helping a friend move and he tells you to put full boxes in the car but empty boxes in the bin. Your friend has created what is known as a conditional in the form of an `if` statement, i.e. 'If a box is full, put it in the car. Otherwise put it in the bin'. `if` statements in programs are used in the same way.*

`if` statements perform selection based on the value of a logical expression. In its basic form, an `if` statement has this form:

```
if logical-expression
    statements
end
```

The statements within the `if` statement are executed only if the logical expression is true. Otherwise those statements are skipped, and execution continues following the `end` line. For example:

```
if isprime(a)
    disp('prime number detected')
end
```

Simple expressions can be combined by logical operators into compound expressions (where short-circuit operation can be applied). For example:

```
if isprime(a) && a>100
    disp('Big prime number detected')
end
```

If the logical expression evaluates to a nonscalar value (e.g. a vector), then *all* the elements of the argument must be true (nonzero) for the expression to be evaluated as true. Note also that `if` statements can be *nested* by putting one inside another. The following example performs the same function as the preceding one:

```

if isprime(a)
    if a>100
        disp('Big prime number detected')
    end
end

```

For more complex conditional evaluations, the **else** and **elseif** statements can be incorporated into an **if** statement. The general form here becomes:

```

if expression1
    statements1
elseif expression2
    statements2
.
.
.
else
    statementsX
end

```

where any number of **elseif** groups can be used, and the **else** group is optional. The code above will execute **statements1** if **expression1** is true. If **expression1** is false, and **expression2** is true, then **statements2** will be executed. If none of the **elseif** expressions are true, the statements under the **else** line will be executed.

**Switch statements**

**switch** statements select between multiple code blocks based on the value of a single variable or expression (the switch). The basic form is:

```
switch expression
    case value1
        statements1
    case value2
        statements2
    .
    .
    .
    otherwise
        statements3
end
```

In a **switch** statement, any number of **case** groups can be used, and the **otherwise** group is optional. In MATLAB (unlike C), if any case statement is true, subsequent cases are not evaluated or executed.

## 5.5 Loops

Loop control statements enable a code block to be repeatedly executed. There are two loop types in MATLAB: **for** and **while**.

**Analogy: For and While Loops**

*Now let's assume your friend knows that he has three boxes. He may ask you to take those three boxes to the car. From a programming perspective he has created a **for** loop with three iterations, i.e. 'Take Box 1 to the car, take Box 2 to the car, take Box 3 to the car, then you are finished.'*

*However, if your friend did not know the number of boxes, he may ask you to take the boxes to the car until there are none left. In that case he has created a **while** loop, i.e. 'While there are still full boxes, take them to the car'.*

**for** loops execute a code block a number of times determined on entry to the loop. The syntax normally uses the **colon** operator introduced in section 4.

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



**for** loops can be nested and this can be useful to access each element in a two-dimensional matrix. However, nesting loops in MATLAB can result in slow code execution, especially for nesting beyond two levels. Fortunately, there are ways to avoid using heavily nested loops in many circumstances, and we will look at this in section 11.

**while** loops repeat execution of a block of code as long as a test expression is true.

```
while expression
    statements
end
```

To avoid infinitely-repeating **while** loops, one or more statements inside the loop must change the value of the test expression (although see the **break** statement below). This is in contrast to the **for** loop, where the statements inside the loop do not generally affect the number of loop repeats. Compare:

```
n=10
for i=1:n
    i
    n=n-1
end
```

```
n=10
while n>0
    n=n-1
end
```

### **continue and break**

**continue** and **break** statements can be used to interrupt the ongoing repeats in a loop. The **continue** statement immediately passes control to the next repeat of the loop in which it appears, skipping any remaining statements in the body of the loop. The **break** statement terminates the execution of a loop and passes control to the point following the **end** line in the loop.

In nested loops, **continue** and **break** statements only apply to the innermost loop containing that statement.

Good programming practice would be to limit the use of **continue** and **break** commands in code as they cause discontinuous jumps that are often difficult to follow, especially when looking through large programs.

## 5.6 Return and Keyboard

`return` and `keyboard` statements interrupt the execution of a MATLAB program. A `return` statement immediately exits a running program or function, returning control to either the keyboard or the function/script that called it (depending on how the function was called).

A `keyboard` statement suspends program execution at the point where it is encountered, and gives control to the keyboard. Keyboard mode is indicated by a `K` appearing in front of the usual command prompt. This allows the user to examine and change variable values at any point in a running MATLAB program and can be very useful for debugging code. Keyboard mode is terminated by entering `return` at the keyboard.

<b>Exercise 6. Control Statements - Basic Concepts</b> <ul style="list-style-type: none"> <li>• Create a 2D matrix</li> <li>• Matrix scanning and selective display</li> </ul>	
Key Functions	<b>rand, for, length, size</b>
<b>Task 1</b> Create a 2D matrix	<b>Step 1</b> Using the <b>rand</b> function create a 2-dimensional matrix with 3 rows and 4 columns and assign it to a variable.
	<b>Step 2</b> Display the whole matrix. Assuming you do not know the dimensions of the matrix, choose a function that will provide you with this information, apply it to your matrix, display the result, and make sure it is the expected one.
<b>Task 2</b> Matrix scanning and selective display	<b>Step 1</b> Imagine you want to see the individual elements of your matrix displayed on the command window one-by-one. Using a <b>for</b> loop, display on the command window every element of the first column of your matrix one-by-one (i.e. first display the element in row 1 column 1, and then display the element in row 2 column 1, etc.). If you do not remember how to index specific elements of a matrix, please read section 4.4 again. Also consider how you would define the start and end in the syntax of the for loop, assuming you do not know the dimensions of the matrix you scan.
	<b>Step 2</b> Using two <b>for</b> loops, one inside the other, display all the elements of your matrix. The first for loop will go through the columns and the second through the rows of the matrix.
	<b>Step 3</b> Now, instead of displaying all the elements of the matrix, display only those that are larger than 0.5. Consider using the if-else statement within the scanning code you have created in step 2 in order to examine the value of an element and then display it according to the above criterion.

<b>Exercise 7. Fibonacci Numbers</b> <ul style="list-style-type: none"> <li>• Open M Files</li> <li>• Sum of Fibonacci numbers - Part I</li> <li>• Sum of Fibonacci numbers - Part II</li> </ul>	
Key Functions	<b>for, while, disp, num2str</b>

<b>Task 1</b> Open M File	<b>Step 1</b> Change directory to where your Matlab files are stored, and open Fibonacci.Sequence.m. This script is the basis of the exercise
<b>Task 2</b> Sum of Fibonacci numbers Part I	<b>Step 1</b> The first six numbers of the Fibonacci sequence are: 0, 1, 1, 2, 3, 5. In order to calculate each number $F_n$ of the sequence, the previous two numbers, $F_{n-1}$ and $F_{n-2}$ are required. The formula is: $F_n = (F_{n-1}) + (F_{n-2})$ . Create a matrix $F$ that will contain the Fibonacci sequence and fill the first two elements with 0 and 1 respectively.
	<b>Step 2</b> Create a <b>for</b> loop that will calculate the first 20 numbers of the Fibonacci sequence. Every time a new number is calculated it must be displayed on the command window.
	<b>Step 3</b> Create a variable called 'sumFib' that will contain the sum of the calculated Fibonacci numbers. Initialize (i.e. set the initial value of) this variable to 0. Every time a new number from the Fibonacci sequence is calculated within the <b>for</b> loop, this variable will be updated by adding the new number.
	<b>Step 4</b> When the <b>for</b> loop terminates, display a message regarding the sum of the first 20 Fibonacci numbers. Consider using the function <b>disp</b> , as well as <b>num2str</b> to transform the value of the 'sumFib' variable into a string in order to be incorporated in the displayed message.
<b>Task 3</b> Sum of Fibonacci numbers Part II	<b>Step 1</b> Copy the Fibonacci.Sequence.m script and rename it into Fibonacci.Sequence2.m
	<b>Step 2</b> Replace the <b>for</b> loop with a <b>while</b> loop that will again calculate the Fibonacci numbers, add them, and save the result in the variable 'sumFib'. The while loop should stop executing once the summation of Fibonacci numbers exceeds 1000.
	<b>Step 3</b> When the <b>while</b> loop terminates, display a message regarding the sum of the Fibonacci numbers calculated so far, as well as the number of the Fibonacci numbers required to reach the termination criterion of the 'while' loop. In order to achieve this, consider defining a new variable that will act as a counter of the 'while' loop iterations. The new variable must be updated in each iteration of the loop.

<b>Exercise 8. Roots of quadratic equation</b> <ul style="list-style-type: none"> <li>• Open M Files</li> <li>• Quadratic equation roots Part I</li> <li>• Quadratic equation roots Part II</li> </ul>	
Key Functions	<code>if-else</code> , <code>switch</code> , <code>disp</code> , <code>num2str</code>
<b>Task 1</b> Open M Files	<b>Step 1</b> Open Quad_Equation_Roots.m in your Matlab files directory. This script is the basis of the exercise.
<b>Task 2</b> Quadratic equation roots	<b>Step 1</b> The roots of a quadratic equation $ax^2 + bx + c = 0$ depend on the value of the discriminant D, which is defined as $D = b^2 - 4ac$ . Define the variables a, b, c and initialize them with random values.
	<b>Step 2</b> Calculate the discriminant D.
	<b>Step 3</b> If $D > 0$ , the equation has two real roots. If $D = 0$ , the equation has one real root. Finally, if $D < 0$ , it has no real roots. Use conditional statement <code>if else</code> to display a message in each case indicating the number of the equations roots.
	<b>Step 4</b> The general form of the quadratic equation roots is $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ . In the case where there is at least one real root, calculate the corresponding roots and display them along with the previous message.

## 6 Programming - How to organise your code

Rather than entering commands into MATLAB one at a time at the command line, you can also write a series of commands to a program file that you then execute as you would any MATLAB function.

We already explored some programming essentials in section 2.2.6 and exercises 2-4. In particular, we introduced both the MATLAB Editor, and the simplest kind of MATLAB program file (M-file): the script. In this section we will go into a little more detail.

In actual fact, there are two kinds of M-file:

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions, which can accept input arguments and return output arguments. They operate in private workspaces: internal variables are local to the function. This means a variable created inside a function only exists within the function and cannot be accessed once the function has finished (reached the `return` command).

### 6.1 Scripts

Scripts are simply a list of commands stored in an M-file. When you invoke a script, MATLAB executes the commands found in the file. The result is exactly the same as if the commands were entered one-by-one in the command window.

Scripts share the base workspace with your interactive MATLAB session and with other scripts. They operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations.

### 6.2 Functions

Functions are program files that can accept input arguments and return output arguments. The names of the file and of the function *must* be the same.

Each function operates on variables within its own workspace, which is separate from the workspace you access at the MATLAB command prompt. The input arguments are the initial contents of the private function workspace, and the output arguments are what is returned to the workspace from which the function was called. Since functions can call functions, there can be a hierarchy of separate, private workspaces.

**Analogy: The Lawyer Function**

*Generally we go to see a lawyer if we need legal advice, a doctor if we are sick, and a hairdresser for a haircut. These people all do specific things, i.e. they have a specific function. In a similar way in programming we can create independent pieces of code called functions that perform certain jobs. Let's assume we have a 'Lawyer' function. When meeting a lawyer you may take certain documentation with you. This documentation would be called an 'Input Argument' to the 'Lawyer Function'. The lawyer would then perform some legal analysis, and may even give you some results. The results are what the 'Lawyer Function' returns.*

## 6.3 Script Components

To study the components of a script, we will look at `tutorial_script1.m`.

### 6.3.1 Comments

The first lines of the file are preceded by percent symbols, which in MATLAB denote **comments**. A comment is a message to someone reading or editing the M-file and the text is not interpreted as code by MATLAB. The first paragraph of comments at the top of an M-file is used to explain the intended usage of the script or function. These comments are also harvested for the **MATLAB help system**: `help tutorial_script1`.

Additional comments should be used in the body of the file to explain particular sections or lines of code. Finally, comments are also useful for temporarily disabling a piece of code during development. This is referred to as 'commenting out' code.

### 6.3.2 Housekeeping Code

Below the first help paragraph is a section of housekeeping code, used to keep the MATLAB workspace and display area tidy. In the tutorial example, we use `clear` to clean up the workspace, `close` to close figures, and `home` to tidy up the command window. A commented out line gives the option of using `clc` instead of `home`.

Usually housekeeping code goes at the start of a program to ensure that the figures shown and values in the variables are not left over from a different program.

### 6.3.3 Script Body - drawing a circle

The script body is the main subsection of executable code. Here it draws a circle, using built-in MATLAB trigonometric functions.

- First, the centre co-ordinates and radius of the circle are declared and reported using `disp`.

- Next, a vector (theta) of N=1000 parameter values is created, evenly spaced between 0 and 2pi. The standard parametric equations for a circle are applied to calculate x and y co-ordinate pairs for the perimeter of the circle.
- Finally, the circle is plotted in a figure window, and modified to display in the correct aspect ratio, using `axis`. `pause` is one way of interrupting program flow, and potentially requiring human interaction with the program.

**Running the script** There are two main ways to run a script in MATLAB.

1. From the Command Window, you can enter the script name. Note, this only works if the script is either in the current folder or the path.
2. From the Editor, you can press F5, or click the Run button (which features a green 'play' triangle).

## 6.4 Function Components

To study the components of a function, open up `tutorial_fun1.m` in the Editor. This function performs the same operations as `tutorial_script1.m` but is more flexible by allowing for input arguments.

### 6.4.1 Function Declaration

In MATLAB, `function` is a special function used to declare all other functions. The function declaration must be the first executable line of any MATLAB function, and follows the form:

```
function [out1, out2, ...] = myfun(in1, in2, ...)
```

`tutorial_fun1.m` has inputs called C, R and FIG, and an output called CIRCLE.

