

KING'S COLLEGE LONDON

BENG BIOMEDICAL ENGINEERING

COMPUTER PROGRAMMING

---

## Laboratory Manual

---

Andrew KING

Sebastien ROUJOL

Jorge CARDOSO

Antonios POULIOPoulos

Elsa-Marie OTOO

2023-2024



*“When somebody has learned how to program a computer ...  
You’re joining a group of people who can do incredible things.  
They can make the computer do anything they can imagine.”*

**Sir Tim Berners-Lee**

# **Course Information**

## **Module Aims**

This module will provide the students with a good understanding of the fundamental concepts of computer programming. Practical skills will be learnt using the MATLAB software package, focusing on the procedural programming paradigm. The module should enable the students to tackle real-world biomedical engineering programming problems, by performing program design and making good use of programming language features in the implementation to produce elegant and efficient solutions. Laboratory sessions will encourage acquisition of practical problem-solving skills.

## **Learning Objectives**

On completion of the course the students should be able to:

- analyse problems and apply structured design methods to produce elegant and efficient program designs
- implement a program design making good use of MATLAB programming features, including control structures, functions and advanced data types

## **Suggested text books**

- MATLAB Programming for Biomedical Engineers and Scientists, 2nd Edition, Andrew P. King & Paul Aljabar, Academic Press, 2022
- MATLAB: A Practical Introduction to Programming and Problem Solving, Stormy Attaway, Butterworth-Heinemann, 2016

# Contents

<b>1</b>	<b>Introduction to Computer Programming and MATLAB</b>	<b>8</b>
1.1	Introduction . . . . .	8
1.1.1	Computer Programming . . . . .	8
1.1.2	MATLAB . . . . .	9
1.2	The MATLAB Environment . . . . .	10
1.3	Help . . . . .	12
1.4	Variables, Arrays and Simple Operations . . . . .	12
1.4.1	How Does MATLAB Display Numeric Values by Default?	16
1.5	Data Types . . . . .	18
1.6	Loading and Saving Data . . . . .	19
1.7	Visualising Data . . . . .	21
1.8	Matrices . . . . .	23
1.9	MATLAB Scripts . . . . .	24
1.10	Comments . . . . .	24
1.11	Debugging . . . . .	25
1.11.1	MATLAB Debugger . . . . .	25
1.11.2	MATLAB Code Analyser . . . . .	27
1.12	Summary . . . . .	28
1.13	Further Resources . . . . .	28
1.14	Exercises . . . . .	29
<b>2</b>	<b>Control Structures</b>	<b>34</b>
2.1	Introduction . . . . .	34
2.2	<code>if</code> Statements . . . . .	34
2.3	Comparison/Logical Operators . . . . .	36
2.4	<code>switch</code> Statements . . . . .	38
2.5	<code>for</code> Loops . . . . .	39
2.6	<code>while</code> Loops . . . . .	41
2.7	A Note about Efficiency . . . . .	43
2.8	<code>break</code> and <code>continue</code> . . . . .	44
2.9	Nesting Control Structures . . . . .	45
2.10	Summary . . . . .	46
2.11	Further Resources . . . . .	46
2.12	Exercises . . . . .	46
<b>3</b>	<b>Functions</b>	<b>54</b>

3.1	Introduction . . . . .	54
3.2	Functions . . . . .	54
3.3	Checking for Errors . . . . .	58
3.4	Function <i>m</i> -files and script <i>m</i> -files . . . . .	59
3.5	A function <i>m</i> -file can contain more than one function . . . . .	61
3.6	A script <i>m</i> -file CANNOT include functions . . . . .	63
3.7	<i>m</i> -files and the MATLAB Search Path . . . . .	64
3.8	Naming Rules . . . . .	64
3.9	Scope of Variables . . . . .	65
3.9.1	Be Careful about Script M-files and Scope . . . . .	67
3.10	Recursion: A function calling itself . . . . .	67
3.11	Summary . . . . .	70
3.12	Further Resources . . . . .	71
3.13	Exercises . . . . .	71
<b>4</b>	<b>Program Development and Testing</b>	<b>76</b>
4.1	Introduction . . . . .	76
4.2	Incremental Development . . . . .	76
4.3	Are we Finished? Validating User Input. . . . .	84
4.4	Debugging a Function . . . . .	85
4.5	Common Reasons for Errors when Running a Script or a Function . . . . .	91
4.6	Error Handling . . . . .	93
4.6.1	The <code>error</code> and <code>warning</code> Functions . . . . .	93
4.6.2	The <code>try</code> and <code>catch</code> Method . . . . .	94
4.7	Summary . . . . .	96
4.8	Further Resources . . . . .	96
4.9	Exercises . . . . .	97
<b>5</b>	<b>Data Types</b>	<b>103</b>
5.1	Introduction . . . . .	103
5.2	Numeric Types . . . . .	104
5.2.1	Precision for Non-integer (Floating Point) Numeric Types	105
5.2.2	MATLAB Defaults to Double Precision for Numbers .	106
5.2.3	Take Care when Working with Numeric Types other than Doubles . . . . .	108
5.2.4	Ranges of Numeric Types . . . . .	108
5.3	Infinity and NaN (Not a Number) . . . . .	110

5.4	Characters and Strings . . . . .	112
5.5	Identifying the Type of a Variable . . . . .	115
5.6	The Boolean Data Type . . . . .	117
5.7	Matrices . . . . .	118
5.8	Cell Arrays . . . . .	120
5.8.1	Cell Arrays can Contain Mixed Data Types . . . . .	122
5.8.2	The Different Kinds of Bracket: Recap . . . . .	123
5.9	Converting Between Types . . . . .	124
5.9.1	Converting Between a Number and a Character . . . . .	124
5.9.2	Converting Between a Number and a Logical Type . .	125
5.9.3	Converting Arrays . . . . .	126
5.10	Advanced Data Types . . . . .	129
5.10.1	Structures . . . . .	129
5.10.2	Maps . . . . .	131
5.11	Summary . . . . .	138
5.12	Further Resources . . . . .	139
5.13	Exercises . . . . .	139
<b>6</b>	<b>File Input/Output</b>	<b>147</b>
6.1	Introduction . . . . .	147
6.2	Recap on Basic Input/Output Functions . . . . .	147
6.3	Simple Functions for Dealing with Text Files . . . . .	147
6.4	Reading from Files . . . . .	149
6.5	Writing to Files . . . . .	158
6.6	Summary . . . . .	161
6.7	Further Resources . . . . .	161
6.8	Exercises . . . . .	161
<b>7</b>	<b>Program Design</b>	<b>166</b>
7.1	Introduction . . . . .	166
7.2	Top Down Design . . . . .	167
7.2.1	Incremental Development and Test Stubs . . . . .	172
7.3	Bottom-Up Design . . . . .	174
7.4	A Combined Approach . . . . .	175
7.5	Summary . . . . .	175
7.6	Further Resources . . . . .	176
7.7	Exercises . . . . .	176

<b>8 Visualisation</b>	<b>180</b>
8.1 Introduction . . . . .	180
8.2 Visualisation . . . . .	180
8.2.1 Visualising Multiple Datasets . . . . .	180
8.2.2 3-D Plotting . . . . .	185
8.2.3 The meshgrid Command . . . . .	188
8.2.4 Imaging Data . . . . .	189
8.3 Summary . . . . .	191
8.4 Further Resources . . . . .	191
8.5 Exercises . . . . .	192
<b>9 Code Efficiency</b>	<b>198</b>
9.1 Introduction . . . . .	198
9.2 Time and Memory Efficiency . . . . .	198
9.2.1 Timing Commands in MATLAB . . . . .	200
9.2.2 Assessing Memory Efficiency . . . . .	202
9.3 Tips for Improving Time Efficiency . . . . .	204
9.3.1 Pre-Allocating Arrays . . . . .	204
9.3.2 Avoiding Loops by using Built-In Functions and Vector Operations . . . . .	205
9.3.3 Logical Indexing . . . . .	208
9.3.4 A Few More Tips for Efficient Code . . . . .	211
9.4 Recursive and Dynamic Programming . . . . .	212
9.4.1 A Note on Recursive Functions . . . . .	216
9.5 Dynamic Programming to Improve Performance . . . . .	217
9.6 Summary . . . . .	219
9.7 Further Resources . . . . .	220
9.8 Exercises . . . . .	221
<b>10 Images and Image Processing</b>	<b>226</b>
10.1 Introduction . . . . .	226
10.2 Images on a Computer . . . . .	227
10.2.1 Colour Versus Grey Scale Images . . . . .	228
10.3 Accessing Images in MATLAB . . . . .	229
10.3.1 Accessing Information about an Image . . . . .	229
10.3.2 Viewing an Image . . . . .	230
10.3.3 Accessing the Pixel Data for an Image . . . . .	231
10.3.4 Viewing and Saving a Sub-Region of an Image . . . . .	232

10.4	Image Processing . . . . .	233
10.4.1	Binarising a Grey-Scale Image and Saving the Result . . . . .	233
10.4.2	Threshold-Based Operations . . . . .	234
10.4.3	Chaining Operations . . . . .	236
10.5	Summary . . . . .	237
10.6	Further Resources . . . . .	237
10.7	Exercises . . . . .	238
<b>11</b>	<b>Appendix</b>	<b>241</b>
11.1	Graphical User Interfaces . . . . .	241
11.1.1	Building a GUI with the Guide Tool . . . . .	242
11.1.2	Controlling Components: Callback Functions and Events	245
11.1.3	Maintaining State . . . . .	252
11.2	Summary . . . . .	254
11.3	Further Resources . . . . .	254
11.4	Exercises . . . . .	255
<b>12</b>	<b>Exercise Solutions</b>	<b>258</b>
12.1	Chapter 1 - Introduction to Computer Programming and MATLAB . . . . .	259
12.2	Chapter 2 - Control Structures . . . . .	267
12.3	Chapter 3 - Functions . . . . .	281
12.4	Chapter 4 - Program Development and Testing . . . . .	291
12.5	Chapter 5 - Data Types . . . . .	314
12.6	Chapter 6 - File Input/Output . . . . .	329
12.7	Chapter 7 - Program Design . . . . .	338
12.8	Chapter 8 - Visualisation and User Interfaces . . . . .	345
12.9	Chapter 9 - Code Efficiency . . . . .	353
12.10	Chapter 10 - Images and Image Processing . . . . .	367
12.11	Appendix - Graphical User Interfaces . . . . .	372
<b>Index</b>		<b>381</b>

# 1 Introduction to Computer Programming and MATLAB

*At the end of this section you should be able to:*

- *Describe the fundamental types of programming language and the difference between compiled and interpreted languages*
- *Use and manipulate the MATLAB environment*
- *Form simple expressions using scalars, arrays, variables, built-in functions and assignment in MATLAB*
- *Describe the different MATLAB data types and be able to determine the type of a MATLAB variable*
- *Perform simple input/output operations in MATLAB*
- *Perform basic data visualisation in MATLAB by plotting 2-D graphs*
- *Form and manipulate matrices in MATLAB*
- *Write MATLAB scripts and use the debugger and code analyser to identify and fix coding errors*

## 1.1 Introduction

In this first section of the module we will introduce some fundamental concepts about computer programming, and then focus on becoming familiar with the *MATLAB* software application, which we will be using throughout this module to learn how to program computers.

### 1.1.1 Computer Programming

The development of the modern digital computer in the 1950s marked a step change in the way that people used machines. Until then, mechanical and electrical machines had been proposed and built to perform *specific operations*, such as solving differential equations. In contrast, the modern digital computer is a *general purpose machine*, that is, it can perform any calculation that is computable. The way in which we tell it what to do is through computer programming.

We can make a number of distinctions between different computer programming languages depending on how we write and use the programs. Firstly, a useful distinction to make is between an *interpreted* language and a *compiled* language. With interpreted languages, after you write the program, you

can run (or *execute*) the program straight away. With compiled languages you have to perform an intermediate step, called *compilation*. Compiling a program simply means translating it into a language that the computer can understand (known as *machine code*). Once this has been done the computer can execute your program. With an interpreted language this translation is done in real-time as the program is executed.

In this module we will be learning how to program computers using a software application called *MATLAB*. MATLAB is (usually) an *interpreted* language. Although it is also possible to use it as a compiled language, we will be using it solely as an interpreted language.

Another distinction we can make between different programming languages relates to how the programs are written and what they consist of. Historically, most programming languages have been *procedural* languages. This means that programs consist of a sequence of instructions provided by the programmer. Until the 1990s almost all programming languages were procedural. More recently, another way of programming has gained in popularity, known as *object-oriented programming*. Object-oriented programming languages allow the programmer to break down the problem into *objects*: self-contained entities consisting of both data and operations on the data. You will learn about object-oriented programming in a later module in your BEng programme. Finally, some languages are *declarative*. In principle, when you write a declarative program it does not matter what order you write the program statements in - you are just ‘declaring’ something about the problem or how it can be solved. The compiler or interpreter will do the rest. Currently, declarative languages are mostly research tools.

MATLAB is (mostly) a procedural language. Although it does have some object-oriented features we will not cover them in this module.

### 1.1.2 MATLAB

MATLAB is a commercial software package for performing a range of mathematical operations. It was first developed in 1984 and included mostly linear algebra operations (hence the name, MATrix LABoratory) but has since been expanded to include a wide range of functionality including visualisation, statistics, and algorithm development using computer programming. It has approximately 1 million current users in academia, research and industry,

and has found particular application in the fields of engineering, science and economics. You will be using MATLAB in a number of courses in the BEng programme. In addition, many of the features of MATLAB may well be useful to you in developing biomedical engineering technology in the future.

As well as its core functionality, there are a number of specific toolboxes that are available to extend MATLAB's capabilities. You will make use of some of these toolboxes in other courses in the BEng programme (e.g. the Statistics toolbox in the Computational Statistics module).

Although MATLAB is a very powerful software package, it is a commercial product. Student licenses are relatively cheap ( $\sim £50$ ), but full licenses are quite expensive ( $\sim £1000$ ). If your institution is providing a licence this is not a problem for you, but if you need to acquire a license yourself you may want to consider one of the free, open-source alternatives to MATLAB. Some of the most common are:

- *Octave*: Good compatibility with MATLAB, but includes a subset of its functionality; available for Max/Linux/Windows (<http://www.gnu.org/software/octave>);
- *Freemat*: Good compatibility with MATLAB, but includes a subset of its functionality; available for Max/Linux/Windows (<http://freemat.sourceforge.net>);
- *Scilab*: Some differences to MATLAB, available for Max/Linux/Windows (<http://www.scilab.org/en>).

## 1.2 The MATLAB Environment

After you start up MATLAB, the first thing you will see is the MATLAB environment window. To begin with, it is important for you to spend a few minutes familiarising yourself with the different components of this window and how to use and manipulate them. Figure 1.1 shows a screenshot of the MATLAB environment window (if yours does not look exactly like this, don't worry - as we will see shortly it is possible to customise its appearance).

The *command window* is where you enter your MATLAB commands and view the responses. As you enter commands they will be added to your *command history*. You can always access and repeat old commands through the *command history* window. Another way of cycling through old commands

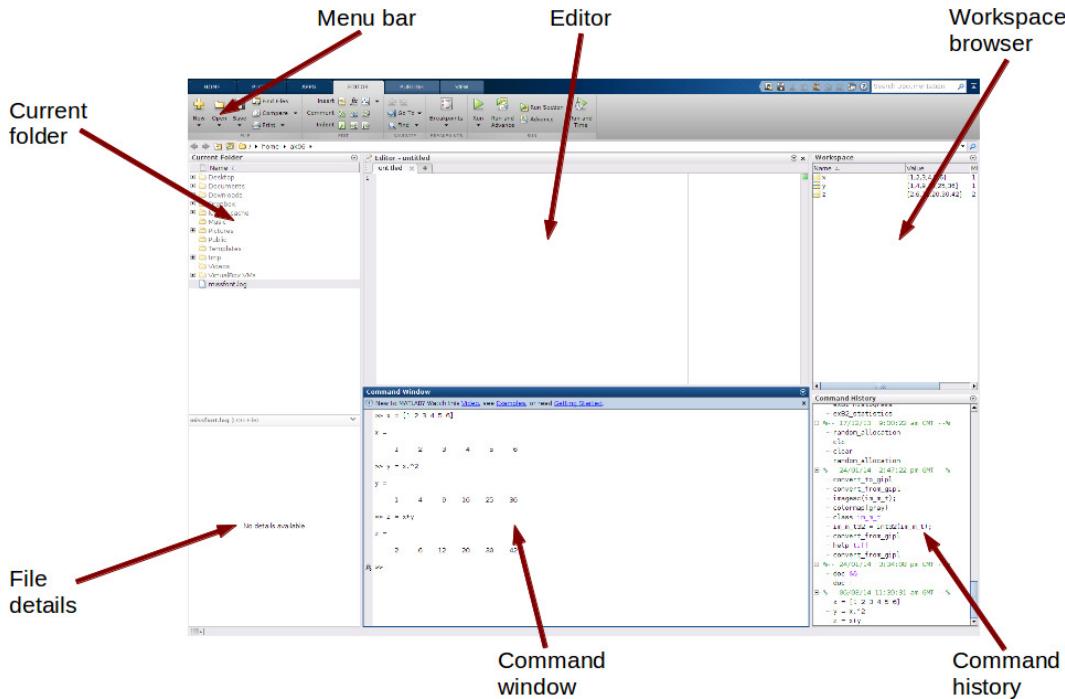


Figure 1.1: The MATLAB environment

in the *command window* is to use the up and down arrow keys. Although the *command window* is the main way in which you will get MATLAB to perform the operations you want, it is often also possible to execute commands via the *menu bar* at the top of the environment window.

As well as allowing you to enter commands the MATLAB environment window can also be used to navigate through your file system. The *current folder* window will always show your current folder and its contents. It can also be used for simple navigation tasks. If you select a file from the current folder contents its details will appear in the *file details* window.

As you use the *command window* to create and manipulate *variables*, these variables will be added to your *workspace*. The workspace is basically a collection of all of the data that has been created or loaded into MATLAB. The current contents of the workspace are always displayed in the *workspace browser*. You can also view the values of these variables from within the workspace browser.

When you start creating scripts to perform more complex MATLAB operations (see Section 1.9), you will use the *editor* window to create and edit your script files. This can be useful if you want to perform the same sequence of operations several times or save a list of operations for future use.

You can customise the MATLAB environment by clicking on the *Layout* button in the menu bar (under the *Home* tab).

### 1.3 Help

MATLAB includes extensive documentation and on-line help. At any time you can access this through the menu system (*Help* → *Documentation* under the *Home* tab) or via the command-line (type `doc` or `help` followed by the command you want information about).

### 1.4 Variables, Arrays and Simple Operations

Now we will start to type some simple commands in the command window. The first thing we will do is to create some *variables*. Variables can be thought of as named containers for holding some information, often numbers. They are called variables because the values (but not the names) can vary. For example, the following commands create two new variables called `a` and `y` with values 1 and 3 respectively.

```
a = 1  
y = 3
```

When you type these commands into the MATLAB command window you should see the new variables appear in the *workspace browser*. These variables can then be used in forming subsequent expressions for evaluation by MATLAB, e.g.

```
c = a + y  
d = a ^ y  
e = y / 2 - a * 4
```

Again, as you type these commands you will create new variables in the workspace, this time called `c`, `d` and `e` (the `^` symbol raises a number to the power of another number, e.g. `a^y` means  $a^y$ ).

MATLAB contains a large number of built-in functions for operating on numbers or variables. If you know which function you want to use you can just type it in the command window, e.g.

```
sin(pi/4)
z = tan(a)
```

The name `pi` is a built-in MATLAB variable that contains the value 3.14159... Alternatively, if you want to browse through all available functions, you can click on the `fx` symbol immediately to the left of the command prompt (see Figure 1.2). A menu will pop up allowing you to navigate through a range of categories of functions. For example, to find trigonometric functions choose *Mathematics*, *Elementary Math* and then *Trigonometric*.

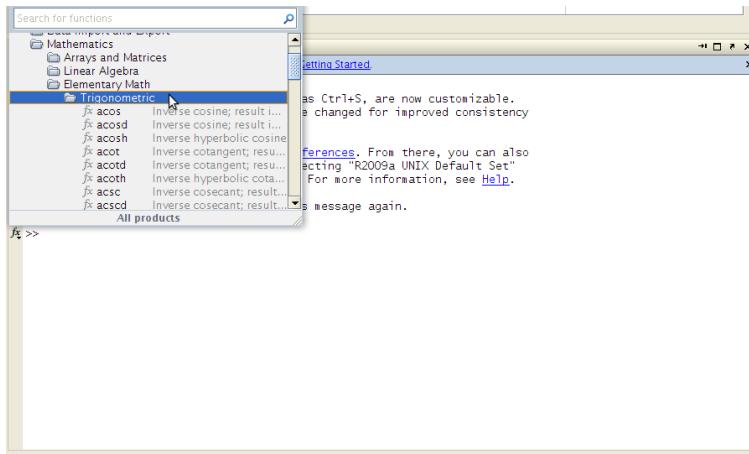


Figure 1.2: Using MATLAB's built-in functions

A list of commonly used mathematical functions that operate on numerical values is given in Table 1.1.

As well as containing single values, variables can also contain lists of values. In this case they are known as *arrays*. It is in the creation and manipulation of arrays that MATLAB can be particularly powerful. To illustrate this, consider the following example. First, we will create a sample array variable,

```
d = [1 3 4 2 5];
```

<code>sin(x)</code>	Sine (input in radians)	<code>log10(x)</code>	Base 10 logarithm
<code>cos(x)</code>	Cosine (input in radians)	<code>sqrt(x)</code>	Square root
<code>tan(x)</code>	Tangent (input in radians)	<code>abs(x)</code>	Absolute value
<code>asin(x)</code>	Arc sine (result in radians)	<code>floor(x)</code>	Round to next smallest integer
<code>acos(x)</code>	Arc cosine (result in radians)	<code>ceil(x)</code>	Round to next largest integer
<code>atan(x)</code>	Arc tangent (result in radians)	<code>round(x)</code>	Round to nearest integer
<code>exp(x)</code>	exponential of $x$ , i.e. $e^x$	<code>fix(x)</code>	Round to integer towards zero
<code>log(x)</code>	Natural logarithm	<code>mod(x,y)</code>	Modulus (remainder)

Table 1.1: A list of common MATLAB numerical functions

We define arrays by listing all values of the array, separated by spaces, enclosed by square brackets. Note that adding a semi-colon at the end of the command causes MATLAB to suppress its response to the command. The variable will still be created and added to the workspace but MATLAB won't display anything to the command window. Note also that by entering the above command we will overwrite our previous definition of  $d = a^y$ .

Now, suppose we wish to compute the value of each array element raised to the power of 3. We simply type the following command,

```
d.^3
```

If you perform an arithmetic operation on an array variable MATLAB will apply the operation to each element of the array. This can make processing of large amounts of data very easy. However, note the ‘.’ symbol before the caret (^). This tells MATLAB to apply the ^ operator to each element *individually*. If you don't include the ‘.’, some of the common arithmetic operators have special meanings when applied to arrays, as we will see in Section 5.7.

A short note here about the use of spaces in MATLAB: spaces are used to separate elements of arrays, as in the command `d = [1 3 4 2 5]` we saw above. At all other times spaces are ignored by MATLAB, e.g. the following two commands are identical:

```
d.^3
d .^ 3
```

MATLAB is, however, *case-sensitive*: the letters `d` and `D` represent different variables in the MATLAB workspace.

Sometimes you may wish to create a very large array with values that are regularly spaced. In this case there is a shorthand way to enter the values, as an alternative to typing in each value individually. The ‘:’ operator allows you to create an array by specifying its first value, the spacing between adjacent values and the last value. For example, try typing the following,

```
x = 0:0.25:3
```

and look at the resulting array. It consists of values separated by 0.25, starting with 0 and ending with 3. The colon operator is a much easier way to create arrays with very large numbers of elements.

An alternative way to create such regularly spaced arrays is to use the MATLAB `linspace` command. Consider the following code

```
x = linspace(0, 3, 13)
```

which has the same effect as the colon operator example shown above. The three arguments to `linspace` are the first value, the last value and the number of values in the array.

So far we have seen how to create arrays and to perform operations on the array elements. Often you will then want to access individual array elements. In MATLAB, this can be done using round brackets, e.g.

```
a = x(3);
```

This command accesses the third element of the array `x` and assigns it to the variable `a`. Similarly you can assign values to individual array elements without affecting the rest of the array,

```
x(3) = 10
```

We have already seen that MATLAB has a number of built-in functions for operating on numbers and numerical variables. There are also a number of built-in functions specifically for operating on arrays. A list of common MATLAB array operations is given in Table 1.2. For example, try typing the following,

```
a = [1 5 3 9 11 3 23 6 13 14 3 9];
```

<code>max(x)</code>	Maximum value	<code>cross(x,y)</code>	Vector cross product
<code>min(x)</code>	Minimum value	<code>dot(x,y)</code>	Vector dot product
<code>sum(x)</code>	Sum of all values	<code>disp(x)</code>	Display an array
<code>prod(x)</code>	Product of all values	<code>mean(x)</code>	Mean value of array
<code>length(x)</code>	Length of array	<code>std(x)</code>	Standard deviation of array

Table 1.2: A list of common MATLAB list functions

```
max(a)
min(a)
sum(a)
```

In addition to the array functions listed in Table 1.2, all of the numerical functions summarised in Table 1.1 can be used with array arguments. In this case, the function will be applied to each array element separately and the results combined into an array result. For example, try typing the following commands,

```
x = [10 3 2 7 9 12];
sqrt(x)
y = [-23 12 0 1 -1 43 -100];
abs(y)
```

#### 1.4.1 How Does MATLAB Display Numeric Values by Default?

We need to take some care when inspecting how MATLAB displays numbers on the command line as this can vary depending on the size of the number. By default, MATLAB tries to display numbers as compactly as possible in what is known as a short format. Asking for the value of  $\pi$  at the command line gives

```
>> pi
ans =
3.1416
```

This does not mean that MATLAB stores only the first four decimal places of  $\pi$ , it is simply only showing the value rounded to the *nearest* 4 decimal places.

We can request that MATLAB uses a longer format to display numbers by

typing `format('long')` at the command line.

```
>> format('long')
>> pi

ans =
3.141592653589793
```

We can revert to the default format by typing `format('short')`.

If numbers become large, then MATLAB uses a compact scientific notation. Recall that in scientific notation, a number is written in the form  $A \times 10^B$  where  $A$  is called the *mantissa* and  $B$  is called the *exponent*.

For example, the value of the factorial of 15 is around 130 billion, to be exact  $15! = 130,767,436,800$ . However, the default way this number is displayed in the MATLAB command window is as follows:

```
>> factorial(15)

ans =
1.3077e+12
```

Which is  $1.3077 \times 10^{12}$ . In other words, a scientific notation has been used which is a 4 d.p. approximation to the exact value in scientific notation (which would be  $1.307674368 \times 10^{12}$ ).

This way of displaying numeric values can make it a little tricky when we have an array containing both small and large values. For example, if we initialise the following array using three integers with very different sizes

```
>> x = [25 357346 2013826793]

x =
1.0e+09 *
0.0000    0.0004    2.0138
```

MATLAB displays the resulting array because we did not use a semi-colon to suppress the output. It tries to show all three numbers in a scientific notation which uses a single exponent. In this case the exponent is 9 which is determined by the largest number in the array. If we were to write all three numbers *exactly* using this exponent, then the three numbers would appear as

$$0.000000025 \times 10^9 \quad 0.000357346 \times 10^9 \quad 2.013826793 \times 10^9$$

MATLAB tries to shorten this to a single array multiplied by the common power of ten:

$$10^9 \times [0.000000025 \quad 0.000357346 \quad 2.013826793]$$

Next, because it uses short notation, each of the mantissa numbers in the array are rounded to their nearest four decimal places. This is why the array appears in MATLAB as

```
x =
1.0e+09 *
0.0000    0.0004    2.0138
```

This does not say that the first number in the array is zero, just that when a large exponent is used ( $10^9$  or in MATLAB notation  $1.0\text{e}+09$ ), the mantissa to multiply is so small that it just appears to be zero.

## 1.5 Data Types

So far, all variables and values we have used have been numeric. Although MATLAB is mostly used for numerical calculations, it is possible, and sometimes very useful, to manipulate data of other types. MATLAB allows the following basic data types (the name(s) in brackets is/are the MATLAB term(s) for the type):

- Floating point, e.g. 1.234 (`single`, `double` depending on the precision required)
- Integer, e.g. 3 (`int8`, `int16`, `int32`, `int64` depending on the number of bits to be used to store the number)

- Character, e.g. ‘a’ (char)
- Boolean, e.g. true (logical)

Other, more advanced types are also possible but we won’t discuss them just yet. (We’ll return to the concept of data types in Section 5.). Note that numeric values can be either floating point or integer types. In fact, unless you specify otherwise, all numeric values will be double floating point. You can always find out the type of a MATLAB variable(s) using the `whos` command. If you just type `whos` on its own it will list the types of all variables in the workspace. If you add a list of variable names it will only display the types of those listed. For example, try typing the following:

```
a = 1
b = 'x'
c = false
whos
d = [1 2 3 4]
e = ['a' 'b' 'c' 'd']
f = ['abcd']
whos d e f
```

Note that a character type is indicated by enclosing the character in single quotes (double quotes are used for something else - see Section 5.4). Note also that the definitions of the variables `e` and `f` are equivalent: the form used in assigning to `f` is just a short-hand way for that used in assigning to `e`. We will see a practical use for arrays of characters in the next section.

## 1.6 Loading and Saving Data

When using MATLAB for biomedical problems, you may be working with large amounts of data that are saved in an external file. Therefore, it is important to know how to load data from, and save data to, external files.

When writing data items to a text file, we normally need a special character called a *delimiter* that is used to separate one data item from the next. We can save data delimited in this way with the `writematrix` command as illustrated in the following example,

```
>> f = [1 2 3 4 9 8 7 6 5];
>> writematrix(f, 'test.txt');
```

Here, the two arguments of `writematrix` specify the variable containing the data to be saved and the file name (a character array). By default, `writematrix` uses commas as the delimiters (i.e. the file will be a comma-delimited text file). We can tell MATLAB to use a different delimiter as shown in the code below.

```
>> writematrix(f, 'testtab.txt', 'Delimiter', 'tab');
>> writematrix(f, 'testspace.txt', 'Delimiter', '');
```

In these commands the delimiters are a *tab* and a space character. Note that the delimiters specified in this way are used to separate the different *columns* of the array. MATLAB will always use a ‘new line’ character to separate *rows* when writing matrices (i.e. 2-D arrays, see Section 1.8).

The `load` and `save` commands are probably the most commonly used commands for reading and writing data. The following examples illustrate the use of `load` and `save`,

```
d = load('test.txt');
e = d / 2;
save('newdata.mat', 'd', 'e');
save('alldata.mat');
clear
load('newdata.mat')
```

When using the `save` command the first argument specifies the file name and any subsequent arguments specify which variable(s) to save to the file. Omitting the variable name(s) will cause the entire workspace to be saved. `save` will create a binary file containing the variables and their values that can be loaded at any time in the future. These binary files are called *MAT* files and it is common to give them the file extension “.mat”. The `load` command can be used to read either text files created by `writematrix` or binary files created by `save`. Note that when using the `load` command with a text file you should assign the result of the `load` into a variable for future use. Otherwise, it will simply be displayed at the command window. The `clear` command will remove all current variables from the workspace.

An alternative, and easier, way of loading *MAT* files into the workspace is to simply use the MATLAB environment to drag-and-drop the appropriate file icon from the current folder window to the workspace browser (or just

double-click on it in the current folder window).

Finally, the *import data* wizard can often be the quickest and easiest way to read data from text files (i.e. not *MAT* files). Selecting the *Import Data* button on the *Home* tab will bring up a wizard in which you can choose the file, and interactively specify such parameters as delimiters and the number of header lines in the file.

We will look again in more detail at the subject of MATLAB file input/output functions in Section 6.

## 1.7 Visualising Data

As well as entering and processing data, MATLAB can also be used for visualising data. The `plot` command is used for simple data visualisation. Consider the following code,

```
x = 0:0.1:2*pi;
y = sin(x);
plot(x,y, '-b');
```

This should produce the output shown in Figure 1.3a. The first line of this code uses the colon operator to create an array called `x` starting with the value 0, and extending up to  $2\pi$  in steps of 0.1. This array acts as the  $x$ -axis of the plot. The second line applies the built-in MATLAB `sin` function to every element of `x`. The resulting array acts as the  $y$ -axis of the plot. Finally, the third line of code displays the graph using the `plot` command. The first two arguments are the  $x$  and  $y$  axis data (which must be of the same length). The third argument specifies the appearance of the line. Table 1.3 summarises the different symbols that can be used to control the appearance of the line and the markers. All of these symbols can be included in the third character array argument to the `plot` command. Try experimenting with some of these to see the effect they have.

Next, we will add some text to our plot. Try typing the following commands,

```
title('Plot of sine wave from 0 to 2\pi');
xlabel('Angle in radians');
ylabel('Sine');
```

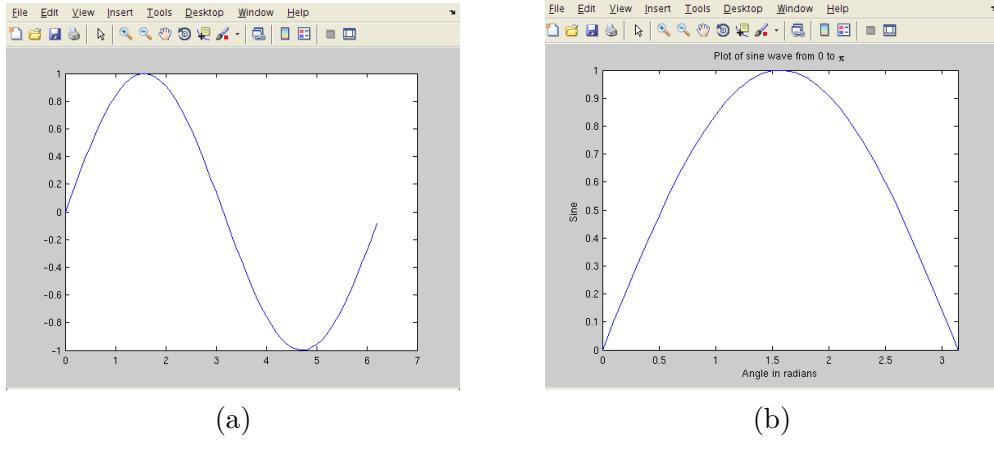


Figure 1.3: Plots of a sine wave between (a) 0 and  $2\pi$ ; (b) 0 and  $\pi$ .

<u>Marker</u>	<u>Line</u>	<u>Colour</u>
.	— Solid line	k Black
x	-- Dashed line	b Blue
o	:	r Red
d	-. Dash-dot line	g Green
s		m Magenta
+		c Cyan
*		y Yellow

Table 1.3: Specifying the appearance of the markers and line in MATLAB plots

These add a title and labels for the  $x$  and  $y$  axes respectively.