### 6.4.2 Help Comment Block

The help comment block is the first paragraph of comments in the function, and normally follows the function declaration. In `tutorial_fun1`, the help comment block is written in the general form of standard MATLAB functions. Compare:  
`help tutorial_fun1`, `help flipplr`.

It is good practice (but not essential) to write help comment blocks in this form.

### 6.4.3 Function Body

The body of `tutorial_fun1` is similar to that of `tutorial_script1`. Both script and function perform the same basic task but there are differences:

- The calls to `disp` and `pause` have been removed in the function



- In the script, the centre co-ordinates, radius and figure number are declared as constants, while in the function they are read in as input arguments
- In the function, a line has been added to form the output matrix

### Running the function

Unlike scripts, functions generally need to be provided with input arguments. These can be entered manually in the Command Window, or the function can be called from another script or function (more on this below).

If you try to run a function from the Editor, MATLAB will usually report an error since, by default, the input arguments are not specified here. However, it is possible to create default run conditions by editing the run configuration in the Editor. This is accessed via a drop-down menu next to the run button. By opening this dialogue, and copying in the example code from the comment block, we can run the function using the example argument values.

## 6.5 Modular Programming

A useful MATLAB program will often need to be more complex than the simple cases explored above. Rather than creating a large script or function that tries to do multiple tasks, it is generally more efficient to construct a modular program from simple building blocks. To see how this can be achieved, open `tutorial_script1b.m`.

This example script begins with a brief help comment subsection followed by a standard housekeeping subsection. In the script body, we see the introduction of the `input` function, which is a way of introducing user interaction into a running program. When `input` is called, the user is asked to enter input via a prompt string.

Below `input`, we see two lines calling `tutorial_fun1` with different inputs. Prior to the first function call, there is a call to `hold`, which enables subsequent graphical commands to be added to an existing plot.

The second function call is inside a `for` loop, which here means run what is inside the loop  $n$  times with index  $i$  increasing from 1 to  $n$ , in increments of one.

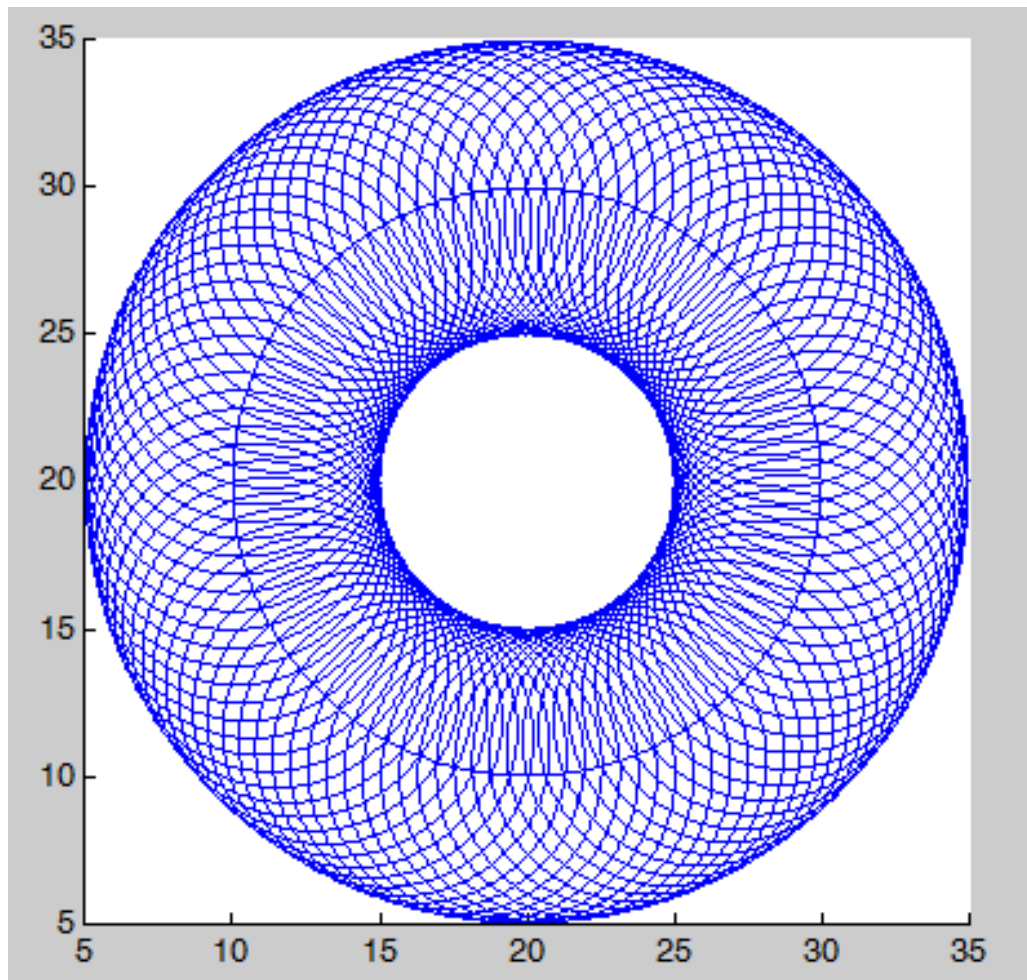


Figure 6.1: Example modular program output.

## 6.6 Toolboxes

MATLAB Toolboxes are specialized sets of functions serving a wide variety of purposes, including data analysis, data acquisition, signal and image processing, parallel computing, statistical data analysis, controller design and implementation. Depending on the MATLAB license there are different toolboxes available. To view installed toolboxes, use the `ver` command. Then by using the `help` command along with the toolbox directory name, a list of the functions available in the specific toolboxes is produced. The available MATLAB toolboxes are listed here: <http://uk.mathworks.com/products/>

<b>Exercise 9. Function for roots of quadratic equation</b> <ul style="list-style-type: none"> <li>• Create M Files</li> <li>• Main body of the function</li> <li>• Calling the function</li> </ul>	
Key Functions	<code>if-else</code> , <code>switch</code> , <code>disp</code> , <code>num2str</code>
<b>Task 1</b> Create M Files	<b>Step 1</b> Create a new M File and name it <code>quadEquationRoot.m</code> .
	<b>Step 2</b> Within the M File define a MATLAB function according to the form: <code>function [out1, out2, ...] = quadEquationRoot(in1, in2, ...)</code> . The inputs of the function should be the coefficients of the quadratic equation, and the output should be an array that will contain the equations roots.
<b>Task 2</b> Main body of the function	<b>Step 1</b> Copy the part of the <code>Quad.Equation.Roots.m</code> script that calculates the discriminant and the roots of the quadratic equation into the main body of the function. Do not include the part of the code responsible for displaying messages.
	<b>Step 2</b> Assign the calculated roots to the elements of the output array. Consider specifically the case where there are no roots.
<b>Task 3</b> Calling the function	<b>Step 1</b> Using random inputs call the ‘ <code>quadEquationRoot</code> ’ function from the command window and display the output.
	<b>Step 2</b> An alternative way to call the function is through an M File. Create a new M File and name it <code>CallQuadEquationRoot.m</code> . Within this file define again the required inputs, call the function ‘ <code>quadEquationRoot</code> ’, and display a message that will include the roots calculated, if any.

<b>Exercise 10. Noughts and Crosses</b> <ul style="list-style-type: none"> <li>• Open M Files</li> <li>• The players choose their symbols</li> <li>• The game begins</li> </ul>	
Key Functions	<b>if-else, switch, disp, return, for, while</b>
<b>Task 1</b> Open M Files	<b>Step 1</b> Change directory to where the Noughts and Crosses files are and open the XandO.m script. This script is the basis of the exercise.
	<b>Step 2</b> Take some time to understand the structure of the code according to the guidelines/comments in the code.
<b>Task 2</b> The players choose their symbols	<b>Step 1</b> Using the function ‘input’ the program must ask from the first player to provide a string (‘x’ or ‘o’) that will correspond to this players symbol. This string must be saved to a variable, and this variable must be then used in a conditional statement to determine the second players symbol.
<b>Task 3</b> The game begins	<b>Step 1</b> After a player has chosen a square on the board, the program must check if this square is empty. An empty square on the ‘board’ matrix has the value ‘e’. Consider applying a control statement that will place the players symbol at the specified position on the board if this position is empty. If it is not empty, a relevant message should be displayed and the program must be terminated.
	<b>Step 2</b> At this step the symbol of the current player must be drawn on the board. The function ‘drawSymbol’ in your current folder can be used for this purpose. Then the available squares on the board must be reduced by one.
	<b>Step 3</b> A check on the board must be performed to determine whether there is a winner. In order to do this, use the function ‘checkWin’ in your current folder. Make sure you provide the right inputs and outputs. If the output of the above function is equal to 1, this means that the current player is the winner, a relevant message must be displayed and the program must be terminated.

**Step 4**

Steps 1-3 must also be followed when the second player. Make sure you copy this code at the right place and make the necessary changes in the variables and messages to be displayed so that they correspond to the second player.

**Step 5**

The board has a limited number of squares, 9 specifically. So far, the program allows each player to play only once. In order for the game to continue, one player must play after the other and the criteria for the game termination should be either to have a winner or to fill completely the board. Since now only one round of the game has been implemented in the code, a loop needs to be introduced that will allow repetitive rounds as long as the criteria for the termination of the game have not been satisfied. Hint: Consider using the 'available\_squares' variable, which is updated every time a player finishes their round, as a criterion to the loop.

## 7 Error Handling - How do deal with things that go wrong

### 7.1 Errors

Typically, in MATLAB code several types of errors might occur. Some of the main classes of errors are described here.

#### 7.1.1 Typographical Errors

These errors are very easy to identify, and are caused by misspelling of a command or a variable that could lead to either an error message returned by MATLAB or unpredictable results. Where MATLAB can return an error message, it will show a sentence in red starting with `Undefined function or variable 'x'`

#### 7.1.2 Syntax Errors

These errors occur when the syntax used to call a function or to use an operator is incorrect. In these cases, MATLAB provides you with an error message indicating the location and type of error. Syntax Errors may include:

- Parenthesis Errors - This error indicates too many or too few brackets have been used. Errors of this form usually read `Error: Expression or statement is incorrect--possibly unbalanced (, or [` but could also say `Error: Unbalanced or misused parentheses or brackets.`
- String Errors - These errors indicate that either a variable is of the incorrect type or is used incorrectly, and may read `Only input must be numeric or a valid numeric class name.` Another straightforward string error would be leaving off an inverted comma, for example writing `A = 'Hello` and pushing enter. In this case MATLAB will return an error along the lines of `Error: A MATLAB string constant is not terminated properly.`

Other common syntax errors may include incomplete expressions, as in `x = 1+2+` in which case MATLAB returns `Error: Expression or statement is incomplete or incorrect.` You may also try and use a character that is non-standard for MATLAB (such as just typing `'`), in which case the error reads `Error: The input character is not valid in MATLAB statements or expressions.`

#### 7.1.3 Array Indexing and Assignment Errors

Array indexing errors typically occur if you try to access a position that is outside the bounds of an array. For example, if an array is defined as `A = [1, 2, 3];` and you try to access position four by writing `A(4)`, MATLAB will give an error reading `Index exceeds matrix dimensions.` Similarly, because the first index in a MATLAB array is always 1 (not 0), if you try to write `A(0)` you will also get an error reading `Subscript indices must either be real positive integers or`

**logicals.** Array indexing errors often happen in `for` loops when the loop counter runs higher than the largest array index.

On the other hand, if you try to perform arithmetic operations on matrices of different sizes you may receive an error reading `Matrix dimensions must agree` or `Subscripted assignment dimension mismatch`. Another assignment error occurs when you try to use a single `=` sign in an equivalence statement instead of `==`. For example, writing `if x = 3` returns `Error: The expression to the left of the equals sign is not a valid target for an assignment`.

#### 7.1.4 Algorithmic Errors

These errors occur when there is an error in the process followed by the MATLAB code to produce the desired result. These errors are quite difficult to detect since MATLAB does not provide error or warning messages. One way to detect them is to go through the code step-by-step and compare each steps result with the expected one. This can be done with a debugger.

**Manual Error Detection** It is often good programming practice to perform checks throughout the program to ensure that the program is performing as expected. One such way to do this is to display certain variables so that you can manually check that they have an expected value during program execution. Another way is to use an `if` statement to validate your data. For example, if you are writing a calendar program to keep track of the days in a month, you know that the value of a day must be somewhere between 1 and 31. Thus you can write the following `if` statement to check, where `calDay` is the variable:

```
if calDay >= 1 && calDay <= 31
    'Everything Okay'
else
    'Something has gone wrong - the day is too large or too small'
end
```

This will allow you to quickly see whether the program is functioning correctly, especially if the day counter is only a small part of a much larger program.