Now suppose we only wish to visualise the plot for values of  $x$  between 0 and  $\pi$ . The `axis` command can be used to specify minimum/maximum values for the  $x$  and  $y$  axes,

```
axis([0 pi 0 1]);
```

This should produce the output shown in Figure 1.3b, i.e. the same plot but zoomed to the  $x$  range from 0 to  $\pi$ , and the  $y$  range from 0 to 1.

Sometimes you may want to display two or more curves on the same plot. This is possible with the `plot` command as the following code illustrates,

```
y2 = cos(x);
```

```

plot(x,y,'-b', x,y2,'--r');
legend('Sine','Cosine');

```

You can display multiple curves by just concatenating sets of the three arguments in the argument list, i.e.  $x$  data,  $y$  data, line/marker style. The `legend` command adds a legend to identify the different curves. The `legend` command should have one character array argument for each curve plotted.

We will return to the subject of data visualisation in Section 8, and discuss more ways of visualising multiple datasets in Section 8.2.1.

## 1.8 Matrices

A MATLAB array is a *one-dimensional* (1-D) list of data elements. *Matrices* can also be defined, which are *two-dimensional* (2-D) arrays,

```

a = [1 2; 3 4];
b = [2, 4;
      1, 3];

```

Here, note that matrices are defined as rows separated by semi-colons. The row elements can be delimited by either spaces or commas. Note also that you can press <RETURN> to move to a new line in the middle of defining an array. MATLAB will not process the array definition until you have closed the array with the ‘`]`’ character.

MATLAB matrices can be accessed in the same way as arrays, except that two arguments need to be specified (i.e. the row followed by the column),

```

a(1,2)
b(2,2)

```

If you want to access an entire row or column of a matrix you can use the colon operator,

```

a(:,2)
b(1,:)

```

MATLAB also features a number of built-in functions specifically intended for use with matrices, and these are summarised in Table 1.4.

<code>eye (n)</code>	Create n by n identity matrix
<code>zeros (m, n)</code>	Create m by n matrix of zeros
<code>ones (m, n)</code>	Create m by n matrix of ones
<code>rand (m, n)</code>	Create m by n matrix of random numbers in range [0-1]
<code>size (a)</code>	Return the number of rows/columns in the matrix a

Table 1.4: MATLAB built-in matrix functions.

Finally, defining matrices allows you to perform a range of different linear algebra operations in MATLAB, and we will return to this topic in Section 5.7.

## 1.9 MATLAB Scripts

Now that you are starting to write slightly longer pieces of MATLAB code, you will probably find it useful to be able to save sequences of commands for later reuse. MATLAB script *m*-files are the way to do this. To create a new MATLAB script file you can use the *editor* window. First, click on the *New* button (under the *Home* tab) and choose *Script*. This will create a blank file. You can now type your MATLAB commands into this file. To save the file, click on the *Save* button,  (under the *Editor* tab). The first time that you save the file you will be prompted for a file name. Note that it is common practice to save MATLAB script files with the file extension “.m”. To run the commands in your script, click on the *run* icon, .

**Example 1.1.** For example, try entering the following command sequence into a MATLAB script file, save it with the file name *test.m* and then run the script.

```
x = 1:0.1:10;
y = 10*x - x.^2;
plot(x, y, 'ob');
```

What does this script do?

## 1.10 Comments

When writing MATLAB scripts it is possible to add *comments* to your code. A comment is a piece of text that will be ignored by MATLAB when run-

ning your script. Comments in MATLAB are specified by the ‘%’ symbol: any text after a ‘%’ symbol on any line of a script will not be interpreted by MATLAB. It is good practice to add comments to your code to explain what the code is doing. This is useful if you or other people need to look at your code in the future with a view to modifying it or rectifying errors.

**Example 1.2.** For example, let’s add some comments to the script we looked at in Example 1.1 to see how it improves code readability.

```
% define x-axis values
x = 1:0.1:10;

% define y-axis values
y = 10*x - x.^2;

% produce plot of 10x-x^2
plot(x, y, 'ob');
```

## 1.11 Debugging

Unfortunately, often the programs you write will not work the first time you run them. If they have errors, or *bugs*, in them, you must first *debug* them. MATLAB has two useful features to help you in the debugging process.

### 1.11.1 MATLAB Debugger

MATLAB has a built-in debugger to help you identify and fix bugs in your code. One of the most useful features of the MATLAB debugger is the ability to set *breakpoints*. A breakpoint at a particular line in a program will cause execution of the program to automatically stop at that line. You can then inspect the values of variables to check that they are as you expect, and step through the code line by line. To set a breakpoint on a particular line, in the *editor* window left-click once in the grey vertical bar just to the right of the line number. To confirm that the breakpoint has been created MATLAB will display a small red circle (●) in the grey bar (see Figure 1.4, line 2). Now, when you click on the *run* icon to run the script, execution will stop at the line with the breakpoint. This fact will be indicated by a green arrow just to the right of the breakpoint: ➔ (see Figure 1.4).

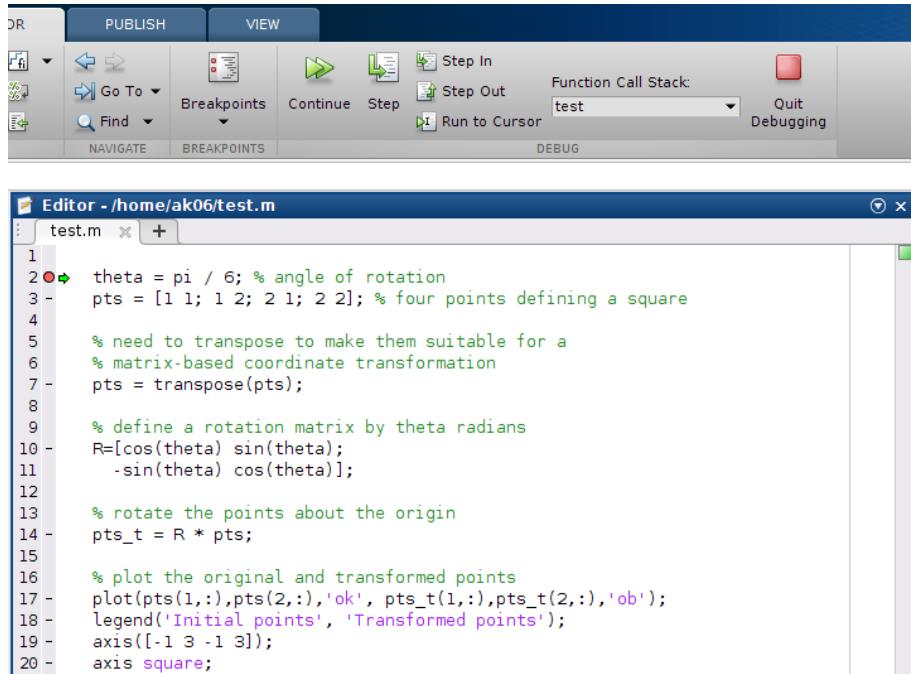


Figure 1.4: The MATLAB debugger

Once execution has been stopped, you are free to inspect variable values as you normally would in MATLAB, i.e. by typing at the command window or by looking at the workspace browser. Once you are ready to continue executing the program, there are a number of options, each available from an icon in the *Debug* section of the menu under the *Editor* tab:

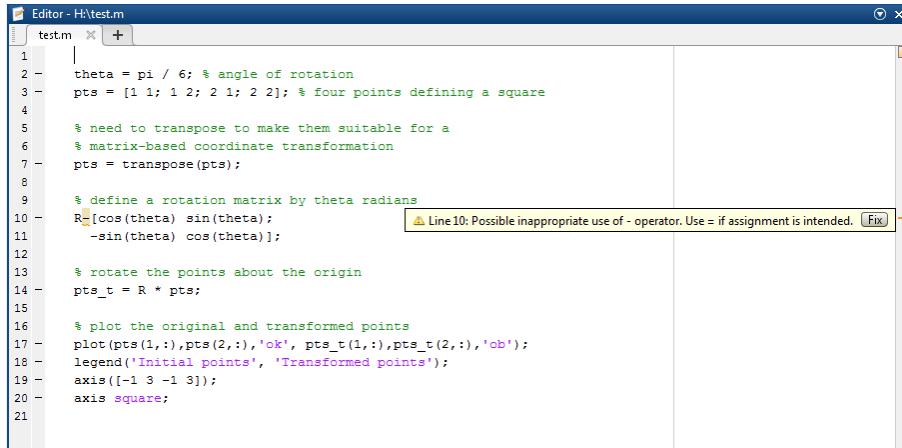
- *Continue*: Execute the program from the breakpoint until program termination
- *Step*: Execute the current line of code, then pause again
- *Step in*: If the current line is a function, step into that function
- *Step out*: If we are currently inside a function, step out of the function
- *Run to cursor*: Execute the program until the line where the cursor is positioned is reached, then pause
- *Quit debugging*: Exit the debugger

### 1.11.2 MATLAB Code Analyser

The second MATLAB feature that can be useful in the debugging process is the code analyser. The code analyser can be used to automatically check your code for possible coding problems. It can be run in one of two ways: either on demand by requesting a code analyser report, or in real time in the editor window.

Your MATLAB installation is probably set up to analyse your code in real time (i.e. whenever you save a script file in the editor window). To check-/change this preference click on *Preferences* under the *Home* tab, and select *MATLAB → Code Analyzer*. Then, either check or uncheck the option labelled *Enable integrated warning and error messages*.

You can see an example of the real time code analyser in Figure 1.5. A potential error has been spotted on line 10 (a minus sign instead of an assignment) and highlighted. A small orange horizontal bar to the right of the text window indicates a warning (a red bar would indicate an error). When the user hovers their mouse pointer over this bar a message pops up, in this case warning of a “possible inappropriate use of - operator”. Clicking on the *Fix* button will perform the fix recommended by the code analyser, in this case to replace the ‘-’ with a ‘=’.



A screenshot of the MATLAB Editor window titled "Editor - H:\test.m". The code in the editor is:

```
1 theta = pi / 6; % angle of rotation
2 pts = [1 1; 1 2; 2 1; 2 2]; % four points defining a square
3
4 % need to transpose to make them suitable for a
5 % matrix-based coordinate transformation
6 pts = transpose(pts);
7
8 % define a rotation matrix by theta radians
9 R=[cos(theta) sin(theta);
10 -sin(theta) cos(theta)]; Line 10: Possible inappropriate use of - operator. Use = if assignment is intended. [Fix]
11
12 % rotate the points about the origin
13 pts_t = R * pts;
14
15 % plot the original and transformed points
16 plot(pts(1,:),pts(2,:),'ok', pts_t(1,:),pts_t(2,:),'ob');
17 legend('Initial points', 'Transformed points');
18 axis([-1 3 -1 3]);
19 axis square;
```

The line 10 is highlighted with an orange background. A tooltip appears above the line 10 code with the text "Line 10: Possible inappropriate use of - operator. Use = if assignment is intended. [Fix]".

Figure 1.5: The MATLAB code analyser

Alternatively, if you don’t want such real time warning and error messages, you can change the preference described above, and just request a code anal-

yser report whenever you want one. To do this, click on the  symbol at the top right of the editor window, and select *Show Code Analyzer Report*.

We will return to the important topic of code debugging in Section 4.

## 1.12 Summary

Learning how to program a computer opens up a huge range of possibilities: in principle, you can program the computer to perform any computable operation. Programming languages are the languages which we use to tell the computer what to do. There are several ways of distinguishing between different types of programming language, e.g. interpreted/compiled, procedural/object-oriented/declarative. In this module you will learn procedural programming using MATLAB, which is normally used as an interpreted programming language.

As well as being a programming language, MATLAB is a powerful commercial software package that implements a wide range of mathematical operations. It can be used to assign values to variables, which can be of different types such as numbers, characters, Booleans or one-dimensional or two-dimensional arrays (matrices). Basic arithmetic operations and a large number of built-in functions can be applied to values as well as to arrays and matrices. MATLAB can also be used to visualise data by plotting one array of values against another.

MATLAB *m*-file scripts allow you to save a sequence of MATLAB commands for future use. It is a good idea to add comments to your scripts to make them easier to understand. The MATLAB debugger and code analyser can be used to track down and fix bugs in your code.

## 1.13 Further Resources

- The MATLAB documentation contains extensive information and examples about all commands and built-in functions. Just type `doc` followed by the command/function name at the command line. This documentation is also available on-line at the Mathworks web-site.
- The Mathworks web-site contains a number of useful video tutorials: [www.mathworks.co.uk/academia/student\\_center/tutorials/launchpad.html](http://www.mathworks.co.uk/academia/student_center/tutorials/launchpad.html).

- The File Exchange section of the same web-site can be useful for downloading functions and MATLAB code that has been shared by others: [www.mathworks.com/matlabcentral/fileexchange](http://www.mathworks.com/matlabcentral/fileexchange).

## 1.14 Exercises

1. First, write a command to clear the MATLAB workspace. Then, create the following variables:

- (a) A variable called `a` which has the character value ‘q’
- (b) A variable called `b` which has the Boolean value true
- (c) A variable called `c` which is an array of integers between 1 and 10
- (d) A variable called `d` which is an array of characters with the values ‘h’, ‘e’, ‘l’, ‘l’, ‘o’

Now find out the data types of all variables you just created.

2. Create a list of all even integers between 32 and 55 and save it to a text file called `evens.txt`. Use the comma character as the delimiter.
3. Let `x = [5 2 4 7 9 1]`.
  - (a) Add 3 to each element.
  - (b) Add 4 to just the even-indexed elements.
  - (c) Create a new array, `y`, with each element containing the square root of the corresponding element of `x`.
  - (d) Create a new array, `z`, with each element containing the cube of the corresponding element of `x`.
4. A class of students have had their heights in centimetres measured as: 159, 185, 170, 169, 163, 180, 177, 172, 168 and 175. The same students’ weights in kilograms are: 59, 88, 75, 77, 68, 90, 93, 76, 70 and 82. Use MATLAB to compute the mean and standard deviation of the students’ height and weight.
5. Let `a = [1 3 1 2]` and `b = [7 10 3 11]`.
  - (a) Sum all elements in `a` and add the result to each element of `b`.

- (b) Raise each element of  $b$  to the power of the corresponding element of  $a$ .
  - (c) Divide each element of  $b$  by 4.
6. Write MATLAB commands to compute the *sine*, *cosine* and *tangent* of an array of numbers between 0 and  $2\pi$  (in steps of 0.1). Save the original input array and all three output arrays to a single *MAT* file and then clear the workspace.
7. Write a MATLAB *m*-file to read in a user's height (in metres) and weight (in kilograms) from a user, then compute and display the user's body mass index (BMI), according to the formula

$$\text{BMI} = \text{mass}/\text{height}^2.$$

(Hint: look at the MATLAB documentation for the *input* command.)

8. Write a MATLAB *m*-file to read a floating point number from the user, which represents a radius  $r$ , and then compute and print the surface area and volume of a sphere of that radius, according to the following equations:

$$\text{Surface Area} = 4\pi r^2, \quad \text{Volume} = \frac{4}{3}\pi r^3$$

9. Write a MATLAB *m*-file to read in three floating point values from the user ( $a$ ,  $b$  and  $c$ ), which represent the lengths of three sides of a triangle. Then compute the area of the triangle according to the equation:

$$\text{Area} = \sqrt{s(s - a)(s - b)(s - c)}$$

where  $s$  is half of the sum of the three sides.

10. Let  $m = [1 \ 2 \ 3 \ 4 \ 5 \ 6]$  and  $n = [10 \ 1 \ 8 \ 5 \ 3 \ 0]$ . Plot  $m$  against  $n$ .
11. Write a MATLAB *m*-file to plot curves of the *sine*, *cosine* and *tangent* functions between 0 and  $2\pi$ . Your script should display all three curves on the same graph and annotate the plot. Limit the range of the  $y$ -axis to between -2 to 2.
12. Write a single command that will create a  $3 \times 4$  matrix in which all elements have the value -3.

13. Write a single command that will create a 4x4 matrix containing all zeros apart from 3s on the diagonal.
14. Write a MATLAB *m*-file to define the following matrices:

$$X = \begin{pmatrix} 4 & 2 & 4 \\ 7 & 6 & 3 \end{pmatrix}, Y = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \end{pmatrix}, Z = \begin{pmatrix} 2 & 0 & 9 \\ 1 & 5 & 7 \end{pmatrix}.$$

Now perform the following operations:

- (a) Replace the coefficient in the second row and second column of  $Z$  with 1
- (b) Replace the first row of  $X$  with [2 2 2]
- (c) Replace the first row of  $Y$  with the second row of  $Z$
15. The injury severity score (ISS) is a medical score to assess trauma severity. Data on ISS and hospital stay (in days) have been collected from a number of patients who were admitted to hospital after accidents. The ISS data are [64 35 50 46 59 41 27 39 66], and the length of stay data are [8 2 5 5 4 3 1 4 6]. Use MATLAB to plot the relationship between ISS and hospital stay.  
If you select *Tools* → *Basic Fitting* in the MATLAB figure window it is possible to fit regression lines between datasets. Using the Basic Fitting window, fit a linear regression line to these data and estimate what length of hospital stay would be expected for a patient with an ISS of 55.
16. The MATLAB script shown below is intended to read a matrix from a text file and display how many coefficients the matrix has. However, the code does not currently work correctly. Use the MATLAB code analyser and debugger to identify the bug(s) and fix the code.

```
clear
m = load('m.txt');
s = size(m);
n = s^2;
disp(['The matrix has ' num2str(n) ' elements']);
```

(This file *debug\_exercise.m* and the test file *m.txt* are available to you

from within KEATS.)

17. Data has been collected of the ages and diastolic/systolic blood pressures of a group of patients who are taking part in a clinical trial. All data is available to you through the KEATS system in the files *age.txt*, *dbp.txt* and *sbp.txt*. Write a MATLAB *m*-file that reads in the data and creates a plot containing two subplots: one of age against systolic blood pressure and one of age against diastolic blood pressure. Your script should annotate both subplots.

*(Hint: look at the MATLAB documentation for the subplot command.)*

18. A short way of determining if an integer is divisible by 9 is to sum the digits of the number: if the sum is equal to 9 then the original number is also divisible by 9. For integers up to 90 this requires only a single application of this rule. Write MATLAB commands to verify the rule, i.e. create an array of all multiples of 9 between 9 and 90, and compute the sums of their two digits.

*(Hint: look at the MATLAB documentation for the idivide and mod commands. These can be used to find the result and remainder after integer division, so if you use them with a denominator of 10 they will return the two digits of a decimal number. Because these two commands only work with integer values you will also need to know how to convert from floating point numbers to integers: type doc int16.)*

## Notes

## 2 Control Structures

At the end of this section you should be able to:

- Write conditional MATLAB statements using `if-else` and `switch` statements
- Form complex logical expressions in MATLAB and explain the rules for operator precedence
- Write iterative MATLAB statements using `for` and `while` loops
- Select an appropriate iterative statement for a given programming situation

### 2.1 Introduction

In the previous section we introduced the fundamental concepts of computer programming, as well as the basics of the MATLAB software package. In particular, we saw how to create, save and run MATLAB scripts, or *m*-files. Scripts allow us to save sequences of commands for future use and can be a very useful time-saving device. MATLAB scripts become an even more powerful tool once you learn how to do some simple computer programming. Computer programming involves executing a number of commands but controlling the sequence of execution (or *control flow*) using special programming constructs. In this section we introduce some of the programming constructs provided by MATLAB.

### 2.2 `if` Statements

The normal flow of control in a MATLAB script, and procedural programming in general, is *sequential*, i.e. each program statement is executed in sequence, one after the other. This is illustrated in Figure 2.1.



Figure 2.1: The normal sequential control flow in procedural programming

If all of your programs featured only sequential control flow you would be quite limited in what you could achieve. To write more complex programs you need to make use of programming constructs to alter this normal control flow. One type of programming construct is the *conditional* statement. The

control flow of a conditional statement is illustrated in Figure 2.2. In this example there are two possible paths through the program, involving execution of different statements. Which path is taken depends on the result of a *condition*.

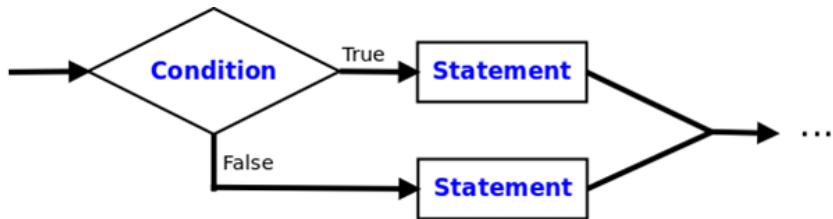


Figure 2.2: The control flow of a conditional statement

**Example 2.1.** MATLAB `if` statements are one way of achieving a conditional control flow. The use of an `if` statement is illustrated in the example below.

```

a = input('Enter a number:');
if (a >= 0)
    root = sqrt(a);
    disp(['Square root = ' num2str(root)]);
else
    disp(['Number is -ve, there is no square root']);
end
  
```

Try entering these statements and saving them as a MATLAB *m*-file. Examine the code to make sure you understand what it does (recall that the `disp` statement displays an array to the command window - in this case an array of characters). You can even step through the code using the MATLAB debugger. Pay particular attention to the alternative *paths of execution* that the program can take depending on the result of the comparison ( $a \geq 0$ ). This comparison corresponds to the *condition* box in Figure 2.2. The two statements after the `if` will be executed only if the condition is true. The statement after the `else` will be executed if the condition is false. After one of these two paths has been taken control will resume immediately after the `end` statement. Note that the `if` and `end` statements are both compulsory and should always come in pairs. The `else` statement is optional and if excluded the program execution path will jump to after the `end` statement

Relational	Logical
<code>==</code> Equals to	<code>~</code> NOT
<code>~=</code> Not equals to	<code>&amp;&amp;</code> AND
<code>&gt;</code> Greater than	<code>  </code> OR
<code>&lt;</code> Less than	
<code>&gt;=</code> Greater than or equal to	
<code>&lt;=</code> Less than or equal to	

Table 2.1: Common relational and logical operators

if the condition is false.

### 2.3 Comparison/Logical Operators

The previous example introduced the concept of a *condition*. Conditions commonly involve comparisons between expressions involving variables and values. For example, we saw the `>=` comparison operator in the example. A list of common comparison (or *relational*) operators such as `>=` is included in Table 2.1, along with common *logical* operators for combining the results of different comparisons.

**Example 2.2.** The use of these relational and logical operators is illustrated in the following code excerpt.

```
...
at_risk = false;
if (gender == 'm') && (calories > 2500)
    at_risk = true;
end
if (gender == 'f') && (calories > 2000)
    at_risk = true;
end
...
```

Here, in both `if` statements, the Boolean `at_risk` variable is set to true only if both comparisons evaluate to true, e.g. if `gender` is equal to ‘m’ and `calories` is greater than 2500. If either comparison evaluates to false the flow of execution will pass to after the corresponding `end` statement.

Note that there were round brackets around the comparisons in the above example. These are not essential, but they may change the meaning of the

Precedence	Operator(s)
1	Brackets ()
2	Matrix transpose ', power .^, matrix power ^
3	Unary plus +, unary minus -, logical not ~
4	Element-wise multiplication .* , element-wise division ./, matrix multiplication *, matrix division /
5	Addition +, subtraction -
6	Colon operator :
7	Less than <, less than or equal to <=, greater than >, greater than or equal to >=, equal to ==, not equal to ~=
8	Logical and &&
9	Logical or

Table 2.2: MATLAB operator precedence

condition. For example, suppose the brackets were not included, i.e.

```
...
if gender == 'm' && calories > 2500
...
```

There are three different operators in this condition: ==, && and >. In what order would MATLAB evaluate them? The answer to this question lies in the rules for *operator precedence*. Whenever MATLAB needs to evaluate an expression containing multiple operators it will use the order of precedence shown in Table 2.2. You should have come across all of these operators before with a few exceptions. The ' operator, when placed after a matrix, is just a short way of calling the transpose function, which we will discuss in Section 5.7. The *unary* plus and minus operators refer to a + or - sign being placed before a value or variable to indicate its sign.

Now let's return to our condition without brackets. Of the three operators present (==, && and >), we can see from Table 2.2 that the ones with the highest precedence are == and >. Therefore, these will be evaluated first. As they are equal in precedence they will be evaluated from left to right, i.e. the == first followed by the >. After these evaluations, the condition is simplified to just the && operator with a Boolean value either side of it (whether they are true or false will depend on the values of the gender and calories variables). Finally, this && operator is evaluated.

In this example, removing the brackets still resulted in the desired behaviour.

However, this will not always be the case. Furthermore, even if the brackets are not essential it can still be a good idea to include them to make your code easier to understand.

## 2.4 `switch` Statements

Sometimes you may have many `if` statements which all use conditions based on the same variable. It is not incorrect to use `if` statements in such cases, but it can lead to a large number of consecutive `if` statements in your code, making it more difficult to understand, and more prone to having errors. A preferable technique is to use a `switch` statement. The `switch` statement offers an easy way of coding situations where the same variable needs to be checked against a number of different values.

**Example 2.3.** The following example illustrates the use of a `switch` statement.

```
switch day
    case 1
        day_name = 'Monday';
    case 2
        day_name = 'Tuesday';
    case 3
        day_name = 'Wednesday';
    case 4
        day_name = 'Thursday';
    case 5
        day_name = 'Friday';
    case 6
        day_name = 'Saturday';
    case 7
        day_name = 'Sunday';
    otherwise
        day_name = 'Unknown';
end
```

MATLAB will compare the *switch expression* (in this case, `day`) with each *case expression* in turn (the numbers 1-7). When a comparison evaluates to true, MATLAB executes the corresponding statements and then exits the `switch` statement, i.e. control flow passes to after the `end` statement. The

`otherwise` block is optional and executes only when no comparison evaluates to true. Note also that the `switch` statement is used only for equality tests - you cannot use it for other types of comparison (e.g. `>`, `<`, etc.).

In the above example the switch expression was compared to a single value in each case. It is possible to compare the expression to multiple values by enclosing them within curly brackets and separating them by commas. The corresponding statements are executed if *either* of the values is matched. This is equivalent to an `if` statement with multiple equality tests combined using a `||` operator.

**Example 2.4.** A `switch` statement with multiple values in each case is illustrated below.

```
switch day
    case {1,2,3,4,5}
        day_name = 'Weekday';
    case {6,7}
        day_name = 'Weekend';
    otherwise
        day_name = 'Unknown';
end
```

## 2.5 `for` Loops

In addition to conditional statements, the second fundamental type of programming construct is the *iterative* statement. There are several types of iterative statement provided in MATLAB. The first is a `for` loop. The control flow of a MATLAB `for` loop is shown in Figure 2.3. With iteration the same statement, or statements, is executed a number of times. In Figure 2.3 the *statement* is continually executed until a certain *condition* becomes false. A *loop variable* will be updated in each iteration to take on a different value from an *array*, which has been initialised at the beginning of the `for` loop. When the statement has been executed once for each element of the array, the condition becomes false.

**Example 2.5.** `for` loops can be useful when you want to execute the same, or a similar, sequence of statements a fixed number of times. Consider the

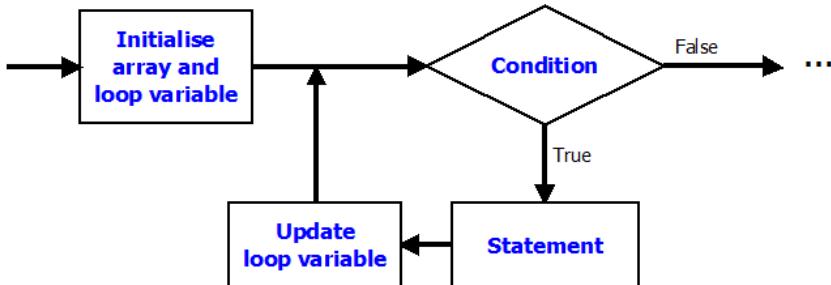


Figure 2.3: The control flow of a `for` loop

following example first, then we will return to the flow chart in Figure 2.3.

```
% ask user for input
n = input('Enter number:');
f = 1;
if (n >= 0)
    for i = 2:n
        f = f * i;
    end
    disp(['The factorial of ' num2str(n) ' is ' num2str(f)]);
else
    disp(['Cannot compute factorial of -ve number']);
end
```

This code reads in a number from the user and computes its factorial. (Note that in reality MATLAB has a built-in function to compute factorials: it's called `factorial`.) The code first checks to see if the number is greater than or equal to zero. If not, program execution jumps to after the `else` statement and an error message is displayed. If it is greater than or equal to zero, the program executes the statement `f = f * i` once for each value in the array `2:n`. For each time around this loop (an *iteration*), the variable `i` takes on the value of the current element of the array. So, for the first iteration `i=2`, for the second iteration `i=3`, and so on, until `i` takes on the value of `n`.

As an aside, it may seem strange at first to see an equals sign with expressions on the left and right hand sides that are clearly not the same, i.e. `f` is not equal to `f * i` unless `i` is 1. But actually it's wrong to think of this as a statement of equality. Rather, it's an *assignment* of a new value (the expression on the right hand side) to a variable (the left hand side). It just

so happens, in this case, that the expression used to compute the new value of  $f$  includes the old value of  $f$ .

Going back to the code, after the loop has finished all of its iterations (i.e.  $i$  has taken on all values of the array  $2:n$  and  $f = f * i$  executed for each one), it exits and program execution continues after the `end` statement. Since  $f$  was initialised to 1, the result of this loop is to multiply 1 by every integer between 2 and  $n$ . In other words, it computes the factorial of  $n$ .

Try typing in this code, saving it as an *m*-file and running it. Read through it carefully to make sure you understand how it works. You can step through it using the debugger to help you.

Now let's return to the flow chart in Figure 2.3. First, the *initialise array and loop variable* box corresponds, in our MATLAB example, to setting up the array  $2:n$  and initialising  $i$  to its first element. The *condition* is to test if we have come to the end of the array. When we have and this condition becomes false the loop will exit. The *statement* corresponds to  $f = f * i$ . The *update loop variable* box will modify  $i$  so that it takes on the value of the next element in the array.

Generally, `for` loops are appropriate when it is known, before entering a loop, exactly how many times the loop should be executed. For instance, in the example above the value of  $n$  was entered by the user so the number of loop iterations could be computed from this value. If it is not possible to know how many times the loop should be executed, an alternative iterative construct is necessary, as will be described below.

## 2.6 `while` Loops

The second type of iterative statement available in MATLAB is the `while` loop. The control flow of a `while` loop is shown in Figure 2.4. First, a condition is tested. If this condition evaluates to false then control passes to immediately after the `while` statement. If it evaluates to true then a statement (or sequence of statements) is executed and the condition is retested. This *iteration* continues until the condition evaluates to false.

Therefore, a `while` loop will execute continuously until some condition is no longer met. This type of behaviour can be useful when continued loop execution depends upon some computation or input that cannot be known

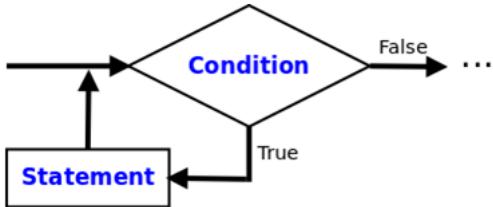


Figure 2.4: The control flow of a `while` loop

before the loop starts.

**Example 2.6.** Consider the following simple example.

```

i=round(rand(1) * 9) % random integer between 0 and 9
guess = -1;
while (guess ~= i)
    guess = input('Guess a number:');
    if (guess == i)
        disp('Correct!');
    else
        disp('Wrong, try again ...');
    end
end

```

This program first generates a random integer between 0 and 9. The MATLAB `rand` statement generates a random number from a uniform distribution between 0 and 1, so by multiplying this number by 9 and rounding to the nearest integer (using the `round` statement) we can get a random integer between 0 and 9. The following `while` loop will iterate whilst the condition `guess ~= i` is true. In other words, the loop will execute so long as `guess` is not equal to the random integer `i`. Since `guess` is initialised to -1 before the loop starts, the loop will always execute at least once. Inside the loop, a number is read from the user, and an `if` statement is used to display an appropriate message depending upon whether the guessed number is correct or not. The program will continue asking for guesses until the correct answer is entered.

Generally, `while` loops are useful when it is not known in advance how many times the loop should be executed. Both `for` loops and `while` loops can be very useful, but in different types of situation. In time you will

learn to recognise when to use each of these types of iterative programming construct.

## 2.7 A Note about Efficiency

**Example 2.7.** There are many programming situations in which an array of values needs to be built up inside a loop. For example, the following piece of code<sup>1</sup> uses a `for` loop to build up an array of 1,000,000 random numbers between 0 and 1.

```
tic
nRands = 1000000;
for i=1:nRands
    rand_array(i) = rand(1);
end
toc
```

The code also uses some built-in MATLAB functions (`tic` and `toc`) to measure how long it takes to build the array. However, the code is not particularly efficient. The reason is that in MATLAB, arrays are *dynamically* allocated. This means that memory is set aside (or *allocated*) for the array based on the highest indexed element that you have created so far. Therefore, whenever you need a larger array than the one you already have, you can just index using a larger number and MATLAB will automatically resize the original array to be the new bigger size. This is a nice feature that makes programming easier at times. However, there is a cost associated, which is that your program may not execute as quickly. This is because sometimes MATLAB will not be able to simply extend the current block of memory that the array is stored in, and will have to allocate a new, larger block somewhere else and copy everything over to there. Therefore, if you know (or have a good guess) at how big the array needs to be in the first place, you can “pre-allocate” it, and your program may be more efficient as a result.

**Example 2.8.** Below is shown the same piece of code with pre-allocation of the array.

---

<sup>1</sup>Adapted from  
[www.cs.utah.edu/~germain/PPS/Topics/MATLAB/preallocating\\_space.html](http://www.cs.utah.edu/~germain/PPS/Topics/MATLAB/preallocating_space.html)

```

tic
nRands = 1000000;
rand_array = zeros(1,nRands);
for i=1:nRands
    rand_array(i) = rand(1);
end
toc

```

Here, the `zeros` function is used to pre-allocate a large array of zeros. Therefore, MATLAB knows straight away how large a block of memory to set aside and will never have to allocate a new block and copy data across. Thus, the code should run quicker. You can verify this yourself by typing in the two versions of the code and running them. The code should measure how long it took to execute and display this information.

We will return to the important topic of code efficiency in Section 9.

## 2.8 `break` and `continue`

Another way of altering the control flow of a program is to use jump statements. The effect of a jump statement is to unconditionally transfer control to another part of the program. MATLAB provides two different jump statements: `break` and `continue`. Both can only be used inside `for` or `while` loops.

The effect of a `break` statement is to transfer control to the statement immediately following the enclosing control structure.

**Example 2.9.** As an example, consider the following code.

```

total = 0;
while (true)
    n = input('Enter number: ');
    if (n < 0)
        disp('Finished!');
        break;
    end
    total = total + n;
end
disp(['Total = ' num2str(total)]);

```

This piece of code reads in a sequence of numbers from the user. When the user types a negative number the loop is terminated by the `break` statement. Otherwise, the current number is added to the `total` variable. When the loop terminates (i.e. a negative number is entered), the value of the `total` variable, which is the sum of all of the numbers entered, is displayed.