## 7.2 Debugging

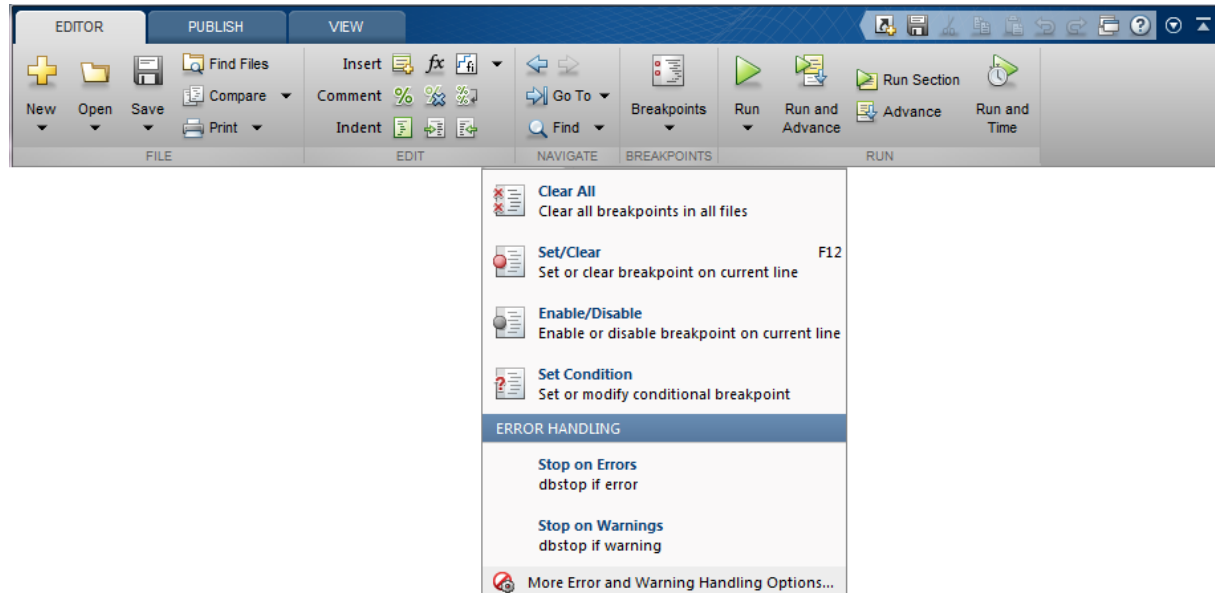
While manual error checking is useful, sometimes it is still very difficult to detect where an error in from, and in these cases MATLAB provides a useful set of tools known as a debuggers. These tools allow you to step through single lines or blocks of code to identify an error. There are several debugging tools available in MATLAB, and in this section the most commonly used are presented:

### 7.2.1 Debugging in the MATLAB editor

**Using MATLAB GUI tools:** In order to go through a part of a MATLAB code, a breakpoint can be used that stops execution of a program at a specified line.

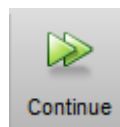
Breakpoints are marked as red-filled circles right next to the number of a line, and can be applied in two ways:

- By left-clicking on the dash line next to the number of the line where we want the breakpoint to be placed.
- By placing the cursor anywhere on the desired line and then clicking Editor → Breakpoints → Set/Clear (or pressing F12 on the keyboard).

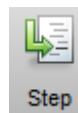


When the user runs a program, the execution will stop at the line where the first breakpoint is placed. Then the user has the following options:

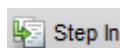
- Click Continue: the execution will continue until the next breakpoint is met. If there are no breakpoints later in the code, the program will execute until it terminates.



- Click Step: when this option is chosen, the current line of the program will be executed and the execution will stop at the next line. This button provides the user with a way of investigating step-by-step the functionality and results of the code.



- Click Step In: This option provides you access to the content of a function called at the line where the execution of the program currently is. Then by using Step, the user can go through each line of this function.



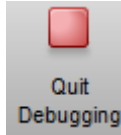
- Click Step Out: This option allows the user to step out of a function that



has previously stepped in without executing some or all of the lines in this function.



- Click Quit Debugging: This option terminates program execution and allows the user to access the command prompt, and edit the m-files.



**Using a try / catch block** A special case of debugging is the try/catch block, and its structure is shown below.

```
try
    some code

catch
    error handling code

end
```

This block is used to capture errors in specific lines in the code whose execution is expected to fail under specific circumstances, and avoid failure of the entire program. The part of the code that is right after the **try** section corresponds to the code being executed and checked for errors, and the part of the code that is after the **catch** section corresponds to the code being executed when an error has been captured.

### 7.2.2 Debugging in the MATLAB command prompt

Similarly to the MATLAB Editor Breakpoints menu, debugging can be performed using specific commands in the command window. A subset of the available commands is briefly described in this section.

- **dbstop** *in file*: to set a breakpoint at the first possible line in the file.
  - **dbstop in file at location**: to set a breakpoint at the specified location in the file.
  - **dbstop in file if expression**: to set a breakpoint at the line in the file where this expression exists.
  - **dbstop if condition**: to pause execution at the line that satisfies the condition.
- **dbclear**: to clear all the breakpoints
  - **dbclear all**: to remove all breakpoints in all files.
  - **dbclear in file**: to remove all breakpoints in the file.
- **dbquit**: to terminate program execution and exit debugging.

<b>Exercise 11. Error Checking</b> <ul style="list-style-type: none"> <li>• See some common errors</li> <li>• Check a user input</li> <li>• Use the Matlab Debugger</li> </ul>	
Key Functions	<code>input</code> , <code>ischar</code> , <code>disp</code> , <code>if-else</code> , <code>isnumeric</code> , <code>while</code>
<b>Task 1</b> See some common errors	<b>Step 1</b> Open the <code>commonErrors.m</code> script and take some time to look through the code. It contains three errors. See if you can find them all before running the script.
	<b>Step 2</b> Run the script and look at the errors Matlab shows. For each error, go to the line it indicates and fix the error. For more information on the types of common errors, see Chapter 7 in the handbook.
<b>Task 2</b> Check a user input	<b>Step 1</b> Open the <code>starSign.m</code> script and take some time to familiarise yourself with what the code is doing. Remember, if you don't know what a command does, you can type <code>help</code> followed by the name of the command in the Command Window to get more information.
	<b>Step 2</b> This code asks users for their date of birth, and returns their star sign based on that information. At the moment, there is no error checking in the code, and a user could input an invalid date of birth. Use the knowledge you have gained so far to write in some manual error checks and output messages that will tell the user that what they have entered is invalid.
	<b>Step 3</b> Once you are confident in your error checks, run your code and ask someone near you or the teacher to try and 'break' the program by entering invalid inputs.
<b>Task 3</b> Use the Matlab Debugger	<b>Step 1</b> Open <code>codeToDebug.m</code> and run it. The code executes without errors, but the answer is incorrect. Add some breakpoints in the script, and use the Debugger to step through the code line by line to help you find where the error has occurred.

## 8 Graphics 1: Figures, Axes and Graphs

To begin introducing MATLAB graphics, we need to start with some terminology:

### 8.1 Figures, Axes and Graphs

- A *figure* is a MATLAB window used for graphical display.
- An *axes* is a 2D or 3D data space defined within a figure.
- A *plot* is any graphic display you can create within a figure window. Figures can contain any number of plots and each plot is created within an axes.
- A *graph* is a plot of *data* within an axes. Most plots are graphs.

To create the graph below, we run the following code:

```
x = -10:.1:40;

y = [1.5*cos(x)+4*exp(-.01*x).*cos(x)+exp(.07*x).*sin(3*x)];

plot(x,y)

title('y = 1.5cos(x) + 4e^{-0.01x}cos(x) + e^{0.07x}sin(3x)')

xlabel('X Axis')

ylabel('Y Axis')
```

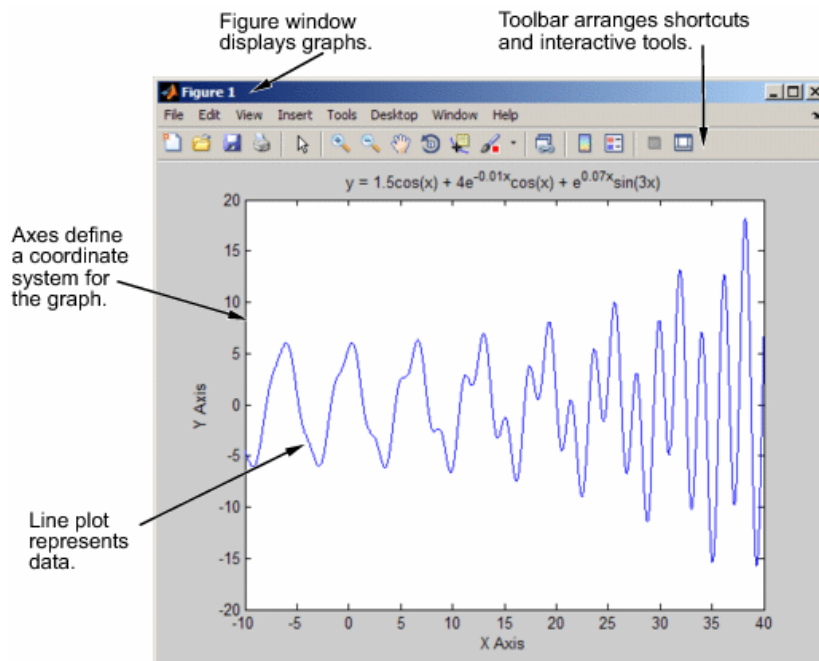


Figure 8.1: The basic components of a graph.

## 8.2 Setting up Figures

Various MATLAB commands are useful for setting up figures. We consider a few of the most useful here.

### 8.2.1 figure

`figure` is a command (function) that opens or selects a figure window. Entering `figure` on its own creates a new figure window with default properties, and makes it the current figure for subsequent plotting commands. `figure(h)` will select figure number `h` if it already exists, and create it if not.

```
>> figure  
>>
```

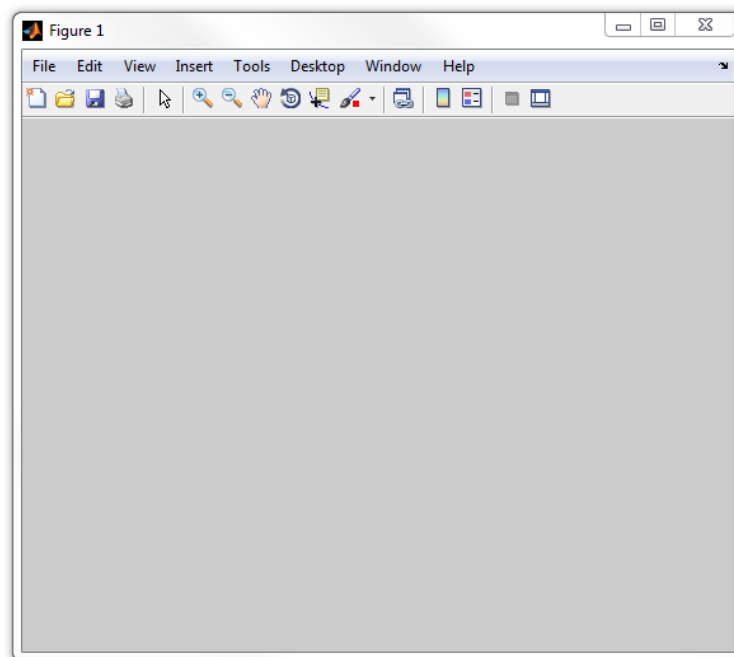


Figure 8.2: Default Figure.

### 8.2.2 subplot

To place multiple plots on a single figure, in a regular tiled pattern, you can use `subplot`. For example:

```
subplot(2,3,4)
```

defines a 2-by-3 grid on a figure and opens or selects the 4th subplot *counting along rows*, from the top left. Note that `subplot` creates an axes in the position specified. A graphics command at this point will output to this axes object. Try `plot(rand(1,10))`. Another call to `subplot` will create/select a different axes:

```
subplot(2,3,2)
```

It is usually a good idea to use the same tiling grid in multiple calls to `subplot` on a single figure because, when subplots overlap, the existing axes is deleted.

```
subplot(2,4,5)
```

If we avoid overlaps, it is possible to create complex `subplot` arrangements. For example:

```
subplot(2,1,2)
```

```
subplot(2,2,2)
```

```
subplot(4,4,1)
```

```
subplot(4,4,2)
```

```
subplot(4,4,5)
```

```
subplot(4,4,6)
```

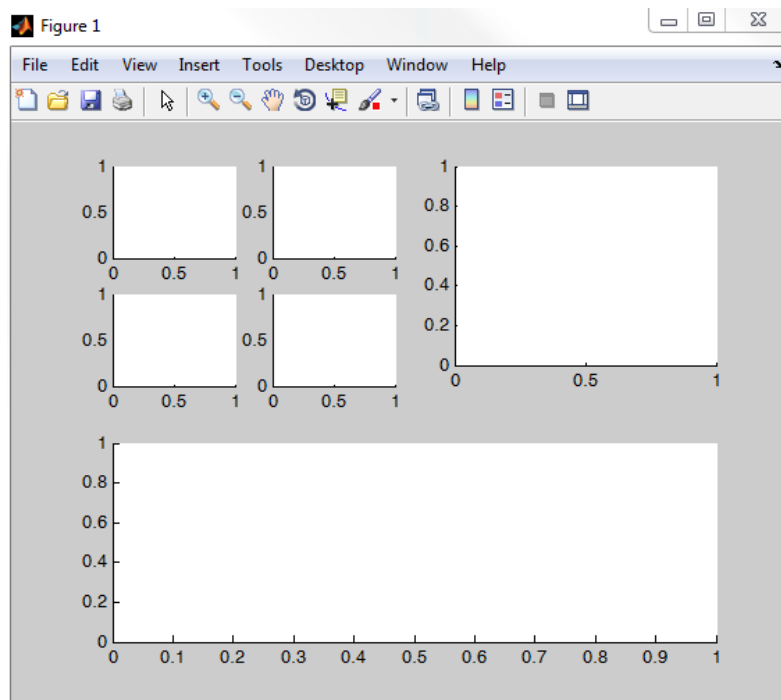


Figure 8.3: Tiled subplot example.

With Data Linking, such a figure can be used for continuous monitoring of multiple data streams.