The `continue` statement is similar to the `break` statement, but instead of transferring control to the statement following the enclosing control structure, it simply terminates the current iteration of the loop. Program execution resumes with the next iteration of the loop.

**Example 2.10.** For example, examine the following piece of code.

```
total = 0;
for i = 1:10
    n = input('Enter number: ');
    if (n < 0)
        disp('Ignoring!');
        continue;
    end
    total = total + n;
end
disp(['Total = ' num2str(total)]);
```

A sequence of numbers is again read in and summed. However, this time there is a limit of 10 numbers, and negative numbers are ignored. If a negative number is entered, the `continue` statement causes execution to resume with the next iteration of the `for` loop.

## 2.9 Nesting Control Structures

**Example 2.11.** Any of the above control structure statements can be *nested*. This simply means putting one statement inside another one. We have already seen nested statements in the examples given above. As another example, examine the following piece of code and see if you can work out what it does.

```
n = input('Enter number: ');
while (n >= 0)
    f = 1;
```

```

for i = 2:n
    f = f * i;
end
disp(f);
n = input('Enter number: ');
end

```

Here, we have a `for` loop nested inside a `while` loop. The `while` loop reads in a sequence of numbers from the user, terminated by a negative number. For each positive number entered, the `for` loop computes its factorial, which is then displayed.

## 2.10 Summary

Control structures allow you to alter the natural sequential flow of execution of program statements. Two fundamental types of control structure are *conditional* statements and *iterative* statements. MATLAB implements conditional control flow using the `if–else–end` and `switch` statements. Iterative control flow is implemented using `for` loops and `while` loops. All of these control structures can be *nested* inside each other to produce more complex forms of control flow. The `break` and `continue` statements allow ‘jumping’ of control from inside loop statements.

## 2.11 Further Resources

- MATLAB documentation on control flow statements:  
<http://www.mathworks.co.uk/help/matlab/control-flow.html>

## 2.12 Exercises

1. Write a script that asks the user to enter a number, and then uses an `if` statement to display either the text “odd” or “even” depending on whether the number is odd or even.  
*(Hint: see the documentation for the `mod` function)*
2. If  $a=1$ ,  $b=2$  and  $c=3$ , use your knowledge of the rules of operator precedence to predict the values of the following expressions (i.e. work them out in your head and write down the answers). Verify your predictions by evaluating them using MATLAB.

- (a)  $a+b * c$
- (b)  $a \wedge b+c$
- (c)  $2 * a == 2 \&& c - 1 == b$
- (d)  $b + 1:6$
3. Cardiac resynchronisation therapy (CRT) is a treatment for heart failure that involves implantation of a pacemaker to control the beating of the heart. A number of indicators are used to assess if potential patients are suitable for CRT. These can include:
- New York Heart Association (NYHA) class 3 or 4 - this is a number representing the severity of symptoms, ranging from 1 (mild) to 4 (severe);
  - 6 minute walk distance (6MWD) less than 225 metres; and
  - Left ventricular ejection fraction (EF) less than 35%.

Write a MATLAB *m*-file containing a single `if` statement that decides if a patient is suitable for CRT, based on the values of three numerical variables: `nyha`, `sixmwd` and `ef`, which represent the indicators listed above. All three conditions must be met for a patient to be considered suitable. You can test your program by assigning values to these variables from the test data shown in Table 2.3.

Patient	NYHA Class	6MWD	EF	Suitable for CRT?
1	3	250	30	No
2	4	170	25	Yes
3	2	210	40	No
4	3	200	33	Yes

Table 2.3: Data used to select patients for CRT.

4. Not all patients respond positively to CRT treatment. Common indicators of success include:
- Decrease in NYHA class of at least 1;
  - Increase in 6MWD of at least 10% (e.g. from 200m to 220m); and
  - Increase in EF of at least 10% (e.g. from 30% to 40%).

Table 2.4 shows post-CRT data for patients 2 and 4 from Exercise 3 (who were considered suitable for CRT). Write a MATLAB *m*-file that determines if a patient has responded positively to CRT treatment. All three conditions must be met for a patient to be considered a positive responder. The program should use a single `if` statement which tests conditions based on the values of six numerical variables: `nyha_pre`, `sixmwd_pre`, `ef_pre`, `nyha_post`, `sixmwd_post` and `ef_post`, which represent the pre- and post-treatment indicators.

Patient	NYHA Class	6MWD	EF	Responder?
2	3	220	30	No
4	2	230	45	Yes

Table 2.4: Post-CRT patient data.

5. Write a MATLAB *m*-file to read a character from the keyboard, then read a number. Based on the value of the character the program should compute either the *sine* of the number (if the character was a ‘s’), the *cosine* (if it was a ‘c’) or the *tangent* (if it was a ‘t’). If none of these three characters was entered a suitable error message should be displayed. Use a `switch` statement in your program.  
*(Hint: if you use the `input` command to read a character, by default it will try to ‘evaluate’ it as a variable. To prevent this, add a second argument ‘s’. See the MATLAB documentation for `input` for details.)*
6. The normal human heart rate ranges from 60-100 beats per minute (BPM) at rest. *Bradycardia* refers to a slow heart rate, defined as below 60 BPM. *Tachycardia* refers to a fast heart rate, defined as above 100 BPM. Write a MATLAB *m*-file to read in a heart rate from the keyboard, and then display the text “Warning, abnormal heart rate” if it is outside of the normal range.
7. Now modify your solution to Exercise 6 to display one of two warning messages, depending on whether the heart rate indicates bradycardia or tachycardia. If the heart rate is within the normal range the text “Normal heart rate” should be displayed.

8. In Section 1 Exercise 7 you wrote MATLAB code to compute a patient's body mass index (BMI), according to the formula

$$\text{BMI} = \text{mass}/\text{height}^2$$

based on a height (in metres) and weight (in kilograms) read in from the keyboard. Modify this code to subsequently display a text message based on the computed BMI as indicated in Table 2.5.

Range	Message
$\text{BMI} \leq 18.5$	“Underweight”
$18.5 < \text{BMI} \leq 25$	“Normal”
$25 < \text{BMI} \leq 30$	“Overweight”
$30 < \text{BMI}$	“Obese”

Table 2.5: BMI classifications.

9. Write a MATLAB *m*-file that reads in two numbers from the keyboard followed by a character. Depending on the value of the character, the program should output the result of the corresponding arithmetic operation ('+', '−', '\*' or '/'). If none of these characters was entered a suitable error message should be displayed. Use a `switch` statement in your program.
10. Figure 2.5<sup>2</sup> shows how blood pressure can be classified based on the diastolic and systolic pressures. Write a MATLAB *m*-file to display a message indicating the classification based on the values of two variables representing the diastolic and systolic pressures. The two blood pressure values should be read in from the keyboard.
11. Write a `switch` statement that tests the value of a character variable called `grade`. Depending on the value of `grade` a message should be displayed as shown in the table below. If none of the listed values are matched the text “Unknown grade” should be displayed.

---

<sup>2</sup>Adapted from <http://www.bloodpressureuk.org/BloodPressureandyou/Thebasics/Bloodpressurechart>

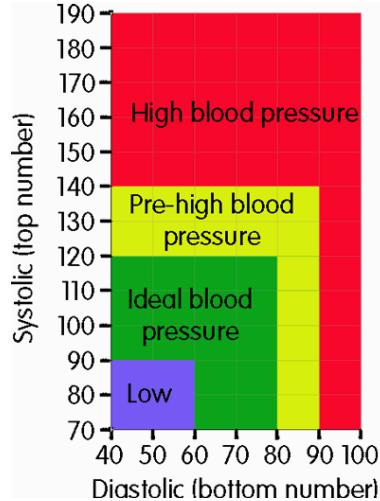


Figure 2.5: Blood pressure classifications.

<u>Grade</u>	<u>Message</u>
'A' or 'a'	"Excellent"
'B' or 'b'	"Good"
'C' or 'c'	"OK"
'D' or 'd'	"Below average"
'F' or 'f'	"Fail"

12. Write MATLAB code to solve the following problems:
- (a) Write a script containing a `for` loop that displays the square roots of the numbers from 1 to 20.
  - (b) Extend the script to read a number from the user and use this as the maximum number.
  - (c) Further extend the script so that it displays the square root values only if they are less than 5.
13. Write a MATLAB *m*-file that uses an appropriate iterative programming construct to compute the value of the expression:
- $$\sum_{a=1}^5 a^3 + a^2 + a$$
14. Write a MATLAB *m*-file to read in an integer *N* from the keyboard and then use an appropriate iterative programming construct to compute

the value of the expression:

$$\sum_{n=1}^N \frac{n+1}{\sqrt{n}} + n^2$$

15. Write a MATLAB *m*-file that continually displays random numbers between 0-1, each time asking the user if they want to continue. If a ‘y’ is entered the program should display another random number, if not it should terminate.
16. Modify your solution to Exercise 15 so that rather than displaying a single random number, it asks the user how many random numbers they would like. If they enter a number greater than zero the program displays that many random numbers. Otherwise it should repeat the request for the number of random numbers. The program should still terminate when any character other than a ‘y’ is entered when the user is asked if they want to continue.
17. The Taylor series for the exponential function  $e^x$  is:

$$1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} \dots$$

Write a MATLAB *m*-file to read a number  $x$  from the keyboard and compute its exponential using the Taylor series. Also compute it using the built-in MATLAB function `exp`. Run your program several times with different numbers of terms in the Taylor series to see the effect this has on the closeness of the approximation.

18. Now modify your solution to Exercise 17 so that, as well as computing and displaying the result of the Taylor series expansion, it also computes and displays the number of terms required for the approximation to be accurate to within 0.01.
19. Write a MATLAB *m*-file to compute the value of the expression

$$\sin(x) + \frac{1}{2}\cos(x)$$

for values of  $x$  between 0 and  $2\pi$  in steps of 0.0001, and produce a plot of the results.

Write two separate MATLAB *m*-files to implement this task: one that produces the result using MATLAB array processing operations as we learnt about in Section 1; and one that builds up the array result element-by-element using an iterative programming construct. Add code to time your two implementations. Which is quicker? Why?

20. Modify your solution to Exercise 5 so that, rather than reading a single character/number, the program continually reads character/number pairs and displays the trigonometry result. The program should terminate when the character ‘x’ is entered. In this case, no number should be read and the program should exit immediately.
21. In Section 1 Exercise 15 we introduced the concept of the injury severity score (ISS). The ISS is a medical score to assess trauma severity and is calculated as follows. Each injury is assigned an abbreviated injury scale (AIS) score between 0-5 which is allocated to one of six body regions (head, face, chest, abdomen, extremities and external). Only the highest AIS score in each body region is used. The three most severely injured body regions (i.e. with the highest AIS values) have their AIS scores squared and added together to produce the ISS score. Therefore, the value of the ISS score is always between 0-75.  
Write a MATLAB *m*-file to compute the ISS given an array of 6 AIS values, which represent the most severe injuries to each of the six body regions. Test your code using the array [3 0 4 5 3 0], for which the ISS should be 50.
22. In Section 1 Exercise 18 you wrote MATLAB code to check if an integer was divisible by 9 by summing its digits: if they add up to 9 it is divisible by 9. For integers up to 90 only a single application of the rule is required. For larger integers, multiple applications may be required. For example, for 99 we add its digits getting the result 18. Then, because 18 still has multiple digits we add them again to get 9, confirming that 99 is divisible by 9. You should continue adding the digits until you are left with a single number.  
Write a MATLAB *m*-file to read in a single number from the keyboard, and use the rule to determine it is divisible by 9. An appropriate message should be displayed depending on the result. You can assume that the number will always be less than 1000.

## Notes

## 3 Functions

*At the end of this section you should be able to:*

- Write and run a MATLAB m-file that contains a sequence of commands
- Write and run a MATLAB m-file that contains a function which takes arguments and returns output
- Explain the meaning of the scope of a variable in a function
- Include checks for errors or warnings in your own functions
- Set the MATLAB path so that functions you write can be called from any location in the file system
- Write recursive functions in MATLAB

### 3.1 Introduction

In this section, we will consider the programming construct known as the ‘function’. We have already seen how MATLAB provides a large number of built-in functions for our use, so you should be aware that **a function is a piece of code that can take some arguments as input and carry out some tasks that can produce a result**. Functions can be particularly useful if you want to perform exactly the same operation many times with different input values.

### 3.2 Functions

MATLAB has its own built in functions such as `sin`, `*`, `exp`, etc. but it also allows us to define our own functions. We can save any function we write in an *m*-file in the same way as we can save any set of commands to form a script and so that we can share our work with others.

So it is important to note that there **two kinds of m-file!** There are *function m*-files and *script m*-files. We will discuss this more later in section [3.4](#).

The following examples illustrate some functions.

**Example 3.1.** Here is a very simple (and quite silly) function. Use the editor window to type this code into a new file. Then save it in a function *m*-file with the name `add_two.m`.

```

function out = add_two(in)
% Usage:
%   out = add_two(in)
%   in : Argument passed in
%   out : Output, result of adding two to input.

out = in + 2;
end

```

This very simple function has a single input argument, `in`, and a single output argument, `out`, which is the result of adding two to the input.

Once the file containing the function is saved, we can call our function from the command window:

```

>> a = 5
a =
5
>> b = add_two(a)
b =
7

```

A function can have any number of input arguments (including none) and any number of outputs (including none).

**Example 3.2.** This function takes two inputs and prints their sum to the command window without returning any output. Type it into a file and save it with the name `report_sum.m`:

```

function report_sum(x, y)
% Usage:
%   report_sum(x, y)
%   x, y : numbers to be added

fprintf('The sum is %u\n', x + y)
end

```

Here, we have used the built-in `fprintf` function to print output to the screen. We will discuss this function in more detail in Section 6.5.

**Example 3.3.** This function takes no inputs and gives no return value, it simply prints one of the words ‘low’ or ‘high’ to the screen. Type it into a file and save it with the name `say_low_or_high.m`:

```
function say_low_or_high()
% Usage:
%   say_low_or_high

a = rand;
if a < 0.5
    fprintf( 'low\n' )
else
    fprintf( 'high\n' )
end

end
```

Here, we have again used the built-in `fprintf` function to print output to the screen. We have also used the built-in function `rand` to generate a random number in the interval  $[0, 1]$  (type `help rand` at the command line).

**Example 3.4.** This example is slightly more complex than the previous ones. Use the editor window to type this code into a new file. Then save it with the file name `compare_to_mean.m`.

```
function [l,m] = compare_to_mean(x)
% usage:
%   [l,m] = compare_to_mean(x)
% Input:
%   x - A list of numbers
% Outputs:
%   l - Number of values in x less than mean
%   m - Number of values in x greater than mean

m = mean(x);
less_array = find(x < m);
l = length(less_array);
more_array = find(x > m);
```

```
m = length(more_array);  
end
```

Now type the following commands in the command window (not the editor window, see Figure 1.1).

```
>> a = [1 2 4 9 16 25 36 49 64 81 100];  
[countLess, countMore] = compare_to_mean(a)
```

Note how this call captures both return values and assigns them to the two variables called `countLess` and `countMore`. These are the values called `l` and `m` in the function itself.

It is also possible to call the function and not assign all output variables. This call

```
x = compare_to_mean(a)
```

will only capture the first of the return values (`l` in the original function), assigning it to `x` at the command line. The second return value is discarded.

However, the following call will result in an error because it asks for more return values than the function can provide:

```
[a, b, c] = compare_to_mean(a)
```

As we have seen, in this example we have created a new function that can be executed from the command window in the same way that built-in MATLAB functions can. The function takes an array as its argument, and returns the counts of the array elements that are less than or greater than the mean value of the array elements. Therefore, we must assign the result of the function to *two* variables, enclosed by square brackets. (Effectively we are returning a *single* result that is an array of two elements.)

Examine the `compare_to_mean` function code. The first line is the function **prototype**, or **definition**. This specifies the name of the function, how many input arguments it expects and how many values it will return:

```
function [l,m] = compare_to_mean(x)
```

Note that we do *not* need to state the *types* of the values (e.g. number, character, array) as we do in many programming languages. This is because MATLAB programming uses *dynamic typing*, i.e. variables can be assigned to without specifying their type and their type can change.

The next seven lines are all comments, as they start with the `%` symbol. As pointed out in Section 1.10, it is good practice to include comments like this to help other people to understand and make use of your functions. Furthermore, these comments are used by MATLAB when our function is used with the built-in `help` command, i.e. if we type `help compare_to_mean` in the command window, the comments at the top of the function are printed to the screen<sup>3</sup>.

After the comments, a series of steps are carried out:

- the mean value of the input array, `x` is computed and assigned to the variable, `m`
- The `find` command is used to find the elements of the array that are less than `m`. Note that the expression `x < m` will return an array containing 1s (true) where the original element was less than `m` and 0s (false) where they weren't. The `find` command will return a new array containing the indices (i.e. the array element numbers) of the non-zero elements of this array.
- Therefore, the `length` of the array returned by the `find` command will be the number of the original array elements that were less than the mean. This is assigned to the variable `l`, which was one of the return variables specified in the function prototype
- The same procedure is followed to assign a value to the other return variable, `m`

Read through this code carefully and make sure you understand how it works.

### 3.3 Checking for Errors

Sometimes we might need to check for errors during the running of a function and return before it completes. One common check is to ensure that the user

---

<sup>3</sup>As noted in Section 1.3, this works for all built-in MATLAB functions, e.g. try typing `help sin` at the command line.

has called the function with correct arguments. For instance, in Example 3.4 the user might give an empty array [] as the input. In this case the function should return before attempting any calculation. This can be achieved using the built in `error` function. Example 3.4 can be adjusted slightly as shown below

```
function [l,m] = compare_to_mean(x)
% [Usage and help section goes here ....]

if isempty(x)
    error('compare_to_mean: %s', 'The input array is empty.');
end

% [... function continues as before]
```

The `error` call has two arguments: the first contains a `%s` *format string* that is substituted with the second argument. So, when the `if` condition is true and the user has given an empty array, the full formatted message ‘`compare_to_mean: The input array is empty.`’ is printed in the command window and the function exits immediately without giving any return value. It is important that the error message contains some text to say which function was being called when the error occurred, especially in complex code when lots of functions are being called.

Other format strings are possible when using the `error` command. For example `%u` will allow a later argument to contain an integer that will be inserted. For more information on the `error` function use `doc` or `help` at the command window. We will return to the topic of format strings in Section 6.4. Alternatively, see [this link](#).

### 3.4 Function *m*-files and script *m*-files

We introduced the concept of a script *m*-file in Section 1.9. To recap, a script *m*-file simply contains a sequence of MATLAB commands. It is different from a function *m*-file because it does not take any input arguments or return any value. It does *not* have a function definition at the top of the file.

The following table lists the main differences between function *m*-files and script *m*-files

Script <i>m</i> -file	Function <i>m</i> -file
A simple list of commands	Commands enclosed by <b>key words</b>
No <b>function</b> key word at start	<b>function</b> key word at start of file
No input arguments	Function definition at start lists input arguments
No output arguments	Function definition at start lists output arguments
No key word when finished	<b>end</b> key word at end of function <sup>4</sup> .

**Example 3.5. A script *m*-file** The following is just a list of commands, no key words or definitions are used. Type them into a script *m*-file and with the name *run\_some\_commands.m*

```
disp('Hello world')

total = 0

for i = 1:1000
    total = total + i;
end

fprintf('Sum of first 1000 integers: %u\n' , total)
```

At the command window you can run your script by simply calling the name of the file (without the .m suffix)

```
>> run_some_commands
Hello world
The sum of the first 1000 integers is 500500
>>
```

**Important:** The script *run\_some\_commands.m* will *always do the same thing*. If we want to sum the first 100 or 1,000,000 numbers, we need to edit the file and change it. This is one reason why it is generally better to write a function that can take an input argument.

We can create a function with input and output arguments with the following, save it in a function *m*-file called *sum\_function.m*

---

<sup>4</sup>Not compulsory but very good practice

```

function total = sum_function(N)

total = 0

for i = 1:N
    total = total + i;
end

end

```

We can call the function in a similar way to the script but now we must pass an input argument, e.g., we can pass an input argument of 1000 and collect the return into a variable value

```

>> value = sum_function(1000)

value =

500500

```

If we want the result for a different input argument, we can just call again from the command line.

### 3.5 A function *m*-file can contain more than one function

If we are working on a *m*-file that contains a main function and this main function gets complicated, we can break up some of its work and carry them out in sub-functions that are also contained later within the same file.

This is an example:

**Example 3.6.** Type the following into a file and save it as *get\_basic\_stats.m*

```

function [xMean, xSD] = get_basic_stats(x)
% Usage :
%   [xMean, xSD] = get_basic_stats(x)
%
% Return the basic stats for the array x
% Input: x, array of numbers

```

```

% Outputs: mean, SD of x.

xMean = get_mean(x);
xSD   = get_SD(x);

end

% Helper functions below:
function m = get_mean(x)
m = mean(x);
end

function s = get_SD(x)
s = std(x);
end

```

This example illustrates the following rules:

- The main function appears first (`get_basic_stats`)
- The name of the main function and the *m*-file must match
- Two further *helper* functions are typed in after the end of the main function (`get_mean` and `get_SD`)
- The helper functions are called from within the main function

Note that these helper functions *can only be called by the main function in this file*. They cannot be called, for example, from the command window or from a function in another *m*-file.

**Good advice:** When typing multiple functions into a single *m*-file, make sure that a function only starts when the preceding one has ended (with the `end` key word!). In other words the structure should be like this

```

function val = f1(input1)

% ... f1 code here

end % for f1

function val = f2(input2)

% ... f2 code here

end % for f2

```

It is even better to put a comment line in between functions to make the code readable, as in the following

```
function val = f1(input1)

% ... f1 code here

end % for f1

%%%%%%%%%%%%%%

function val = f2(input2)

% ... f2 code here

end % for f2
```

(*It is actually possible to nest one function inside another but you are **strongly** advised not to do this, it is only rarely used in special circumstances.*)

### 3.6 A script *m*-file CANNOT include functions

As we saw above, a script *m*-file is just a list of commands. The following example modifies the code in Example 3.5 and shows what NOT to do

```
disp('Hello world')

total = find_sum_illegal(1000);

fprintf('Sum of first 1000 integers: %u\n' , total)

function find_sum_illegal(N)

for i = 1:1000
    total = total + i;
end

end
```

This is illegal because the file has started off as a script and later on introduces a function. MATLAB will not allow this to happen. To make the above work, we would need to separate the script part and function part and put them in separate files, a script *m*-file and a function *m*-file.

```

% In file run_some_commands.m
disp('Hello world')

total = find_sum_legal(1000);

fprintf('Sum of first 1000 integers: %u\n' , total)

% In file find_sum_legal.m
function total = find_sum_legal(N)

for i = 1:1000
    total = total + i;
end

end

```

## 3.7 m-files and the MATLAB Search Path

When using *m*-files to save functions, we may need to save them in different locations. This introduces the subject of which set of paths and directories MATLAB will search for *m*-files. We can inspect the current list of paths by typing path at the command window.

If the *m*-file for a function we have written is in the current folder (see Figure 1.1) then we can call the function from the command window directly. If the function's *m*-file is located in a different location from the current one, we need to add that location to the path list if it is not already included. This can be done by using the addpath command, giving the name of the file's location in single quotes:

```
>> addpath('/path/to/some/directory')
```

Once this is done, any script or function *m*-files that are contained in the directory can be called from the command window, whatever the current folder is.

## 3.8 Naming Rules

When saving a file, you need to be aware that MATLAB has rules on how *m*-files can be named. The name of a *m*-file

- must start with a letter, which can only be followed by
  - letters or
  - numbers or
  - underscores
- must have maximum of 64 characters (excluding the .m extension)
- must not be the same as any MATLAB reserved word (e.g. `mean` or `sin`).

For example, names that start with a number, or contain dashes or spaces cannot be used.

For *m*-files that contain a function, the name of the *m*-file and the name of the function (in the definition line) should match. If they do not match then MATLAB may give a warning (depending on the version). For example, if the following function is saved to a file called `give_random.m`

```
function result = get_random()
% Usage get_random: Return a random number

result = rand;
end
```

then there will be a mismatch between the function definition and the file-name.

### 3.9 Scope of Variables

The *scope* of a variable refers to the parts of a program where the variable is ‘visible’ and can be accessed or set.

We say that the scope of variables in a function is *local*, i.e. they can only be accessed and affected by commands within the function and cannot be affected by commands typed at the command window or within other functions or scripts.

**Example 3.7.** Type the following into a function *m*-file called `my_cube.m`

```
function y = my_cube(x)
% my_cube(x) : returns the cube of x
y = x * x * x;
```

```
end
```

Now type the following at the *command window*:

```
>> x = 4;  
>> y = 7;  
>> my_cube(12)
```

We have first defined two variables in the command window called *x* and *y*. Then, after the call to the *my\_cube* function, which also contains variables called *x* and *y*, we can re-inspect the variables *x* and *y* defined in the command window to see that they are unchanged:

```
>> disp([x y])  
4 7
```

In other words, the variables *x* and *y* we created at the command window are not considered to be the same as the variables *x* and *y* in the *my\_cube* function - because they have different *scope*.

The scope of a variable in a function is also local when we use more than one function as shown in the next example.

**Example 3.8.** Type the following code into a function *m*-file and call the file *log\_of\_cube.m*. This function contains a command which calls the *my\_cube* function from Example 3.7

```
function z = log_of_cube(y)  
% log_of_cube(x) : return the log of the cube of x  
z = log(my_cube(y))  
end
```

Calling the *log\_of\_cube* function will create a variable *y* that is local to the *log\_of\_cube* function and pass this as an argument to the function *my\_cube*. The function *my\_cube* also has a local variable named *y* (as an output variable) but this is distinct from the one in the function *log\_of\_cube* and the value it has in the *calling* function does not affect the value it has in the *called* function.

### 3.9.1 Be Careful about Script M-files and Scope

Note that the rules of scope for functions described above *do not apply to script m-files*, i.e. files which simply contain a sequence of commands.

Confirm this by saving the following command into a script *m*-file called *my\_script.m*

```
a = sqrt(50);
```

Now, return to the command window and set the value of a variable called *a*, then follow this by calling *my\_script*, e.g.

```
a >> a = 4  
a =  
    4  
  
>> my_script  
>> disp(a)  
    7.0711
```

In other words, the variable we set at the command window has subsequently been affected by running the *script* saved in the file *my\_script.m*. This would not have happened if we called a *function*, even if that function had a local variable with the name *a* within it.

## 3.10 Recursion: A function calling itself

We can define some functions *recursively*. This means that we can evaluate the function for some inputs in terms of the same function for other inputs.

**Example 3.9.** A good example of a recursive definition is given by the

*factorial* function:

$$\begin{aligned}0! &= 1 \\1! &= 1 \\2! &= 2 \times 1 = 2 \\3! &= 3 \times 2 \times 1 = 6 \\4! &= 4 \times 3 \times 2 \times 1 = 24 \\\vdots\end{aligned}$$

This shows that evaluating the factorial for a number carries out the same operations as evaluating the factorial for the previous number, for example  $4! = 4 \times 3!$ . This is what we will use to write a recursive definition for the factorial operation:

```
function result = my_factorial(n)
% my_factorial(n) : return the factorial of n

if (n<=0)
    result = 1;
else
    result = n * my_factorial(n-1);
end

end
```

In this code, we can see that *the function calls itself* and this defines the recursive nature of the function. Calling the function at the command line, e.g.

```
>> my_factorial(3)
```

will lead the second part of the `if/else` clause to be run:

```
...
result = n * my_factorial(n-1);
...
```

with a value of  $n = 3$ . In other words, a call is made *to the same function* passing an argument of  $n-1 = 2$ . The return value from that call is multiplied by 3 to give the final result.

Each call for a value of  $n > 0$  will involve a call to the same function with a value  $n - 1$ . Thus, the value used when calling the function will decrease by one each time until, eventually,  $n = 0$ . In this case, and only in this case, the function will actually run the first part of the `if/else` clause

```
...
    result = 1;
...

```

and a value of 1 will be returned. Then the return value will be passed to the calling function, which will multiply the result by *its* local value of  $n$  and return the result to its calling function, which will multiply the result by *its* local value of  $n$  and ... so on.

Try out the code for yourself to confirm it works and read through it carefully to make sure you understand it.

**Example 3.10.** Another good example of working recursively is given by the evaluation of a polynomial. Recall that a polynomial is defined by its coefficients. In this example, we will work with a cubic polynomial with the coefficients 2, 3, 7, 1, i.e. the polynomial  $2x^3 + 3x^2 + 7x + 1$ . If we evaluate this expression directly it requires 3, 2 and 1 multiplications for the first three terms and three additions to give a total of nine operations. We can, however, write the evaluation in a recursive way that requires fewer operations: If we take out successively more factors of  $x$  from the terms that contain them we get the following

$$\begin{aligned} 2x^3 + 3x^2 + 7x + 1 &= x(2x^2 + 3x + 7) + 1 \\ &= x(x(2x + 3) + 7) + 1 \\ &= x(x(x(2) + 3) + 7) + 1 \end{aligned}$$

On the first line, we see that the original cubic polynomial is written as  $x$  multiplied by a quadratic plus a constant. In the second line, the quadratic is written as  $x$  multiplied by a linear term plus a constant. In the last line, the linear term is written as  $x$  multiplied by a constant plus another constant.

The last line shows how the polynomial may be evaluated with three additions and three multiplications giving a total of six operations, three fewer than the direct approach. This illustrates one of the benefits of working recursively.

The above suggests how a function to evaluate a polynomial may be written recursively. In a MATLAB style pseudocode, the sequence for the above polynomial could be written as a sequence of calls to a function  $f$  that starts off by taking an array containing the coefficients.

$$2x^3 + 3x^2 + 7x + 1 = f([2, 3, 7, 1])$$

$$\begin{aligned}f([2, 3, 7, 1]) &= x f([2, 3, 7]) + 1 \\f([2, 3, 7]) &= x f([2, 3]) + 7 \\f([2, 3]) &= x f([2]) + 3 \\f([2]) &= 2\end{aligned}$$

The writing of the MATLAB function that will achieve this is left as an exercise.

### 3.11 Summary

Functions can be defined in MATLAB, and can be very powerful if used effectively. They allow programmers to expand the basic capabilities of MATLAB for their own specific purposes. Be careful to note the difference between scripts and functions.

Functions should be saved in a file with the same name as the function. If the file is saved in a different location in the file system then that location must be added to the *path* that MATLAB uses to search for *m*-files. The file containing the function can also contain *sub functions* that are called by the main function in the file.

Functions take input values (arguments) and return values. Variables defined within functions have a *scope* that is *local* to the function.

MATLAB allows definition of *recursive* functions. Recursive functions are functions that make calls to themselves. Each recursive call will have its own local scope.

The `error` command can be used to report errors within functions.

## 3.12 Further Resources

Look at the *File Exchange* section of the *Mathworks* website (The company that develops MATLAB). You will find a large number of useful MATLAB functions that have been written and shared by other people. This can be an extremely valuable time-saving resource. The URL is:  
<http://www.mathworks.com/matlabcentral/fileexchange/>.

## 3.13 Exercises

1. Write MATLAB code to solve the following problems:
  - (a) Write a function called `mycube` to compute and return the cube of a number provided as an argument.
  - (b) Call this function several times from the command window using different input argument values.
  - (c) Write a script to call the function twice with different input arguments and assign the results to two different variables.
  - (d) Extend the script to read a number from the user and supply this as an argument to the function. The script should display the result of the calculation to the command window.
2. A function is defined as

```
function c = blah(a)
b = a + 3;
c = a * 2;
d = b + c

end
```

- (a) Predict the values of `blah(10)` and of `blah(blah(3))` without using MATLAB
- (b) Re-write the function `blah` so that it does exactly the same thing but does not contain any redundant code.
3. Identify problems and errors in the following functions
  - (a) `func1`:

```

function f = func1(x, y)
x = 3 * y;
end

```

(b) func2:

```

function f = func2(a, b)
f = sqrt(3 * a);
end

```

(c) foo:

```

function [a, b] = foo(x, y)
b = a + x + y;
end

```

(d) findSquareRoot:

```

function findSquareRoot(x)
% Usage : findSquareRoot(x)
% Provide the square root of the input
returnValue = sqrt(x)
end

```

4. A function takes three numeric arguments representing the coefficients of a quadratic equation and returns two values which are the two roots of the equation. The name of the function is quadRoots
  - (a) Write a suitable prototype or definition for the function. Add an appropriate help and usage section at the top of the function and save it in a suitably named file.
  - (b) Add a test to check if the quadratic roots are complex and to return with an error if this is true (see Section 3.3, doc error or help error).
  - (c) Complete the function quadRoots and test that it gives the correct results.
  - (d) Modify the quadRoots function so that it accepts a single array containing all three coefficients and returns a single array containing the roots.
5. Write a function that accepts an array of numbers and returns two arrays, the first containing all input numbers that are less than 10

and a second containing all those greater than or equal to 10. Call the function `split_at_ten`. Ensure your code has usage and help comments after the definition. Test it on different arrays to make sure it works.

(*Hint: you can generate random arrays of integers with the built-in `randi` function. Your function will need to return one or more empty arrays if necessary.*)

6. Write a function to find the number of digits for a given positive integer. The prototype of your function should be

```
function noOfDigits = digit_count(n)
```

(*Hint: a `while` loop might be useful.*)

7. A triangle is defined by the lengths of its sides. For a triangle to be valid, the sum of the two shorter sides must be greater than the longest side. The code below is a function that accepts a single 3-element array (e.g. [3 4 5]). It returns a value of 0 or 1 to indicate if the three elements of the array can represent the sides of a valid triangle.

```
function result = is_triangle(sides)
% is_triangle(sideLengths):
%     Check if array of 'sideLengths'
%     represents a valid triangle
%
% Input:
%     sideLengths : array of 3 numbers
% Output:
%     result : 1 if sides are valid
%             0 if not valid

longestSide = max(sides);

result = 0;
if longestSide < sum(sides) - longestSide;
    result = 1;
end
```

- (a) Write a line by line explanation of how the function carries out its task.
- (b) Type the code into an *m*-file and save it with the correct name.

Call the function from the command window to check it gives the correct result for triples of numbers where you know what the output should be.

- (c) Modify the function so that it does not use the `max` function but uses the `sort` function instead. Type `help sort` or `doc sort` at the command line to see what it does.
  - (d) Add some code to the beginning of the function to make sure that the user has provided an array with the correct number of elements. If this is not the case, the function should return with an error message. Type `help error` or `doc error` at the command line to learn about this built-in function.
8. Write a **recursive** MATLAB function to evaluate a polynomial with a given set of coefficients for a given  $x$  value. Use the description given in Example 3.10 to help you. The definition of your function should be

```
function y = my_poly_value(coeffs, x)
```

Make sure the code is well commented and that there is a help section at the top. Note that this function needs to carry the  $x$  value as an argument which makes it different from the more mathematical treatment in the notes.

Use your function to find the value of  $y$  when  $x = -2$  for the polynomial  $x^4 - 2x^3 - 2x^2 - x + 3$ . Check that your function agrees with the built-in function `polyval`. Type `doc polyval` for details of how to use this function.