For complete flexibility, you can use the position argument: `subplot('Position',[left, bottom, width, height])`, which creates an axes at the position specified by a 4-element numerical vector with normalised coordinates in (0,1). For example:

```
close all
```

```
subplot('position',[0.1, 0.4, 0.6 0.2])
```

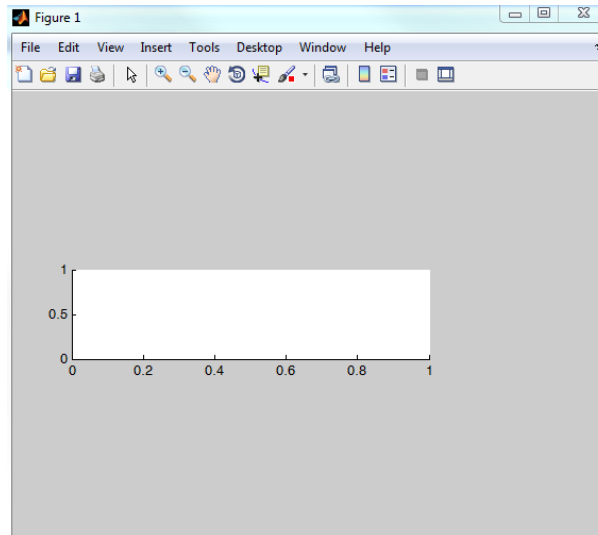


Figure 8.4: Positioned subplot example.

### 8.2.3 axes and axis

`axes` can be used to create an axes, and/or specify its properties. `axes` on its own creates an axes object in the current figure using default properties. Alternatively, `axes('PropertyName',PropertyValue,...)` creates an axes object having specific property values in name-value pairs. We will cover these properties in detail in Graphics 2, but here is a simple example:

```
close all

axes('Box','on','Color','y')
```

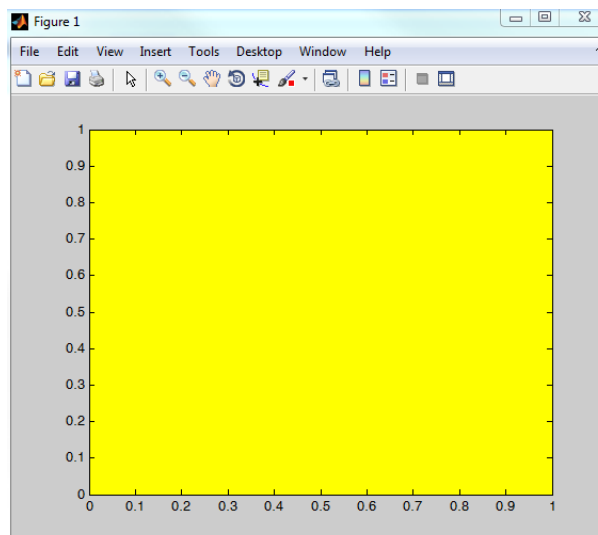


Figure 8.5: Example axes with defined properties.

`axis` is used to manipulate commonly used axes properties, such as the axis limits: `axis([xmin, xmax, ymin, ymax])`. Be careful not to confuse `axis` and `axes`.

### 8.3 GUI Tools

Tools for manipulating and interacting with figure graphics are available in a toolbar at the top of the figure. We will look at these in detail in the next two exercises.

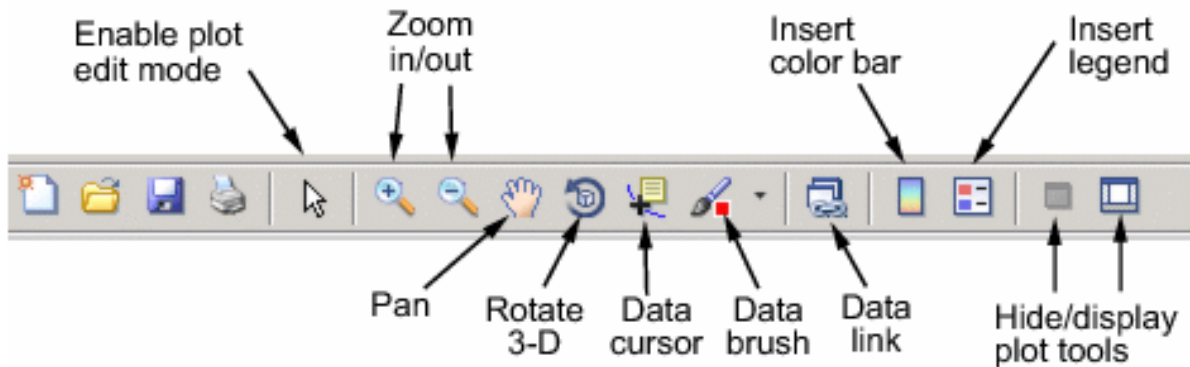


Figure 8.6: GUI Tools

#### Exercise 12. Interactive Plot Tools

- Open the Plot Tools
- Edit the Figure Properties
- Edit the Axes Properties
- Edit the Line Plot Properties
- The Figure Palette

##### Task 1 Open the Plot Tools

##### Step 1

To get started, create the graph from section 8.1, complete with text labels, via copy and paste.

##### Step 2

Click the Show Plot Tools icon in the figure window.



By default, this opens the **Figure Palette**, **Plot Browser** and **Property Editor** tools. These tools can also be selected individually in the **View** menu.

##### Step 3

Each tool can be moved around and docked or undocked from the figure, but we will leave them where they are.

##### Task 2 Edit the Figure Properties

##### Step 1

Ensure Edit Plot is selected in the toolbar. 

##### Step 2

Click the dark grey background area around the graph.

##### Step 3

Change the background colour by selecting from **Figure Color** in the **Property Editor** .

##### Step 4





Enter a Figure Name; deselect Show Figure Number.



<b>Task 3</b> Edit the Axes Properties	<b>Step 1</b> Select the axes by either clicking in the central white area or clicking the axes title in the <b>Plot Browser</b> tool.
	<b>Step 2</b> Toggle the axes visibility off and on using the tick box next to the axes title in the <b>Plot Browser</b> .
	<b>Step 3</b> In the <b>Property Editor</b> , change both the fill and line colours to contrast with each other and the background.
	<b>Step 4</b> Toggle the x and y grids on and off in the <b>Property Editor</b> , and toggle the box off and on.
	<b>Step 5</b> Examine the limits of each axis in the <b>Property Editor</b> and change the font to SansSerif.
<b>Task 4</b> Edit the Line Plot Properties	<b>Step 1</b> Select the line plot by clicking on the line in either the main figure window or the <b>Plot Browser</b> .
	<b>Step 2</b> In the <b>Property Editor</b> , try each of the Plot Types, before returning to the original line plot.
	<b>Step 3</b> Try each line style, and increase line thickness to 2.0 in the <b>Property Editor</b> .
<b>Task 5</b> The Figure Palette	<b>Step 1</b> With the <b>Figure Palette</b> Tool, you can add subplots to your figure, change variable names and plot types, and add annotation to the figure.
	<b>Step 2</b> Add, and then delete, some Annotation features.
	<b>Step 3</b> The Plot Catalogue Tool offers GUI access to many plot types. To open it, right click y in the Variables box and choose Plot Catalogue.
	<b>Step 4</b> In the Plot Catalogue Tool, change Plotted Variables to x,y, choose comet and click Plot in New Figure. Then close the new figure and the Plot Catalogue window.

### Exercise 13. Data Exploration Tools

- Zoom
- Pan
- Rotate
- Cursors
- Data Brushing
- Data Linking



<b>Task 1</b> Zoom	<b>Step 1</b> If you have just finished the previous exercise, select Hide Plot Tools on the Figure Toolbar. If you need to generate the graph again, copy and paste the earlier code into the Command Window.
	<b>Step 2</b> Select the Zoom In tool. 
	<b>Step 3</b> Move the mouse pointer inside the axes noting how the pointer changes to the zoom in symbol. Left click once near the line plot and note how the display changes. Repeat this process several times. To return to the default zoom level, double click anywhere in the axes.
	<b>Step 4</b> For a more controlled zoom, drag a selection box over an area (left click-and-hold, drag, release). The axes is redrawn, changing the limits to display the specified area. Double click to restore the original zoom.
	<b>Step 5</b> For a constrained zoom, right click in the axes with the Zoom In tool selected, and choose Horizontal Zoom from Zoom options. Drag a selection range to see how this works before returning to the default zoom level.
<b>Task 2</b> Pan	<b>Step 1</b> Select the Pan tool from the Figure Toolbar. 
	<b>Step 2</b> Click, hold, and drag to pan.
	<b>Step 3</b> Pan works a lot like zoom in that double clicks return to the default view, and there are both unconstrained (default) and axis-constrained versions available through a right-click menu.
	<b>Step 4</b> Try both Horizontal and Vertical panning, before returning to the default view.
<b>Task 3</b> Rotate	<b>Step 1</b> Select the Rotate tool from the Figure Toolbar. 
	<b>Step 2</b> Grab the axes with a left click-and-hold, and move the graph around in three dimensions by moving the mouse. This option is not so useful with a two-dimensional plot, but can be useful for visualising three-dimensional datasets. Restore the original view with a double click.
<b>Task 4</b> Cursors	<b>Step 1</b> Select the Cursor tool from the Figure Toolbar. 

	<b>Step 2</b> Left Click somewhere on the line plot to place a cursor (or data tip) on the graph. Left click and drag to move the cursor around over the graph.
	<b>Step 3</b> To place additional data tips, right click and choose Create New Datatip.
	<b>Step 4</b> Right click and select Delete All Datatips.
<b>Task 5</b> Data Brushing	<b>Step 1</b> Data brushing is used to interactively select data for further analysis or modification. Select the Data Brushing tool  and drag a selection box inside the axes to choose a dataset.
	<b>Step 2</b> Right click on the brushed data, choose Create Variable and save the brushed data as a named variable.
	<b>Step 3</b> Other options in the Brush menu include Remove (to remove the brushed data from the graph), Remove Unbrushed, and Replace With (to replace all brushed data point y values with a constant value).
<b>Task 6</b> Data Linking	<b>Step 1</b> Linked plots are graphs in figure windows that visibly respond to changes in the current Workspace variables they display and vice versa. Select the Data Linking tool  and note the Linked variables line that appears below the Figure Toolbar.
	<b>Step 2</b> In the Command Window, alter the value of y using <code>y=y.*x</code> (or similar) and observe the resulting changes in the linked figure.
	<b>Step 3</b> Switch off linking by deselecting the linking tool and note how further changes to the data values are ignored in the unlinked plot.

## 8.4 Plotting Functions

Matlab includes a wide range of plotting functions, as summarised in the next two figures. A linked version of these figures is available **online**.

Most plot types are accessible through the GUI, via the Plot Catalogue (see Exercise 5), but they can also be produced from the command line. In the remainder of this section, we focus on the 2D line and bar graphs, via the exemplar functions `plot` and `bar`.

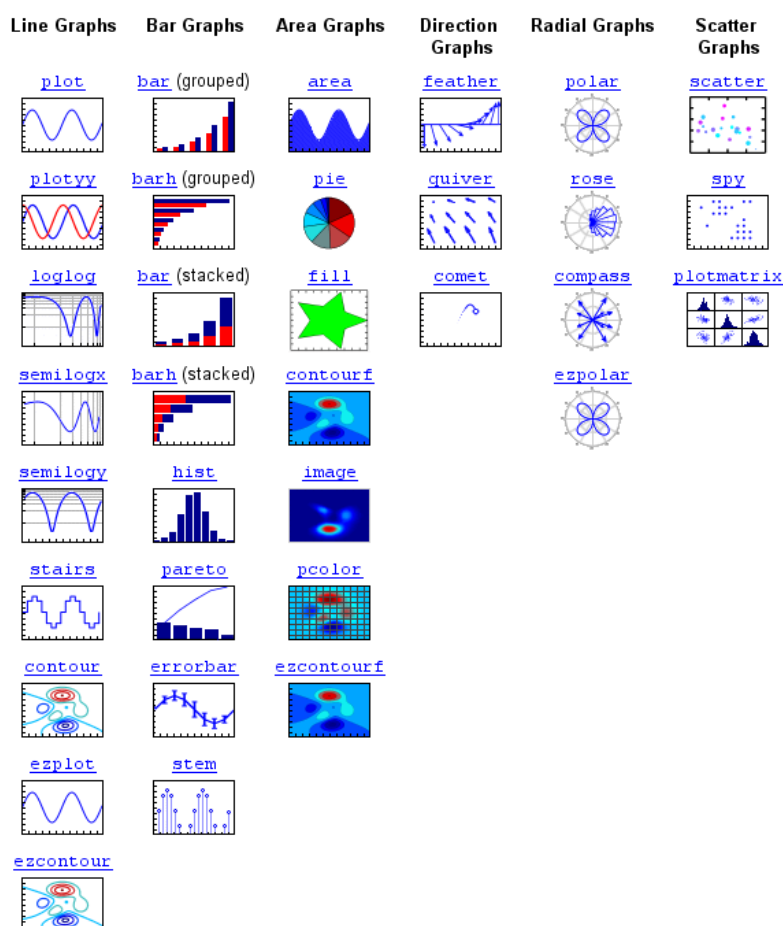


Figure 8.7: Two-dimensional plotting functions.

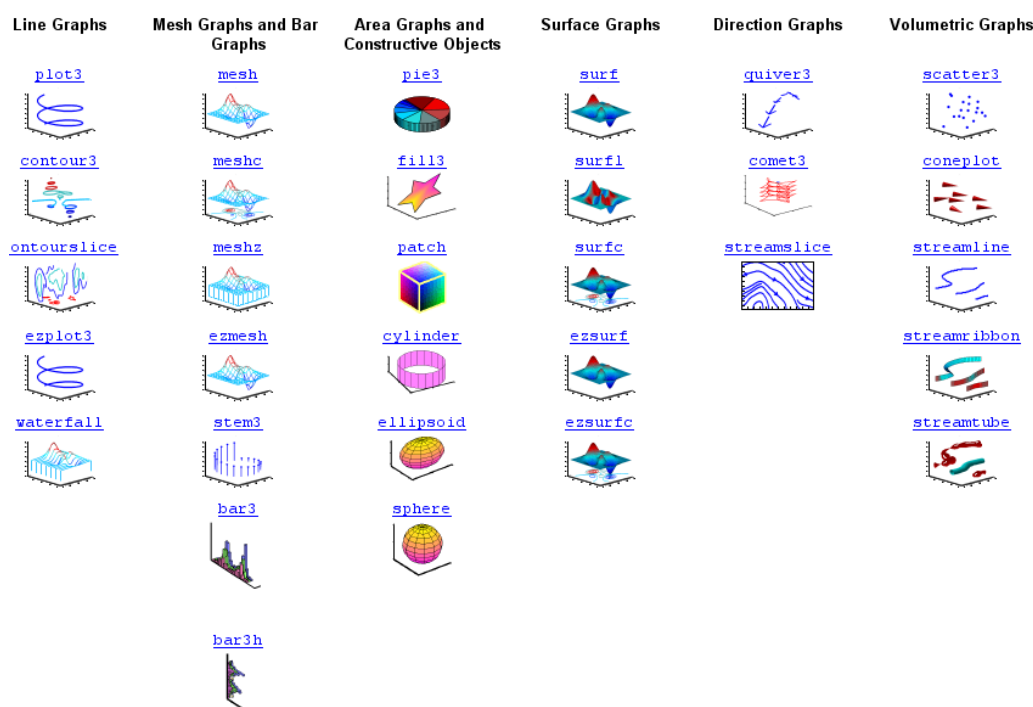


Figure 8.8: Three-dimensional plotting functions.

### 8.4.1 Line Graphs: plot

`plot` is the standard MATLAB line plotting function. If `y` is a vector, then `plot(y)` produces a piecewise linear graph of `y` versus the index of its elements, `1:length(y)`. If you specify two vector arguments, `plot(x,y)` plots `y` versus `x`. Thus, `plot(y)` is shorthand for `plot(1:length(y),y)`. Expanding on our earlier example:

```
x = -10:.1:40;

y = [1.5*cos(x)+4*exp(-.01*x).*cos(x)+exp(.07*x).*sin(3*x)];

figure(1); clf;

subplot(3,1,1)

plot(y)

subplot(3,1,2)

plot(1:length(y),y)

subplot(3,1,3)

plot(x,y)
```

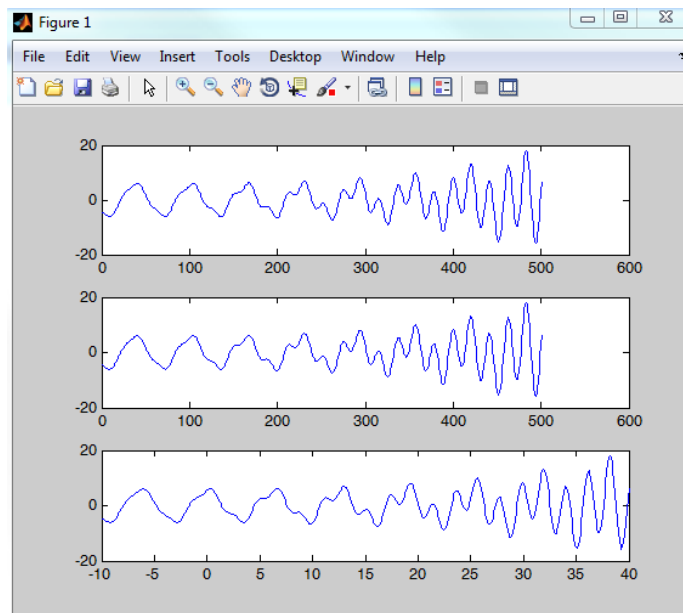


Figure 8.9: Example plot.

A simple way to alter the properties of the lineseries plotted is to add a **LineSpec** string after the `x,y` pair, which consists of a triplet specifying **Line Style**, **Marker Symbol** and **Color** (in any order). For example:

```
subplot(3,1,3)

plot(x,y,'r:x')
```

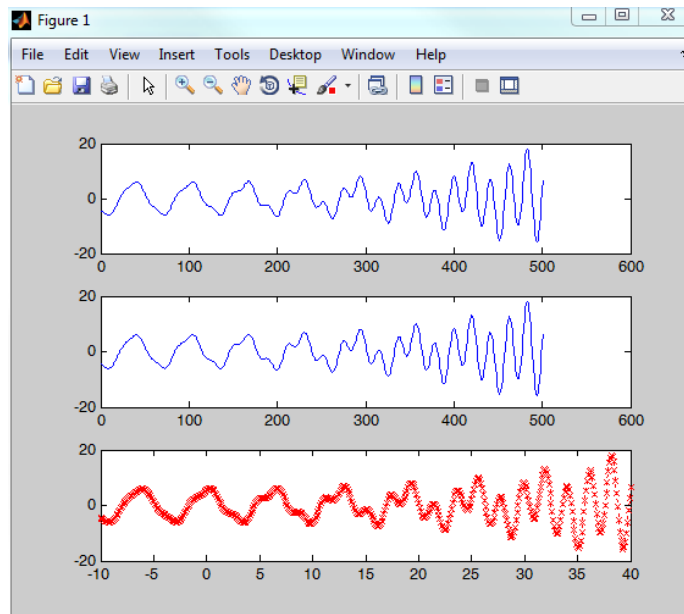


Figure 8.10: Example linespec plot.

Almost all high level line plotting functions accept LineSpec string arguments, so it is well worth learning them. More detailed property manipulation is available through PropertyName, PropertyValue pairs, as we saw above for `axes`. For example:

```
plot(x,y,'r-s','MarkerEdgeColor','b','MarkerFaceColor','y')
```

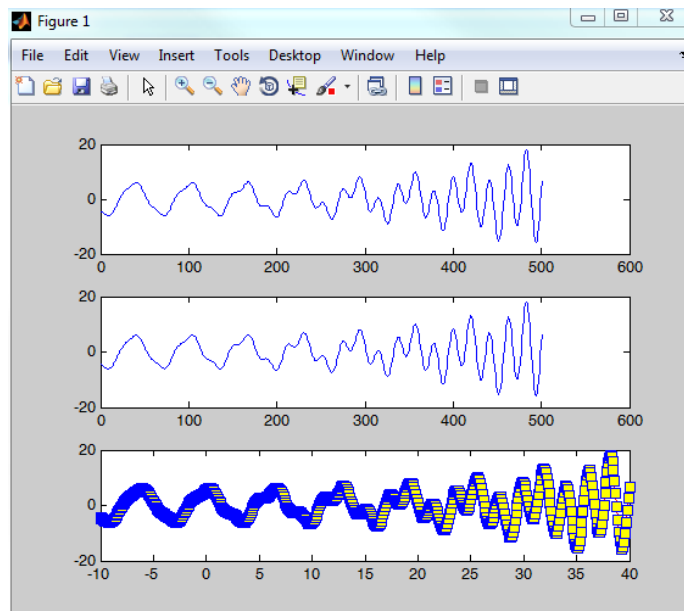


Figure 8.11: Example PropertyName, PropertyValue plot.

You can plot multiple x,y series using `plot(X1,Y1,LineSpec,...,Xn,Yn,LineSpec)`, but these statements can become rather long and unwieldy. An alternative is to use `hold` to accumulate plots on a single axes.

```
hold on
```

```
plot(x,y-20,'b-s','MarkerEdgeColor','g','MarkerFaceColor','r')
```

```
plot(x,y+20,'c-s')
```

If one or both of an x, y pair is a matrix, multiple lines (one for each column) are plotted using cycling, contrasting colours.

```
subplot(3,1,1)
```

```
plot(magic(9))
```

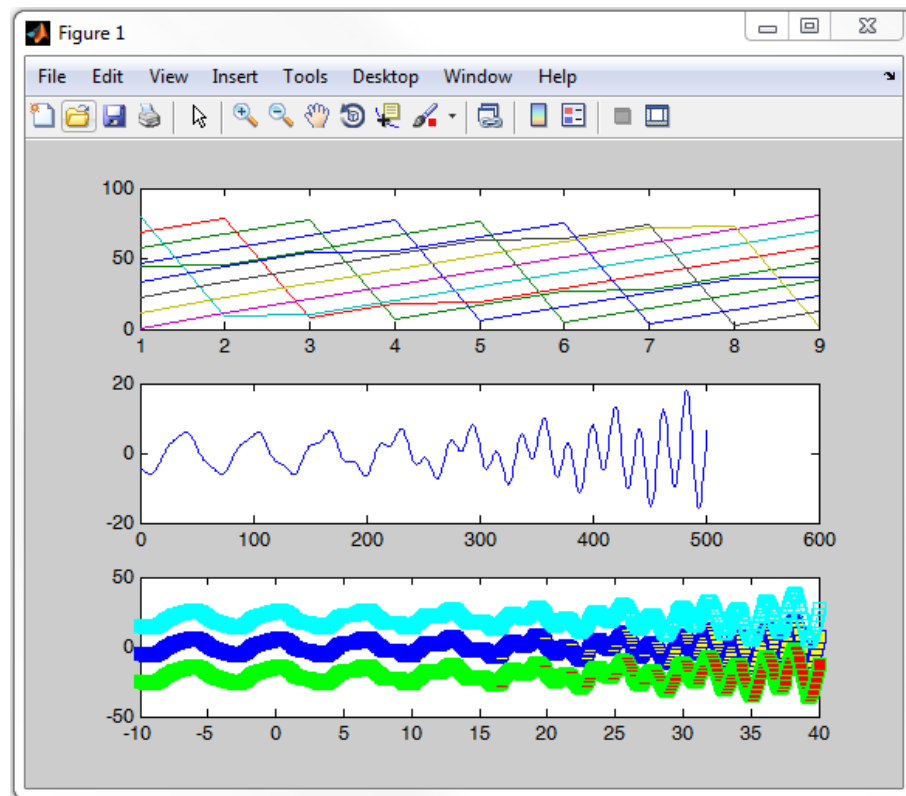


Figure 8.12: Example hold and matrix plots.

Finally, any call to plot can return a column vector of handles to lineseries objects, one handle per line.

#### 8.4.2 Bar Graphs: `bar` and `barh`

Bar graphs display vector or matrix data as vertical (`bar`) or horizontal (`barh`) bars. `bar(Y)` draws one bar for each element in `Y`. If `Y` is a matrix, `bar` groups the bars produced by the elements in each row. The x-axis scale ranges from 1

up to `length(Y)` when `Y` is a vector, and the number of rows when `Y` is a matrix. `bar(x,Y)` draws bars at locations specified in `x`. The 'style' argument specifies the choice between 'grouped' (default), or 'stacked' bars. For example:

```
close all

subplot(3,1,1)

bar(magic(5))

subplot(3,1,2)

barh(magic(5))

subplot(3,1,3)

bar(magic(5),'stacked')
```

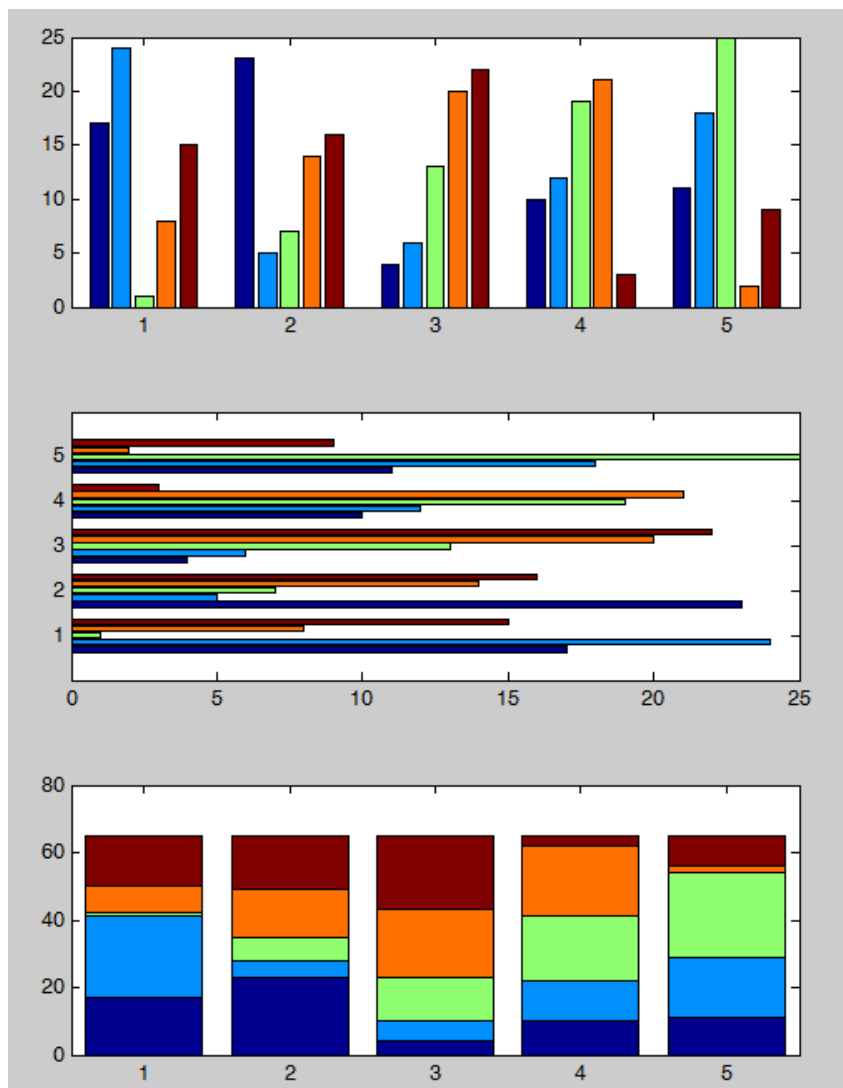


Figure 8.13: Example bar graphs.