9. The Fibonacci sequence is 1, 1, 2, 3, 5, 8, .... It is defined by the starting values  $f_1 = 1$ ,  $f_2 = 1$  and after that by the relation  $f_n = f_{n-1} + f_{n-2}$ .

Write a recursive function to find the  $n^{\text{th}}$  term of the Fibonacci sequence. Use the following definition and ensure the code has a help section and good comments.

```
function f = fibonacci(n)
```

## Notes

# 4 Program Development and Testing

*At the end of this section you should be able to:*

- *Incrementally develop a MATLAB function or program*
- *Test each stage of the incrementally developing function or program*
- *Identify common types of error that can occur in MATLAB functions and scripts*
- *Debug and fix errors in scripts and functions*
- *Use MATLAB's built-in error handling functions to help in program debugging*

*A computer program will always do what you tell it to do, but rarely what you want to do.*

**Murphy's Laws of Computing**

## 4.1 Introduction

Most tasks in realistic computer programming settings involve writing substantial amounts of code. This can be in the form of different sets of functions, modules or libraries that can call each other in complex ways. This means that, when tackling a large computer programming task, which can involve more than one person collaborating, there is a need to work methodically and break a large project down into manageable and testable portions.

We already saw some of the debugging and code analysis tools that MATLAB has to offer (Sections [1.11](#) and [1.11.2](#)) and other languages such as C++, Python or Java also have similar debugging and analysis tools. In this section we will revisit these and discuss a general approach for developing and implementing a significant programming project.

## 4.2 Incremental Development

Working on the principle of breaking a large task down into smaller, more manageable and (importantly) testable sub-tasks, the process of *incremental development* can be useful when trying to solve a difficult task or model a complex system using a computer program.

When programming something substantial, it is better to avoid trying to

take a “big bang” approach to coding. You will generally fail if you try to solve it all at once.

General tips for incremental development are:

- Write your code in small pieces.
- The pieces can be incomplete but should work.
- Don’t try and write everything in one massive function in one go.
- For example, if the program asks for user interaction
  - Start off with just enough code to display output.
  - Run that part and check it.
  - Then write a bit more code to ask the user for input.
  - Check the new part by printing what the user gives as input to the screen.
  - Add a bit more code to actually do something (simple) with the user input.
  - Check the ‘doing something simple part’ - perhaps by again printing out its result.
  - Add a little more code to do something a bit more than the simple stuff already done.
  - etc.
  - Continue these steps of adding a bit more code and checking it works until you have a final version of your working program.

This kind of iterative way of working with frequent checking is important. Adding a large amount of code before checking can be difficult. First, the program itself may not run. Second, it may give incorrect output. In either case, if a large amount of code was introduced since the last check it will be more difficult to find where the bug or error is.

*The bottom line is that you should aim to always have a working version of your program.*

This section tries to illustrate this by example and, necessarily, the actual tasks presented are relatively simple and not genuinely complex but should serve to illustrate the general principle.<sup>5</sup>

#### **Example 4.1.** Incremental programming for interest rate calculation

---

<sup>5</sup>The first example is adapted from one by Sue Evans at UMBC.

**Task:** Write an interactive function to calculate the compound interest given by a savings account in which the interest is added annually. The user should be prompted to give the initial amount, the interest rate and the number of years for which interest should be accumulated.

Broadly, the function needs to do the following:

- Print a description of what the function does
- Get an initial amount from the user
- Get the interest rate from the user
- Get the number of years from the user
- Calculate the interest
- Display a report back to the user showing the interest and final amount

The calculation of the interest can be done in a loop, perhaps in another function, and in terms of *pseudocode* can look something like this

```
amount = initial amount
years counted = 0
while years counted < number of years
    amount = amount + amount x interest rate
    years counted = years counted + 1
end
```

The program should provide output that reflects what the user entered and report the final amount of money in the account and the total interest accrued. For example, the following represents the format of the required output:

```
Initial Amount : GBP 2.00
Interest rate : 3.00 %
Number of years : 5

Amount at end : GBP 2.32

Accrued interest : GBP 0.32
```

## Version 1

```
function calculateInterest
% Usage: calculateInterest

%v1

helpString = [...]
```

```

'This function calculates the interest accrued in\n' ...
'an account. Interest is compound and added each\n' ...
'year. You will need to give the initial amount\n' ...
'of money, the interest rate and the number of\n' ...
'years that the amount is in the account. \n\n'];

```

```

fprintf(helpString)

```

Notice that this function does very little. It is a long way from the one we want but it is *a working function* that we can test. In this case, testing just involves running it to see if the help message prints out correctly. When we run it at the command line, we get the following output:

### Output 1

```

>> calculateInterest
This function calculates the interest accrued in
an account. Interest is compound and added each
year. You will need to give the initial amount
of money, the interest rate and the number of
years that the amount is in the account.

```

### Version 2

```

function calculateInterest
% Usage: calculateInterest

% v2

helpString = [...
'This function calculates the interest accrued in\n' ...
'an account. Interest is compound and added each\n' ...
'year. You will need to give the initial amount\n' ...
'of money, the interest rate and the number of\n' ...
'years that the amount is in the account. \n\n'];

fprintf(helpString)

initialAmount = input('Please give the initial amount of ...
money: ');
interestRate = input('Please give the interest rate: ');
noOfYears = input('Please give the number of years: ');

fprintf('\nInitial Amount : GBP %0.2f\n', initialAmount);
fprintf('Interest rate : %0.2f %%\n', interestRate);

```

```
fprintf('Number of years : %u\n\n', noOfYears);
```

This version only represents an incremental difference from the previous one. It still works as a function even if it is not yet the complete finished version we seek. It simply asks for input and writes it right back to the user, without actually doing any of the calculation. The writing back actually carries out part of the requirement, the reporting part.

We can test our function by inputting some values and testing the small increment we made.

## Output 2

```
>> calculateInterest
This function calculates the interest accrued in
an account. Interest is compound and added each
year. You will need to give the initial amount
of money, the interest rate and the number of
years that the amount is in the account.

Please give the initial amount of money: 2
Please give the interest rate: 3
Please give the number of years: 4

Initial Amount    : GBP 2.00
Interest rate    : 3.00 %
Number of years  : 4
```

## Version 3

```
function calculateInterest
% Usage: calculateInterest

% v3
helpMessage = [' .... help message as before ... \n\n'];
fprintf(helpMessage)

% Store the original amount and track
% the varying amount with a new variable.
amount = initialAmount;

% Accumulate interest
amount = amount + (interestRate / 100) * amount;

% Calculate total interest
```

```

interest = amount - initialAmount;

% Produce report.
fprintf('\nInitial Amount : GBP %0.2f\n', initialAmount);
fprintf('Interest rate : %0.2f %%\n', interestRate);
fprintf('Number of years : %u\n\n', noOfYears);

fprintf('Amount at end : GBP %0.2f\n', amount);
fprintf('Accrued interest : GBP %0.2f\n\n', interest);

```

Between this version and the previous one, a part of the interest rate calculation is carried out. Specifically, interest for one year only is calculated and included in the report. The function still does not do what we want as it ignores the number of years specified by the user. Again, we test the function to see if it gives the expected output:

### Output 3

```

>> calculateInterest
... help message stuff as before ...

Please give the initial amount of money: 2
Please give the interest rate: 3
Please give the number of years: 4

Initial Amount : GBP 2.00
Interest rate : 3.00 %
Number of years : 4

Amount at end : GBP 2.06
Accrued interest : GBP 0.06

```

### Version 4

The increment for the next version is simply to put the line that calculates the interest in a very trivial loop. The loop just goes from 1 to 1 so it only executes its contents once. This program should give the same output as the previous increment:

```

function calculateInterest

% usage and help message stuff as before

initialAmount = input('Please give the initial amount of ...

```

```

    money: ');
interestRate = input('Please give the interest rate: ');
noOfYears     = input('Please give the number of years: ');

% Store the original amount and track
% the varying amount with a new variable.
amount = initialAmount;

% Accumulate interest
for i = 1:1
    amount = amount + (interestRate / 100) * amount;
end

% Calculate total interest
interest = amount - initialAmount;

% Produce report
% ... reporting as before ...

end

```

## Output 4

The output from this version should be identical to the output from the previous version so we do not show it here. The way we have incremented the structure of the program means that if we see any difference in the output, then there must be an error somewhere.

Having said that, we can still test the program with different figures. For example, the following shows a single year's worth of interest being added and the total number of years is still effectively ignored:

```

>> calculateInterest
... help message stuff as before ...

Please give the initial amount of money: 100
Please give the interest rate: 7
Please give the number of years: 6

Initial Amount : GBP 100.00
Interest rate : 7.00 %
Number of years : 6

Amount at end : GBP 107.00
Accrued interest : GBP 7.00

```

## Version 5

Now we can make a final increment to our program to take into account the number of years for the interest. Again, only a very small change is made as shown below, where the main loop uses this variable input from the user and all of the interest is calculated instead of just that for a single year.

```
function calculateInterest

    % ... Usage and help message stuff as before
    % ... Input from user as before

    amount = initialAmount;

    % Accumulate interest
    for i = 1:noOfYears
        amount = amount + (interestRate / 100) * amount;
    end

    % Calculate total interest
    interest = amount - initialAmount;

    % ... reporting as before ...
end
```

Now we are in a position to test what is hopefully a fully working function.

## Output 5

```
>> calculateInterest
... help message stuff as before ...

Please give the initial amount of money: 100
Please give the interest rate: 7
Please give the number of years: 6

Initial Amount : GBP 100.00
Interest rate : 7.00 %
Number of years : 6

Amount at end : GBP 150.07
Accrued interest : GBP 50.07
```

### 4.3 Are we Finished? Validating User Input.

In one sense, the answer is yes because we have a working function that achieves its initial aim. In another sense we could still have work to do because the function might need to be made more *robust*. This could involve checking that the user is giving sensible or valid values for the inputs prior to calculation of the interest.

**Validating user input** is an important part of programming because a user giving input to a function or program that does not match what the programmer expects is one of the main reasons for failure.

In the case of Example 4.1, we could still need to check if the user has given a sensible starting investment (perhaps it cannot be negative). The number of years might be constrained to be a positive integer and the interest rate should not be a complex number or a character array. The interest rate in particular was assumed to be a percentage but the user may give it as fractional value between zero and one and the program could be made more careful in how the user understands this point.

For example, if we want to implement a rule that the starting investment amount should not be negative, then we can introduce the following code at the top of the function:

```
function calculateInterest
    % ... Usage and help message stuff as before
    % ... Input from user as before ...

    if (amount < 0)
        error('calculateInterest: Amount is negative!');
    end

    % ... rest of function as before.
```

If the user gives a negative amount the function now prints out the error message and returns without continuing the rest of the code. Notice how the error message begins by giving the name of the function where the error is raised.

If we want to give a warning message to the user but still want to carry on executing the function (rather than returning), then we can use the built-in function `warning`. Type `help warning` and `help error` at the command

window for more information.

## 4.4 Debugging a Function

The basics of the MATLAB debugger were described in Section 1.11. Here, we consider some of the types of errors that the debugger is useful to help identify and fix.

When writing a program or function that is non-trivial, the chances of it working perfectly first time are quite low, even for a skilled programmer. Debugging tools are there to help identify why a program does not work correctly or as expected.

When debugging, we need to work systematically and run our program with various sets of test data to find errors and correct them when we find them. If we work incrementally as shown in Example 4.1, then the repeated nature of the testing allows errors to be caught earlier, which reduces the chance of later errors or minimises their effect.

There are three main types of error that a function or program can contain:

- Syntax errors
- Run-time errors
- Logic errors

The term ‘syntax’ relates to the *grammar* of the programming language. Each language has a specific set of rules for how commands can be written in that language and these rules are collectively known as the language’s syntax.

When a line in a MATLAB function contains a syntax error, the built-in *Code Analyser* (See Section 1.11.2) should highlight in red the corresponding line of code. Hovering the mouse over the line should result in a message being displayed to the user that describes what MATLAB thinks is the particular syntax error in the line. In other words, MATLAB can detect an error with the script or function before it is run.

### Example 4.2. A syntax error

If the following code is entered into a script, say *my\_script.m*, using the editor, then the line containing the `if` statement should be highlighted in red to indicate a problem with the syntax.

```
clear, close all  
  
randVal = rand;  
  
if randVal > 0.5  
    disp(theta)
```

This is because the syntax rules of MATLAB require a closing `end` statement for each `if` statement.

If we fail to notice this error and attempt to run the script from the command window, we will get an error when MATLAB reaches the relevant part of the code and a more dramatic message will be displayed to the command window:

```
>> my_script  
  
Error: File: my_script.m Line: 5 Column: 1  
At least one END is missing: the statement may begin here.
```

In this case, clicking on the part of the error report with the line number will take us directly to the corresponding place in the script file so that we can figure out how to correct the error. In this case, adding an `end` statement after the `disp` call should be enough.

### Example 4.3. Another syntax error

The following code contains a syntax error. The MATLAB code analyser should highlight it in red. The code begins by generating a single random number between 1 and 10 and then it seeks to decide if the random number equals three or not, adjusting its output in each case.

```
clear, close all  
  
a = randi(10, 1);  
  
if a = 3  
    disp('Found a three!')  
else  
    disp('Not a three')
```

```
end
```

Type the code into a script called *someScript.m* and note that the `if` statement gets highlighted as a syntax error. The reason for this syntax error is because the = sign used in the `if` statement is an *assignment operator* and should not be used to test for *equality*. The correct symbol to use here is the *comparison operator* which is == in MATLAB (and in quite a few other languages). It is very common to get these two mixed up.

#### Example 4.4. A run-time error

Some errors cannot be determined by MATLAB while we are editing the code and can only be found when the code is run. This is because a specific sequence of commands needs to be run to bring about the conditions for the error. Consider the following code in a script called *someScript.m*. It aims to generate three sequences of numbers and assign them to array variables *a*, *b* and *c*. Then it tries to find the sum of each of these sequences (line numbers have been shown here to help the description).

```
1 clear, close all
2
3 a = 1:10;
4 b = 2:2:20;
5 c = 3:3:30;
6
7 sumA = sum(a);
8 disp(sumA)
9
10 sum = sum(b);
11 disp(sum)
12
13 sumC = sum(c);
14 disp(sumC);
```

In this case, running the script at the command line leads to a run-time error as follows:

```
>> someScript
      55
      110
```

```
Index exceeds matrix dimensions.  
  
Error in someScript.m (line 13)  
sumC = sum(c);
```

The problem here is related to the choice of variable names. The sums of the arrays `a` and `c` have been given sensible names (`sumA` and `sumC`). The sum of the array variable `b` has been stored in a variable called `sum`. This is a problem because the local script has used the name of a *built-in MATLAB function* as a name for a local variable.

This is called ‘shadowing’, i.e. using the name of a built-in function for a variable that we use locally. We can check if a name is already used by MATLAB or in the workspace using the `exist` command. In this case, we can trivially check

```
>> exist('sum')  
  
ans =  
      5
```

The return value of 5 indicates that the name `sum` exists and is attached to a function (as opposed, say, to a file). Type `help exist` at the command window for more details. We can check a more sensible name as follows:

```
>> exist('sumB')  
  
ans =  
      0
```

The value of 0 returned indicates that the name `sumB` is not being used and we can safely use it in our code.

Returning to the code, when the execution reaches line 13, the programmer’s intention was to use the built-in MATLAB function called `sum`, but this now refers to a local variable that contains a single number. The use of the brackets in line 13 is now interpreted as a request for an element in an array but because the variable called `sum` is just a single number, there is no element in the array that it is possible to return. Hence, the code crashes at run-time.

Apart from giving an example of a run-time error this shows why **it is very important not to give names to variables that match names that already exist and refer to built-in functions in MATLAB**. MATLAB will not complain when you do this so be careful!

### Example 4.5. A run-time error

Here is some more code that gives a run-time error:

```
clear, close all  
  
a = 60 * pi / 180;  
  
b = sin(a + x)
```

In this case, when the code is run, the following output is given

```
>> someScript  
Undefined function or variable 'x'.  
  
Error in someScript.m (line 5)  
b = sin(a + x);
```

The script here calls `clear` at the start. This means that even if there were a global variable called `x` available in the function calling the script or in the main workspace, then it will be removed and would no longer be available. Hence, the request to calculate `sin(a + x)` will fail because there is no such variable called `x`. This would need to be defined before the point where the `sin` function is called to make the code work (but after the `clear`).

### Example 4.6. A logic error

Logic errors can be some of the hardest to find, and identifying them often requires careful line-by-line inspection of the code as it is running. This is because no error is reported by MATLAB and the Code Analyser does not highlight anything either. However, when the code is run, an incorrect output or behaviour is the result and MATLAB will not provide any clue as to why this happened. For example, the following code attempts to show that  $\sin \theta = \frac{1}{2}$  when  $\theta = 30^\circ$

```

clear, close all

theta = 30;
value1 = sin(theta);
value2 = 1/2;

fprintf('These should be equal: ')
fprintf(' %0.2f and %0.2f\n', value1, value2)

```

When we actually run the code, we get the following output with no error reported.

```

>> someScript
These should be equal: -0.99 and 0.50

```

This shows two numbers that are clearly not equal so what has gone wrong?

A little investigation can be carried out by going to the command window and looking at the help on the `sin` function which we clearly suspect of being broken:

```

>> help sin
sin    Sine of argument in radians.
       sin(X) is the sine of the elements of X.

```

This shows that the built-in MATLAB trigonometric function `sin` expects its argument in radians so that is what we should give it. The function when used above gave the *correct result* for the sine of 30 radians - not degrees. That is, there was a mismatch between our expectation of the function and what it was designed to do. This error is readily fixed in a number of ways, for example by replacing `theta = 30;` with `theta = 30 * pi / 180;` where we have made use of the built-in MATLAB constant `pi`.