## 9 Graphics 2: Objects and Images

### 9.1 Objects, Handles and Properties

**Figures**, **axes**, and **plots** are all examples of *graphics objects* in MATLAB. Each instance of an object has a unique identifier, or *handle*. Handles are returned by functions that create graphics objects. For example:

```
close all

h1 = figure

h2 = axes('Box','on','Color','y')
```

In addition, handles for the current figure and current axes are returned by **gcf**, and **gca**. Using an object handle, you can alter the *properties* of an existing graphics object. You can also specify property values when you create a graphics object, as we saw above for **axes** and **plot**.

Graphics object properties allow us to flexibly alter figure appearance. To see the extensive range of options available, here are links to lists of **Figure Properties**, **Axes Properties**, and **Lineseries (or Chart Line) Properties**, with the latter being the type of plot object created by **plot**. For an overview of object properties, use this [link](#).

Objects are organised into a hierarchy in MATLAB, with figures being the *children* of the root object, axes being children of figures and plot objects being children of axes. The child-parent relationship within the **figure**→**axes**→**plot object** hierarchy can be used to reference related components within a complex graphical display setup.

#### Querying and Setting Property Values

The two most important functions for fine-tuning MATLAB graphics are **get** and **set**, which query and specify graphic object properties, respectively. **get(h)** returns all properties of the graphics object identified by the handle **h** and their current values:

```
get(gcf)

get(gca)
```

**get(h,'PropertyName')** returns the value of the named property. Compare:

```
get(gcf,'Children')

gca
```

**set(h)** returns adjustable properties and possible values for object **h**:

```
set(gcf)

set(gca)
```

**set(h,'PropertyName',PropertyValue,...)** sets the named properties to the specified values.

```
set(gca,'Color','k')
```



<b>Exercise 14. Exploring Graphics Objects</b> <ul style="list-style-type: none"> <li>• Try the tutorial script</li> <li>• Adjust figure properties</li> <li>• Adjust axes properties</li> <li>• Adjust graph properties</li> </ul>	
Key Functions	<code>get</code> , <code>set</code> , <code>gcf</code> , <code>gca</code> , <code>legend</code>
<b>Task 1</b> Try the tutorial script	<b>Step 1</b> Ensure that the current folder is H:\matlab\tutorial3
	<b>Step 2</b> Open and run tutorial_script2.m.
	<b>Step 3</b> Select Save As from the Save drop-down menu and name your copy of the script graphics_script.m
<b>Task 2</b> Adjust figure properties	<b>Step 1</b> Add a line to your script using <code>get</code> and <code>gcf</code> to examine the object properties of the figure.
	<b>Step 2</b> Use <code>set</code> and <code>gcf</code> to change the background colour (Color) to white. This can be achieved using either a linespec-style colour character or an RGB 3-element vector. Adding <code>pause(1)</code> after each new graphics command in the script will help to view sequential changes.
	<b>Step 3</b> Use <code>set</code> and <code>gcf</code> to set the figure position to [200,400,700,600]. Try some alternative values. Note the current screen resolution is given by <code>get(0,'ScreenSize')</code> .
	<b>Step 4</b> Use <code>set</code> and <code>gcf</code> to set the figure resize property to off. Note the effect on the figure.
<b>Task 3</b> Adjust axes properties	<b>Step 1</b> Use <code>get</code> and <code>gca</code> to examine the axes properties. You may want to comment out the equivalent command for figure properties from Task 2, Step 1.
	<b>Step 2</b> Use <code>set</code> and <code>gca</code> to switch off the axes box.
	<b>Step 3</b> Use <code>set</code> and <code>gca</code> to change x axis limits (XLim) to [0,10].
	<b>Step 4</b> Use <code>set</code> and <code>gca</code> to change direction of the x axis (XDir) to reverse.
	<b>Step 5</b> Use <code>set</code> and <code>gca</code> to change y tick labels (YTickLabel) to the sequence 1:9 (see section 4.2).

<b>Task 4</b> Adjust graph properties	<b>Step 1</b> Use <code>handles = get(gca,'Children')</code> to get handles for the three graphs; handles will be a 3 element vector
	<b>Step 2</b> Use <code>get</code> with (elements of) the handles vectors to examine the graph properties. Use linear matrix indexing here. Can you work out which handle is for which line?
	<b>Step 3</b> Use <code>set</code> with handles to change the marker size in each graph ( <code>MarkerSize</code> ).
	<b>Step 4</b> Use <code>legend</code> to give the figure a legend with text labels for each graph.
	<b>Step 5</b> Use <code>set</code> with handles to make one graph invisible ( <code>Visible off</code> ).

## 9.2 Working with Images

Monochrome images are represented in MATLAB as matrices, with each pixel represented by a single matrix element. RGB colour images are handled as three-dimensional arrays, with the first plane in the third dimension representing the red, the second plane representing green, and the third plane representing blue.

MATLAB supports three different numeric classes for image display: `double`, `uint16` and `uint8`, and image display commands interpret the data values differently depending on the data type. The integer data types are most useful for storage, while double precision is the most flexible for processing.

The major read and write functions are `imread` and `imwrite`, while `imfinfo` returns information about an image file. To see a table listing the file formats supported by these functions, run `imformats`.

### 9.2.1 Displaying Images

To display an image, use `image`, or `imagesc`. For example:

```
figure

imfinfo peppers.png

peppers = imread('peppers.png','png');

image(peppers)
```

To remove axis ticks and set the correct aspect ratio for an image you can run

```
axis off

axis image
```

To display an image with a one-to-one mapping of matrix element to screen pixel, you can resize the figure and axes. For example:

```
[m,n,z] = size(peppers);

figure('Units','pixels','Position',[100 100 n m])

image(peppers); axis off;

set(gca,'Position',[0 0 1 1])
```

Note, when you set the axes Position to [0 0 1 1] so that the displayed image fills the entire figure, the aspect ratio is not preserved in printing because MATLAB uses the PaperPosition property to determine printed size and shape. To preserve the aspect ratio when printing, set the PaperPositionMode to auto as follows:

```
set(gcf,'PaperPositionMode','auto')
```

Images can be cropped in MATLAB using sequential matrix indexing. For example:

```
peppercrop = peppers(1:(m/2),1:(n/2),:);

figure; image(peppercrop)
```

## 9.3 Printing and Exporting

Printing and exporting involves outputting graphics objects (usually figures). There are four basic operations:

- **Print**
- **Print to File**
- **Export to File**
- **Export to Clipboard**

These can be accessed via either the Command Line or the Graphical User Interfaces. We consider the former here, and the latter in exercise 12 below. The general printing and export function is `print`. On its own, `print` send the contents of the current figure to the printer using the printing options specified by `printopt` (this usually sends printout to the default printer). The table below summarises further options available through the addition of input arguments to `print`.

<code>handle</code>	Print the specified object
<code>filename</code>	Print to a named PostScript file
<code>-ddriver</code>	Print using a <b>specified driver</b> (default is in <code>printopt</code> )
<code>-dformat</code>	Export to the clipboard (Windows only). Format must be either <code>-dmeta</code> or <code>-dbitmap</code> .
<code>-dformat filename</code>	Export the figure to file using a specified <b>graphics format</b> .
<code>-options</code>	Specify additional <b>printing options</b> .

To demonstrate some options, we start with an annotated peppers image:

```
peppers = imread('peppers.png','png'); [m,n,z] = size(peppers);
h = figure('Units','pixels','Position',[100 100 n m],'Color','k')
image(peppers); set(gca,'Position',[0 0 1 1]); axis off
annotation('textarrow',[.2,.3],[.8,.6],'String','Chili','Color','w');
set(gcf,'InvertHardcopy','off')
```

and copy to the clipboard (so you can paste it into a word document for example):

```
print(h,'-dmeta')
```

Next, print to PostScript files:

```
print(h,'-dps','PepperFigBW.ps')
```

```
print(h,'-dpasc','PepperFig.ps')
```

Finally, export to Encapsulated PostScript and PDF formats:

```
print(h,'-depsc','PepperFig.eps')
```

```
print(h,'-dpdf','PepperFig.pdf')
```

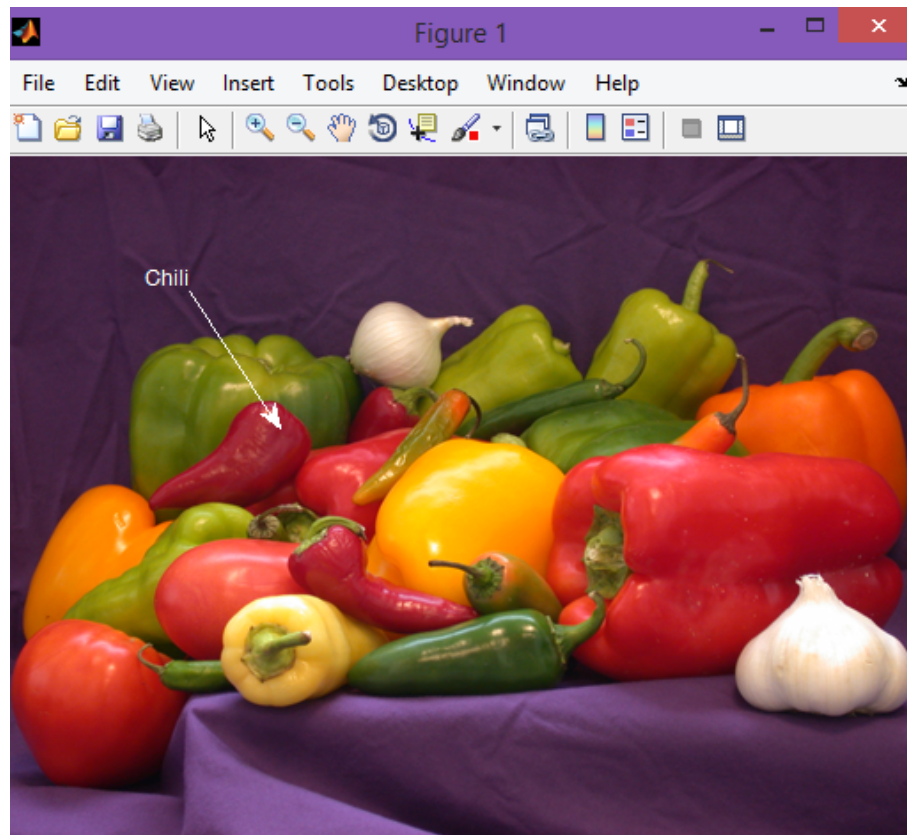


Figure 9.1: Example peppers image.

<b>Exercise 15. Printing and Exporting Graphics through the GUI</b> <ul style="list-style-type: none"> <li>• Print Preview</li> <li>• Export to File</li> <li>• Exporting Line Plots</li> </ul>	
Key Functions	<code>imread</code> , <code>image</code> , <code>annotation</code> , <code>set</code>
<b>Task 1</b> Print Preview	<b>Step 1</b> Open the annotated peppers figure by copying and pasting in code from above.
	<b>Step 2</b> Select <b>File Print Preview...</b> from the Figure window to open the preview.
	<b>Step 3</b> On the Layout tab, switch to landscape mode, and alter the manual size and position settings. Try Fill page, before switching Placement to Auto.
	<b>Step 4</b> On the Lines/Text tab, increase the Font Size and Line Width until you can clearly see both the annotation text and arrow.
	<b>Step 5</b> On the Advanced tab, try each of the Renderers and note any apparent changes.
<b>Task 2</b> Export to File	<b>Step 1</b> Close the print preview, and select <b>File Export Setup...</b>
	<b>Step 2</b> Select Fonts from the Properties menu, and alter the font to Mistral in size 18. Then click Apply to Figure.
	<b>Step 3</b> Select Lines, change to a fixed line width of 1.5, and Apply to Figure.
	<b>Step 4</b> Click Export... and export as a tiff with no compression. Then export as a MATLAB figure, and a jpeg. Compare file sizes and output image quality.
	<b>Step 5</b> Repeat step 4 after applying different Rendering settings (e.g. dpi), fonts and line widths. Note the change in file size with increasing dpi settings.
<b>Task 3</b> Exporting Line Plots	<b>Step 1</b> MATLAB line plot default settings are optimised for visualisation, rather than printing. Working with the example line graph from the start of this section (or another line plot with axes ticks and labels), export first using the default settings and look at the results.
	<b>Step 2</b> Alter export settings for better appearance of lines and text in the exported figure.

<b>Exercise 16. Creating a Video</b> <ul style="list-style-type: none"> <li>• Generate Plots</li> <li>• Create Video</li> <li>• Save Video</li> </ul>	
Key Functions	VideoWriter, writeVideo, plot
<b>Task 1</b> Create a Video	<b>Step 1</b> Using 'help', read about the VideoWriter function.
	<b>Step 2</b> Use the VideoWriter function to create a Video object called myDataPlots, and name the video 'A Video of my Data.avi'.
	<b>Step 3</b> Using the open function, open the 'myDataPlots' video object.
	<b>Step 4</b> Create a for loop that will run 5 times. In the for loop, generate 20 random numbers and plot them in a figure window each time the for loop runs. Make sure you store the figure handle in a variable.
	<b>Step 5</b> Use the pause command with a pause of 1 second to see each plot appear one-by-one when the for loop runs. The pause should be placed after the plot command inside the for loop.
	<b>Step 6</b> Once you have verified that you can see each plot appear one-by-one, use the getframe function to transform the plot in the figure to a video frame. Remember to assign the frame to a variable. Now write the frame you have created to your video using the writeVideo function. Creating the frame and writing the frame to your video should happen inside the for loop where you plot your data (after the pause command).
	<b>Step 7</b> Outside of the for loop, at the end of your code, close the video object using the close function. Check that your video has been created correctly by opening it in an external media player (such as Windows Media Player). Spend some time familiarising yourself with the different video options in Matlab, such as frame rate and video quality.

## 10 File Handling - How to handle internal and external files and data

MATLAB is able to import and export data from many different **file types**, and different import and export methods are used for each type. In this section, we look at four of the most common file types. Image data files are covered in section 9.2.

### 10.1 MAT-Files

MAT-files are the default data storage file type in MATLAB. They are binary files that can store the variables in your current workspace for later use. MAT-files use the .mat extension.

To import and export from MAT-files, we can use the `load` and `save` functions. `save`, by itself, creates a binary MAT-file named `matlab.mat` in the current folder, and saves all current variables in the workspace into that file. `load`, by itself, loads all the variables stored in `matlab.mat` into the current workspace, overwriting existing values. For example:

```
x = 2
y = 5
save
x = 5000
load
```

This is not generally the most useful way to use these functions, since each `save` will overwrite any previously saved data. Instead we can include a filename argument. Additional arguments can be used to load or save only particular variables:

```
save temp x
x = 5000
y = 25
load temp
```

To inspect the values in a saved file, we can use the command `whos -file filename`. This function returns the name, dimensions, size, and class of all variables in the specified MAT-file. Both `whos -file` and `load` will search the MATLAB path as well as the current folder.

```
which durer.mat
whos -file durer.mat
```

```
load durer.mat
```

### Function and Command Syntax

In common with many MATLAB built-in functions, `load` and `save` can be called in two different ways. The examples above use the *command syntax*, where spaces are used between the function name and arguments. Alternatively we can use *function syntax*, where arguments are enclosed in parentheses and separated by commas. Function syntax is more powerful than command syntax for two main reasons:

1. We can use string variables as input arguments
2. We can return values from the function into MATLAB variables

For example:

```
filename = 'durer.mat'  
durerdata = load(filename)
```

Note, `durerdata` is returned as a **struct**, which stores heterogeneous data in named *fields*. You can reference (access) a field like this:

```
durerdata.caption
```

### 10.1.1 GUI Import and Export

It is also possible to use the GUI to inspect and load MAT-file data. To see the variables in a MAT-file before loading the file into your workspace, click the file name in the Current Folder browser. Information about the variables appears in the Details Panel (bottom left of the GUI). Double clicking loads all the data from the selected file into the workspace. To select and load MAT-file variables interactively, use any of the following options:

- Click Import Data on the Desktop Home tab, or run `uiimport`.
- Drag variables from the Details Panel of the Current Folder browser to the Workspace browser.

To save data via the GUI, use either of the following options:

- Click Save Workspace on the Desktop Home tab.
- Drag variables from the Workspace browser to the Current Folder browser.

### Best Practice

While GUI tools may be easier to use when you are getting started, command line methods are much more powerful in MATLAB. In particular, the GUI is only useful for interactive sessions and not for automation.



## 10.2 Excel Files

MATLAB features several ways to **read from** and **write to** Excel spreadsheets. The main read and write functions are `xlsread` and `xlswrite`. Generic import and output methods can also be used, but we focus on the specific functions here.

### Excel Version Support

Full functionality relies on Excel being installed on the system running MATLAB and these functions work best in Windows. If you have Excel 2007 or 2010 (including the COM server component), then `.xlsx` (and `.xlsb`, `.xlsm`) as well as `.xls` files are supported. For Excel 2003, an Office Compatibility Pack is required to read 2007+ formats. Compatibility can be tested using `xlsfinfo`, which returns 'Microsoft Excel Spreadsheet' for readable files and an empty string otherwise.

The `xlsread` function is flexible, and a useful feature is the ability to separately read in numeric and text data. For example, open up the `hospital.xls` example excel spreadsheet (from the MATLAB stats toolbox demo), in Excel. Next, import the data into MATLAB using:

```
[num,txt] = xlsread('hospital.xls');
```

Looking at the Workspace Browser, you will see that the text data have been imported into a MATLAB as a **cell array**. A cell array is a collection of containers called cells and each cell can store data of different sizes and types. Here, the cell array is used to store strings of different lengths.

For spreadsheets with multiple worksheets, `xlsread` only imports the first worksheet by default. To import a different worksheet, you can get the names of the sheets using `xlsfinfo`, and then read in a named sheet:

```
[type, sheets] = xlsfinfo('USTreasury.xls')
```

```
[num,txt] = xlsread('USTreasury.xls',sheets {2})
```

Note the braces around the number 2 here. This is how you reference the contents of a cell in a cell array.

To interactively select data from excel worksheets, use -1 in place of a sheet name:

```
[num,txt] = xlsread('USTreasury.xls',-1)
```

To import all worksheets into a **struct**, use `importdata`:

```
treasuries = importdata('USTreasury.xls')
```

The counterpart to `xlsread` is `xlswrite`. With `xlswrite`, you can export data from the Workspace to any worksheet in an Excel file, and to any location within that worksheet. By default, `xlswrite` writes to the first worksheet in a file, starting at cell A1. If the target file already exists, `xlswrite` writes data in the existing format. If not, a new file is created using the format corresponding to the file extension specified (the default is `.xls`). If Excel is not installed, `xlswrite` exports to a `.csv` file.

## 10.3 Text (ASCII) Files

There are many methods available in MATLAB to **import to** and **export from** ASCII data files.

The `load` and `save` functions can work with text rather than binary files, if an additional argument is added. For example:

```
x = magic(3);

save('temp.txt','-ascii')

load -ascii 'temp.txt'
```

Alternatively, you can launch the generic **import wizard GUI** by clicking the Import Data button or running `uiimport`. The `importdata` function is largely equivalent but without a graphical interface.

For ASCII files, `uiimport` and `importdata` should automatically detect:

- Row and column headers.
- Field delimiters (characters between data items, such as commas, spaces, tabs, or semicolons).
- MATLAB comments (lines that begin with a percent sign).

In general `load`, `uiimport` and `importdata` assume that the data in an ASCII file are:

- Rectangular (with the same number of data fields in each row)
- Numeric

Non-numeric data headers can be handled by calling `uiimport` or `importdata`, with function syntax and a single argument, as we saw for excel files:

```
mydata = importdata('headers.txt')
```

### ASCII files with specific formatting

If the text file formatting is known and/or complex, it can be processed using specific import and export function as follows:

- Comma-separated value (.csv) files can be read and written using `csvread` and `csvwrite`.
- Delimited numeric data, with a specified delimiter such as tab, can be read and written using `dlmread` and `dlmwrite`.
- Complex text files can be read using `textscan` (advanced).
- Single lines of text can be read using `fscanf`, `fgetl` or `fgets`, while `fprintf` prints a single line of formatted data to a file. These low-level functions can be very flexible, but require much more work to set up than the high-level functions.

## 10.4 Binary Files

Generic binary files are handled using low-level I/O functions based on functions in the ANSI Standard C Library. However, the MATLAB versions of `fread` and `fwrite` are vectorized for efficiency (see section 11).

Before reading or writing to a file using the low-level I/O functions, it must first be opened using `fopen` to obtain a file identifier (`fopen` is also required for low-level I/O to text files). The first input argument to `fopen` is a file name string, while the second is a permission string describing the type of access required (e.g. read, write, append, or update). After completing low-level file I/O, a file should be closed using `fclose` (**Tip:** learn to pair `fopen` and `fclose` calls). Here is a simple example of a binary file write:

```
x = [1:3;4:6]

fid = fopen('temp.bin','w');

fwrite(fid, x);

fclose(fid);
```

followed by a binary file read:

```
fid = fopen('temp.bin');

y = fread(fid)

fclose(fid);
```

By default, `fwrite` writes the values from a matrix in column order as 8-bit unsigned integers (uint8). Also by default, `fread` reads a file 1 byte at a time, interpreting each byte as a uint8. `fread` creates a column vector output, with one element for each byte in the file. The values in the output vector are of type double (the default MATLAB numerical data type).

To read only a specified number of values, or import into two dimensional matrix, a size argument to `fread` can be added as follows:

```
fid = fopen('temp.bin');

y = fread(fid, 4)

frewind(fid);

y = fread(fid, [2,2])

frewind(fid);

y = fread(fid, [2,inf])

fclose(fid);
```

To write and read with a different precision, add a precision string argument:

```

xpi = pi*x

fid = fopen('xpi.bin','w+');

fwrite(fid, xpi, 'double');

frewind(fid);

y = fread(fid, [2,inf], 'double')

fclose(fid);

```

<b>Exercise 17. Magic Files</b> <ul style="list-style-type: none"> <li>• MAT Files</li> <li>• Excel Files</li> <li>• Binary Files</li> </ul>	
Key Functions	save, load, xlswrite, xlsread, fopen, fclose, fwrite, fread, frewind
<b>Task 1</b> MAT Files	<b>Step 1</b> Start MATLAB and change the current folder to H:\matlab\tutorial4 (see section 2.2.4).
	<b>Step 2</b> Create two magic squares of size five and ten and <b>save</b> them to two separate MAT files named magic5.mat and magic10.mat, respectively. See section 10.1
	<b>Step 3</b> <b>clear</b> the original magic square variables and then reload them from the MAT files using <b>load</b> .
	<b>Step 4</b> Confirm that the reloaded variables are the expected magic squares.
<b>Task 2</b> Excel Files	<b>Step 1</b> Using <b>xlswrite</b> in the following form: <b>xlswrite(filename,A, sheet, range)</b> (where A is the matrix to be written and the other variables are strings; read the doc examples), write the small magic square to an excel spreadsheet called magic.xls, starting at cell C2, and naming the sheet magic5.
	<b>Step 2</b> Write the large magic square to a second worksheet in the same file called magic10, starting from cell B3.
	<b>Step 3</b> Open the spreadsheet in Excel (outside MATLAB) to confirm it has written correctly.
	<b>Step 4</b> Use <b>xlsfinfo</b> to check the format of the excel file, and the names of the worksheets.

	<p><b>Step 5</b> Clear the magic square variables from the workspace and read them back in from the Excel spreadsheet using <code>xlsread</code> in interactive mode (set sheet to -1) to read the magic squares back into MATLAB. Select the data before clicking OK in the pop-up box.</p>
<p><b>Task 3</b> Binary Files</p>	<p><b>Step 1</b> Using <code>fopen</code>, open a binary file called <code>magic.bin</code> for reading and writing, discarding existing contents. Use <code>help</code> to choose an appropriate permission string for <code>fopen</code>.</p>
	<p><b>Step 2</b> Using <code>fwrite</code>, write the small magic square to the start of the binary file.</p>
	<p><b>Step 3</b> Clear the small magic square variable and use <code>frewind</code>, and <code>fread</code> to rewind the file and then read the small magic square back in from the open binary file. You will need to give a size argument to <code>fread</code>. See the examples in section 10.4 for guidance.</p>
	<p><b>Step 4</b> Close the binary file using <code>fclose</code> and reopen it in append, read-write mode (<code>a+</code>). Write the large magic square to the end of the file.</p>
	<p><b>Step 5</b> Rewind the file, and read in both magic squares from the binary file.</p>
	<p><b>Step 6</b> Finally, close the binary file.</p>

<b>Exercise 18. Demographic statistics of UK and European Union countries</b> <ul style="list-style-type: none"> <li>• Open Excel and MAT Files</li> <li>• Plotting of the data</li> <li>• UK population increase rate</li> <li>• Age structure of UK population</li> </ul>	
Key Functions	<code>xlsread</code> , <code>load</code> , <code>plot</code> , <code>XTickLabel</code> , <code>for</code> , <code>sort</code> , <code>sum</code> , <code>bar</code> , <code>disp</code>
<b>Task 1</b> Open Excel and MAT Files	<b>Step 1</b> Open the Excel file named ‘UK demographics.xls’ in the H:\Matlab\tutorial4 directory outside Matlab and study the different sheets and fields.
	<b>Step 2</b> Open the Excel file named ‘UK demographics.xls’ through MATLAB and assign the different sheets in different variables, as it has been demonstrated in this chapter.
	<b>Step 3</b> Open the MAT file named ‘EuropeanUnionDemographics.mat’ in the H:\Matlab\tutorial4 directory through MATLAB. This file contains three columns: the first one has the names of the countries in the EU, the second has the population of each country, and the third has the area (in $km^2$ ) of each country.
<b>Task 2</b> Plot of the data	<b>Step 1</b> Plot the UK population over time. The x axis of the plot should have the corresponding years.
	<b>Step 2</b> Plot the UK population in 2011 with respect to the age group. Then, change the x axis labels to correspond to the different age groups. In order to do this, consider using the plot property ‘XTickLabel’.
<b>Task 3</b> UK population Increase rate	<b>Step 1</b> Create a new script and name it ‘UKpopulationIncreaseRate.m’.
	<b>Step 2</b> Using the UK demographics over time data, calculate the rate of population change over year. For example for year 2010, the rate is (population in 2010 - population in 2009)/(population in 2009). Save this data along with the corresponding year in a matrix called ‘PopulationChangeRate’.
	<b>Step 3</b> Sort the ‘PopulationChangeRate’ matrix and display the year with the highest population change rate. Consider using either a loop to examine all the data and find the highest value, or the MATLAB function ‘sort’.

<b>Task 4</b> Age structure of UK population	<b>Step 1</b> Create a new script and name it 'UKpopulation-AgeStructure.m'.
	<b>Step 2</b> The UK population in 2011 for five year bands is given. Categorize the UK population according to larger age groups, i.e. 0-14, 15-64, 65+, and calculate each groups' population. Then, calculate the total population for that year and the percentage for each age group.
	<b>Step 3</b> Plot using bar plots the population of each group. In the axis display the age range for each group.
<b>Task 5</b> Population density of EU countries	<b>Step 1</b> Create a new script and name it 'EUCountriesPopulationDensity.m'.
	<b>Step 2</b> Using the EU demographical data, calculate the population density of each country.
	<b>Step 3</b> Sort the EU countries according to population density in descending order.
	<b>Step 4</b> Display a message with the countries that have the highest and lowest population density.

## 11 Performance - How to determine how efficient your code is

MATLAB is an interpreted computing system. This means that the code is not generally compiled into a standalone program, like most applications you will find on a desktop PC. Instead, the code is interpreted on the fly at run-time. The advantages of an interpreted system include rapid development time and the MATLAB GUI facilities. The disadvantage is that interpreted code tends to run slower than compiled code, particularly when the code is not optimised. In this section, we look at ways to monitor and improve the performance of a MATLAB program.

### 11.1 Measuring Performance

#### 11.1.1 Stopwatch timing

A quick way to check the speed of a particular section of code is to use the stopwatch timer functions, `tic` and `toc`. Calling `tic` initialises the stopwatch timer, and a subsequent call to `toc` returns the time elapsed since the last call to `tic`. Wrapping any block of code in a `tic toc` pair should reveal how long that code takes to run. For small programs, it may be more informative to calculate average run time.

```
tic
    for i=1:1000
        r=rand(500)*i;
    end
t1=toc/1000
```

#### 11.1.2 The Profiler

The **profiler utility** is a good way to work out where bottlenecks are in a program, and can be particularly useful on large programs with many components. The profiler generates a report that shows you a breakdown of which components of a program take up most time during execution. For each MATLAB function in a running program, profile records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time.

The profiler is controlled using the `profile` function. Refer to the help files for an explanation of the syntax and operation. To focus on function performance, GUI output from the program should be minimised. Try this example to see why:

```
profile on
for i = 1:1000
    figure(1); close all
    r = rand(100);
end
```



## profile viewer

**Profile Summary**

Generated 08-Oct-2014 10:16:20 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">close</a>	1000	4.214 s	0.096 s	
<a href="#">close&gt;request_close</a>	1000	2.983 s	0.432 s	
<a href="#">closereq</a>	1000	2.234 s	2.072 s	
<a href="#">close&gt;safegetchildren</a>	1000	1.100 s	0.439 s	
<a href="#">setdiff</a>	2000	0.632 s	0.082 s	

Figure 11.1: Example profiler output.

<b>Exercise 19. Measuring Performance</b> <ul style="list-style-type: none"> <li>• Stopwatch timing</li> <li>• Profiling</li> </ul>	
Key Functions	<b>tic, toc, profile</b>
<b>Task 1</b> Stopwatch timing	<b>Step 1</b> Open the solution to exercise 5 Fibonacci sequence and save a copy as performance.m for this exercise.
	<b>Step 2</b> Use the stopwatch timer functions (tic and toc) outside the loop in performance.m to measure the time it takes to calculate the required numbers of the Fibonacci sequence (in this case 20).
	<b>Step 3</b> Create an outer loop that in every iteration will calculate a different number of Fibonacci numbers, starting from 10 until 100 with a step of 5.
	<b>Step 4</b> Store the time results from toc in a results vector (using a new variable as a counter of the iterations in order to index the vector). Plot these time results against the number of Fibonacci numbers calculated in each case.
<b>Task 2</b> Profiling	<b>Step 1</b> Use the profiler (doc profile) to analyse the performance of the program and view the results.

## 11.2 Improving Performance

### Quick Tips

1. `load` and `save` are faster than low-level file I/O functions (see section 10).
2. Use logical indexing to quickly access non-contiguous matrix elements.
3. Don't change a variable's data type. When converting, create a new variable.

```
x = 10
```

```
x = num2str(x)
```

vs.

```
xs = num2str(x)
```

### 11.2.1 Preallocation

In section 4, we explored how matrices can be dynamically increased in size in MATLAB via concatenation, or out-of-bounds assignment. This can be useful when used in moderation, but repeated resizing carries a memory management cost and this can slow a program down. If it is possible to work out the maximum size a matrix will grow to during program execution in advance, then it is more efficient to allocate the required amount of space in advance. This is especially relevant in **for** loops, where the size of a variable is increased by a fixed amount at each repeat of the loop. Since the number of repeats in a **for** is known, it is better to pre-allocate the variable. Note the performance difference between these examples:

```
clear; tic

x = [];

for i = 1:10000

    x = [x,rand(10,1)];

end

toc

clear; tic

x = [];

for i = 1:10000

    x(:,i) = rand(10,1);

end

toc

clear; tic

x = zeros(10,10000);

for i = 1:1000

    x(:,i) = rand(10,1);

end

toc
```

### 11.2.2 Vectorization

In section 5, we mentioned that nested loops can be slow in MATLAB. However, MATLAB performs vector and matrix operations very efficiently using built-in (compiled) code and this leads us to *vectorization*. Vectorization means converting loops into equivalent vector or matrix operations, which can result in significant speed-ups that tend to increase with increasing code complexity. Here is a simple example:

```
clear; tic

i = 0;

for t = 0:.01:10

    i = i + 1;

    for j = 1:100

        y1(j,i) = sin(t)*j;

    end

end

toc

tic

t = 0:.01:10;

j2 = repmat((1:100)',1,length(t));

y2 = repmat(sin(t),100,1).*j2;

toc

numericalerror = max(max(y1-y2))
```

## 12 What Next?

We hope you have found this book useful. If you attended a taught session you will get an email with a link to a web page to give us anonymous feedback. We always value your feedback and use it to improve our sessions.

You may like to consider the following options to follow on from these sessions.

### 12.1 Computer

We encourage everyone to work at their own pace. This may mean that you don't manage to finish all of the exercises presented in this booklet. If you would like to complete the exercises while someone is on hand to help you, come along to one of the sessions that run during term time. More details are available at:

[www.it.ox.ac.uk/courses/](http://www.it.ox.ac.uk/courses/)

### 12.2 IT Services Help Centre

The IT Services Help Centre is open by appointment only, Monday to Friday. The Help Centre is also a good place to get advice about any aspect of using computer software or hardware. You can contact the Help Centre on 01865 (2) 12345 or by email: [help@it.ox.ac.uk](mailto:help@it.ox.ac.uk)

### 12.3 Books

There are many MATLAB textbooks available, from general introductory texts to application-specific guides. For a general introduction, try *Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers* (Rudra Pratap, 2010). For a quick reference, try *MATLAB Primer, 8th Edition* (Timothy A Davis, 2010). You can also find detailed explanations of MATLAB functions on the MathWorks website, <https://uk.mathworks.com/support/learn-with-matlab-tutorials.html>.

## **Helpful Summary Slides**

# The MATLAB Environment (Chapter 2)

**MATLAB = MATrix LABoratory**

High-level programming language and  
interactive environment for:

Algorithm Development

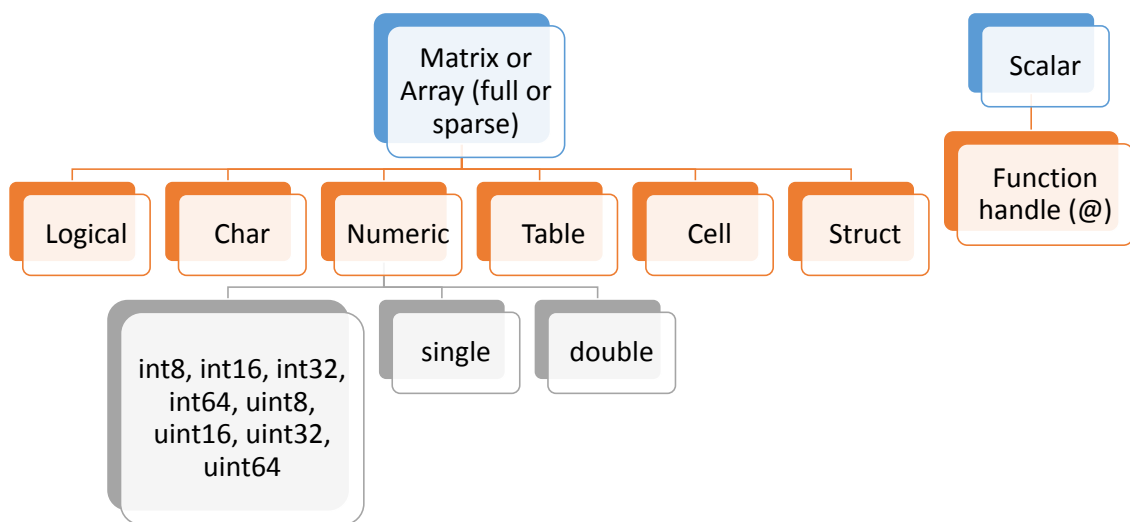
Data Analysis

Numerical Computations

Data Visualization

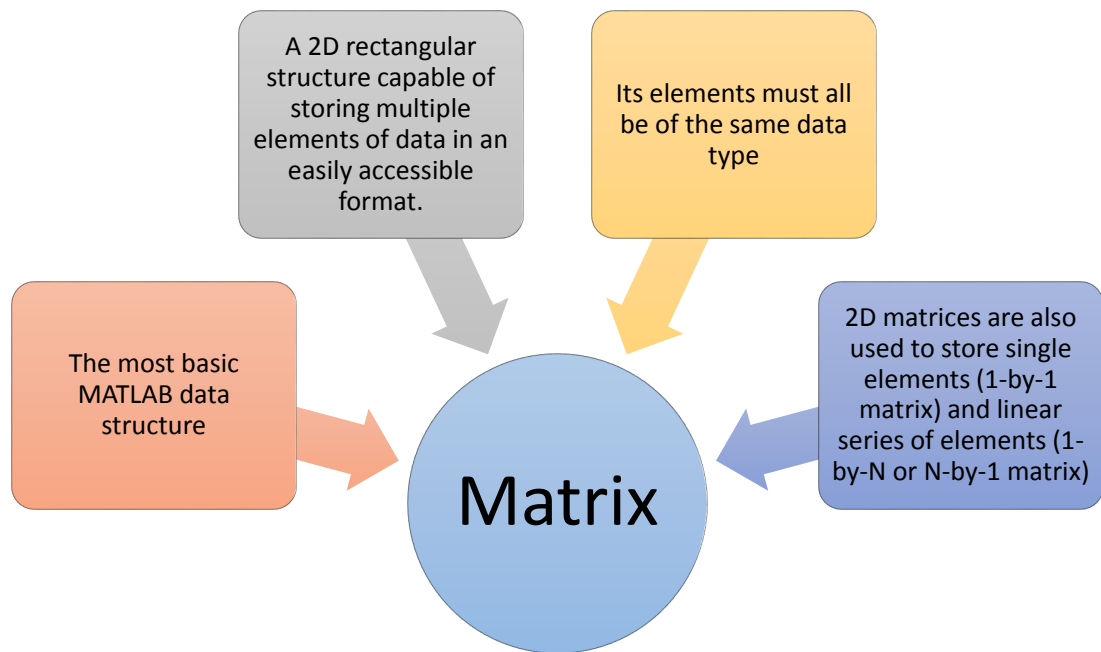
# Data types (classes) (Chapter 3)

There are many different data types or classes supported in MATLAB.





# Matrices (Chapter 4)



# Arithmetic Operators (Chapter 5)

Operator	Description
+ and -	Addition and subtraction. $A+B$ adds A and B. $A-B$ subtracts B from A.
.*	Array multiplication. $A.*B$ is the element-by-element product of the arrays A and B.
.^	Array power. $A.^B$ is the matrix with elements $A(i,j)$ to the $B(i,j)$ power.
./	Array right division. $A./B$ is the matrix with elements $A(i,j)/B(i,j)$ .
.\	Array left division. $A.\backslash B$ is the matrix with elements $B(i,j)/A(i,j)$ .
*	Matrix multiplication. $A*B$ is the linear algebraic product of A and B. For non-scalar A and B, the number of columns of A must equal the number of rows of B.
^	Matrix power. In the non-scalar power case, the calculation here involves eigenvalues and eigenvectors.
/	Matrix right division. $B/A = (A'\backslash B')'$
\	Matrix left division. $X = A\backslash B$ is the solution to the equation $AX=B$ .

# Relational Operators (Chapter 5)

Operator	Description
<	Less than
< =	Less than or equal to
>	Greater than
> =	Greater than or equal to
= =	Equal to
~ =	Not equal to

**Attention!** One equals symbol ( = ) is used for assignment.  
A pair of equals symbols ( = = ) is used for testing equality.

# Logical Operators (Chapter 5)

Operator	Description
&	Logical AND. A&B returns true (1) for every element location that is true (non-zero) in both arrays, and false (0) for all other elements.
	Logical OR. A B returns true (1) for every element location that is true (non-zero) in either one or the other, or both arrays, and false (0) for all other elements.
~	Logical NOT. Complements each element of the input array. Non-zeros become false while zeros become true.
&&	Short-circuit AND
	Short-circuit OR

For the short-circuit operators, if the outcome of the operation can be determined by the value of the first operand, the second is not evaluated.

# Loops (Chapter 5)

Loop control statements enable a code block to be **repeatedly** executed.  
There are two loop types in MATLAB: **for** and **while**.

- **for** loops: execute a code block a number of times determined on the entry to the loop. They can also be nested.
- **while** loops: repeat execution of a block of statements as long as a test expression is true.

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

```
while expression  
    statements  
end
```

- The statements inside the loop do not generally affect the number of loop repeats.
- To avoid infinitely-repeating while loops, one or more statements inside the loop must change the value of the test expression.