Because there is no error message, logic errors can be difficult to detect, and sometimes we might not even notice that the program or function is not behaving correctly. Even when we detect that there is a logic error by seeing that something is incorrect about the behaviour, it can be difficult to identify where the error actually is (again, because there is no error message pointing to a specific line as in the case of syntax errors). Also, the error will not necessarily be restricted to just one line, it can arise from the way different sections of code interact.

The error in the previous example was due to the intention to use degrees where radians should have been used instead. This is only noticed by observing the actual values of the output and seeing that the numeric value is not correct; in other words, by careful inspection of the output.

One way to identify logic errors in a function is to use the MATLAB interactive debugger to **step through the function line-by-line**, perhaps with a calculator or pencil and paper to hand (see Section 1.11).

We can inspect the value of any variable in the code as we step through by typing its name at the command window when the program is stopped at a specific line. Another way is to hover the mouse over the name of the variable inside the editor whilst debugging - its value will then be highlighted (see screenshot to the right).

```

1 - clear, close all
2
3 - theta = 30;
4 - value1 = sin(theta);
5 - value2 = 1/2;
6
7 - fpr
8
9
10 - if a == 3
11 - disp('x')
12
13

```

Another way to hunt for logic errors is to take each line of the code and paste it into the command window, executing each one at a time and assessing its result to see if it is right.

Putting lots of statements in the code that display output and the values of the most relevant variables using the `disp` or `fprintf` commands is also useful. Values of variables can be displayed before and after critical sections of the code, e.g. calls to other functions or important control sections such as `if` clauses and `for` or `while` loops.

Once we have checked and are satisfied that the function or program is working correctly, the extra output statements should be removed.

## 4.5 Common Reasons for Errors when Running a Script or a Function

There are a few common reasons for errors when running a script or a function. These include:

- We use matrix multiplication when we meant to use *element-wise* multiplication. For example, the following code

```

x = [1 2 3];
y = [6 7 8];
z = x .* y;

```

multiplies  $x$  and  $y$  *element-wise* to give a result of [6, 14, 24]. It is a common mistake to write  $z = x * y$  instead of the above which would lead to an error. In this case it would lead to a run-time error because it is not possible to matrix-multiply the values of  $x$  and  $y$  as defined above (because the sizes don't match). We will cover matrix multiplication in Section 5.7.

- The location of the code that we want to run is not in the MATLAB path, or some other code that it relies on is not in the path (see Section 3.7).
- We forget that MATLAB is *case-sensitive*, e.g. the variables  $x$  and  $X$  are different, as are  $myVar$ ,  $myvar$  and  $MyVar$ .
- We accidentally set off an infinite loop, for example

```

count = 0; sum = 0;
while(count < 100)
    sum = sum + count;
end

```

will start but never end because the variable  $count$  never gets updated within the loop. The following is one way to fix this:

```

count = 0; sum = 0;
while(count < 100)
    sum = sum + count;
    count = count + 1;
end

```

- We have exceeded the memory available for our computer. For example, the command  $\text{rand}(N)$  returns a matrix of random numbers in an array of size  $N \times N$ . So if we run the following code

```

N = 100000;
R = rand(N);

```

then the computer could get very tired as it runs out of memory trying to create a  $10^6 \times 10^6$  array. This corresponds to around 3 or 4 terabytes if four bytes are needed for each array element. This can be viewed as a run-time error.

## 4.6 Error Handling

In the preceding sections, we have discussed some of the kinds of error that may occur when writing code. Some of these can be hard to predict in advance but it is common to write code in way that anticipates the kinds of errors that might occur and tries to deal with them in advance so that the consequences of the errors are not too disastrous. This is known as *error handling* and we give some illustrations in this section of how this can be done.

In general if there is a run-time error that is encountered when the program is run, then MATLAB itself will stop running the program and return. It will give an error message (see Example 4.5), but sometimes the error message that MATLAB gives can be uninformative and we might want to make the error reporting more useful and specific.

### 4.6.1 The `error` and `warning` Functions

We first introduced the `error` function in Section 3.3, and in Section 4.3 we saw a way of using it whilst validating user input. This function, and the closely related `warning` function can be useful when testing for correct behaviour when code is running. Once an error is detected, we can use them to give tailored messages to the user.

**Example 4.7.** For example, we may have some function that is expected to carry out some processing and record the activity to a given file that is required to exist beforehand.

```
function doSomeProcessing(logFileName)

if (~exist(logFileName, 'file'))
    msg = sprintf('No file: %s. Bailing out!\n', logFileName);
    error('MATLAB:processFile:fileMissing', msg)
end

% further code below
```

This code first tests to see if the log file with the given name exists. If it does not, a specific error message is generated using the `sprintf` function to say that it is not there. The message is contained in a character array

variable called `msg` which is passed as the second argument to the `error` function. This is similar to the use of `error` we saw in Section 3.3. However, here we also have an extra argument. The first argument is a unique piece of text which identifies the location and nature of the error call: although not required, it is a good idea for each call to `error` to have a unique identifier. In the code above, if a file with the given name does not exist, then execution will transfer to the `error` function and MATLAB will return and halt processing the script and any that have called it. Any further code below this point will not be run.

**Example 4.8.** In the above example, we might however be a little less strict and decide that if the log file is not present we should still carry out the processing but warn the user that there will be no logging carried out. This is where the `warning` function is useful: the above code can be modified slightly to do this:

```
function doSomeProcessing(logFileName)

if (~exist(logFileName, 'file'))
    msg = sprintf('No such file: %s. There will be no ...
                  logging.\n', logFileName);
    warning('MATLAB:processFile:fileMissing', msg)
end

% further code below
```

In this case, if the log file does not exist, a warning message is displayed but the code below the warning is still executed (although in this example without logging).

In other words, both the `error` and `warning` functions can be used to report a problem to the user but the `error` function then returns and stops running the script or function file.

#### 4.6.2 The `try` and `catch` Method

The `error` and `warning` functions described above can be used when we anticipate a particular type of problem that can lead to an error. Sometimes we may not know in advance what type of error may occur but we might want to prepare for any possible error just in case.

One phrase to describe a program encountering an error is to say that the program has *raised an exception*. The exception itself can be raised for a number of reasons which include

- Trying to access an element of an array that is beyond the last element it contains.
- Trying to open a file that does not exist.
- Performing a type error such as trying to perform an arithmetic operation on a variable that is not numeric.
- etc.

If we have a section of code that could raise a number of different exceptions, we can wrap this code inside a *try-catch clause*. This is a little like an *if-then* clause where a section of code is ‘tried’ and, if it raises an exception, this exception is then ‘caught’ by the second part of the clause.

**Example 4.9.** As a simple and artificial example of the use of a try-catch clause, the following is adapted from the MATLAB documentation

```
A = rand(3);
B = ones(5);

try
    C = [A; B];
catch

    % handle the error
    msg = ['Dimension mismatch: First argument has ', ...
            num2str(size(A,2)), ' columns, second has ', ...
            num2str(size(B,2)), ' columns.'];
    error('MATLAB:tryCatch_ex:dimensions', msg);

end % end try/catch
```

The code within the `try` section tries to concatenate two arrays vertically and store the result in a third array, placing A above B. If there is an error when doing this, then the execution of the program will immediately switch to the code inside the `catch` section.

It is important to note that *any* error in the `try` section will lead to the code in the `catch` clause being run. This allows different possible reasons for the

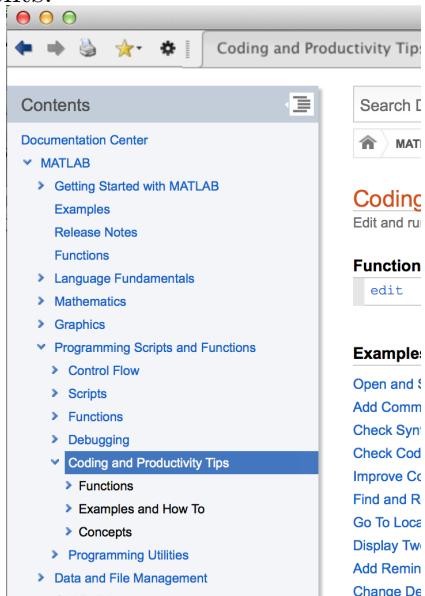
error to be tested inside the `catch` part of the clause.

## 4.7 Summary

This section has considered how we can incrementally develop a function or set of functions that can be used to solve a programming problem. We have looked at how it is important to make small steps, ensuring that the code runs at each stage and testing its output to see if it gives what we expect. We have also looked at common types of error such as run-time errors, syntax errors and logic errors and what we can try and do to identify and fix them, i.e. to *debug* them. We have also briefly looked at how to handle errors in our code, testing for them explicitly or using try-catch clauses to wrap sections of code that might raise an error.

## 4.8 Further Resources

- The MATLAB help is always a good place to start. From the main help menu for MATLAB, go to section on *Programming scripts and functions* (see screenshot below) and browse the subsections. For example, the part on *Coding and Productivity Tips* lists various examples and how-to documents.



- There is a very good MATLAB coding style set of guidelines by Richard

Johnson at: [http://www.datatool.com/downloads/matlab\\_style\\_guidelines.pdf](http://www.datatool.com/downloads/matlab_style_guidelines.pdf)

## 4.9 Exercises

1. The following code is intended to loop over an array and display each number it contains one at a time. It is quite badly written and needs to be corrected.

```
1 % Display elements of an array one by one.  
2  
3 % Array of values  
4 myArray = [12, -1, 17, 0.5]  
5  
6 for myArray  
7 disp(myArray)  
8 end
```

- (a) Type the code into the MATLAB editor, use the hints from the editor to locate any syntax errors. While there are syntax errors in the code, the MATLAB editor will not allow you to use breakpoints for debugging. Once you have corrected any syntax errors, insert a breakpoint at the start and run the script. Step through the code to identify any further errors and fix them as you go along.
- (b) Identify and fix any style violations in the code, i.e. things that do not prevent the code from running but can be changed to improve the appearance or behaviour of the code.
2. Type the following code into the MATLAB editor, note that it contains errors.

```
clear, close all  
  
% Get 30 random integers between -10 and 10  
vals = randi(21, 1, 30) - 11;  
  
for k = 1:1:numel(vals)  
    if (vals(n) > 0)  
        sumOfPositives = sumOfPositives + vals(n)  
    end  
end
```

```

disp('The sum of the positive values is:')
disp(sumOfPositives)

```

- (a) Identify and explain what the different parts of this piece of code are trying to do. What seems to be the main aim of the code?
  - (b) Find the errors and style violations in the code and fix them using the debugger.
3. A function with errors in it is given below. It aims to evaluate a quadratic polynomial for a given  $x$  value or for an array of given  $x$  values. Type this function into a file and save it with the correct name.

```

function [ ys ] = evaluate_quadratic(coeffs, xs)
% Usage: evaluate_quadratic(coeffs, xs)
%
% Find the result of calculating a quadratic
% polynomial with coefficients given at the x
% value(s) given. Return the result in the
% variable ys which must contain the same
% number of elements as xs.

clear
close all

ys = zeros(1, xs);
ys = a * xs^2 + b * xs + c;

end

```

Now type the following into a script, and call it *testScript.m*. This will be used to test the function above.

```

% Test script for evaluate_quadratic.m
clear, close all

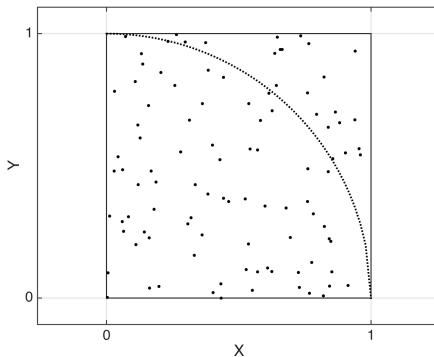
x = 3;
coeffs = [1 2 1];
y = evaluate_quadratic(coeffs, x);
fprintf('f(%0.1f) = %0.1f\n', x, y)

xs = -1:0.5:3;
ys = evaluate_quadratic(coeffs, xs);

```

```
disp(xs), disp(ys)
```

- (a) Try to run the script with the debugger, placing breakpoints in the main function. List the errors in the function and suggest how they should be fixed.
  - (b) Add code to the corrected function to validate the input arguments `coeffs` and `xs` to make sure that they can be correctly processed by the function.
4. This question is about using a method called Monte Carlo simulation to estimate the value of  $\pi$ . The method uses random points chosen inside the unit square in the 2D Cartesian axes, i.e  $0 \leq x \leq 1$  and  $0 \leq y \leq 1$ . This is illustrated in the figure below.



The method picks random points inside the unit square, these are shown by dots in the diagram. Some of these points will lie inside the unit *circle* (which is also plotted) and some of the points will lie outside the unit circle.

We can estimate the area of the quarter circle by finding the fraction of points that are inside it. From this, we can estimate the area of the whole circle and therefore the value of  $\pi$ .

Write MATLAB code to do this. You should break the problem down into a number of sub-tasks and use different functions to carry each sub-task, using an incremental development approach. Use a main function or main script to control the flow of the program. The user should be able to choose how many points are taken and the program

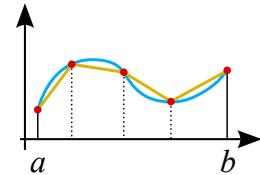
should print a suitable output message.

Run your program and experiment with different numbers of points to see the effect on the estimate of  $\pi$ .

5. The trapezium rule can be used to estimate the integral of a function between a pair of limits. The formula for the trapezium rule is

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} (f(x_1) + 2f(x_2) + 2f(x_3) + \dots + 2f(x_{n-1}) + f(x_n))$$

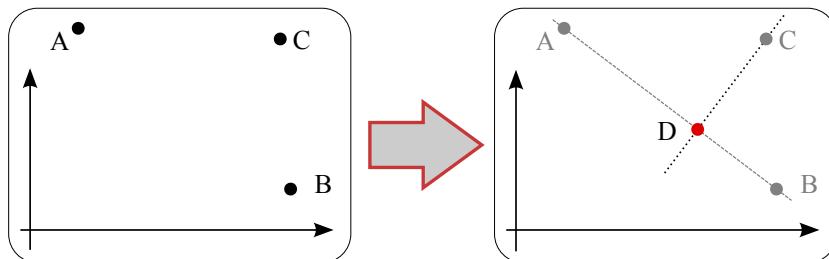
where  $f$  is the function to integrate,  $a$  and  $b$  are the limits of the integration and  $\{x_1, x_2, \dots, x_{n-1}, x_n\}$  are a sequence of points on the  $x$ -axis with  $x_1 = a$  and  $x_n = b$ .  $\Delta x$  is the width of the interval between successive points.



Wikimedia

Write MATLAB code that uses the trapezium rule to estimate the integral of  $f(x) = \sin x$  between a pair of limits and for a given number of points. The user should be able to decide the limits and the number of points, for example they might choose ten points between  $x = 0$  and  $x = \pi/4$ . Use an incremental development approach when writing your solution.

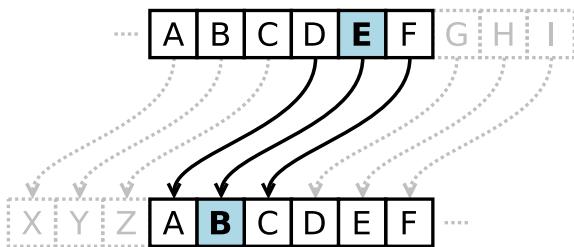
6. A high school geometry problem gives the coordinates of three points, A, B, C in the Cartesian plane. The task is to find a point D on the line AB such that AD is perpendicular to BC. This is illustrated in the figure below.



Write MATLAB code that asks the user for the coordinates of the points A, B, C, and calculates the coordinates of point D.

Break the problem down into sub-tasks and write separate functions for each, using an incremental development approach. Write a single main function or script that controls the flow of the code.

7. A shift cipher is a simple way to make encrypted messages. It works by replacing each letter in a message with another letter that is shifted by a fixed amount in the alphabet (forwards or backwards). This is illustrated below for a backward shift of three.



*Wikimedia*

With this shift, for example, the word EAT is encrypted to become BXQ. If we want to reverse the encryption (decrypt), we can just apply the same encryption process but use a shift in the opposite direction, i.e. a shift of forward three for the BXQ example.

Write MATLAB code that takes a message and encrypts it with a given shift. The user can choose the message and the required shift. You can use the convention that a forward shift is a positive number and a backward one is a negative number. As an example that you can use to check, the message HELLO WORLD encrypted with a shift of +7 is encrypted to become OLSSV DVYSK. Include code to check that your encrypted message can be decrypted correctly afterwards. You should use an incremental development approach when writing your solution.

## **Notes**

## 5 Data Types

*At the end of this section you should be able to:*

- Explain how a data type acts to help interpret the sequences of 1s and 0s stored in the computer's memory
- Identify the fundamental types used in MATLAB: numeric types, characters/strings and Booleans
- Distinguish between integer and floating point numeric types and describe their varying ranges and precision
- Use MATLAB commands to identify the type of a variable
- Convert between types and explain when this is done automatically by MATLAB
- Use cell arrays to contain data of different types
- Explain how different types of array can be created and accessed using different types of bracket: [ ], ( ) and { }

### 5.1 Introduction

As we have already seen, MATLAB can easily handle numeric data and it is well suited to dealing with arrays. In fact, array manipulation is one of its strengths. As discussed in Section 1.5, MATLAB can also process other types of data apart from numeric data, and these can also be accumulated into arrays.

Data types can be roughly divided into fundamental ones for everyday use and more advanced ones. The fundamental types include numeric values, characters, strings and logical values (i.e. ‘true’ and ‘false’). More advanced types include cell data, tables and structs as well as function handles.

Any data in a computer's memory ultimately has to be stored as a sequence of 1s and 0s. These are the bits used to represent the data. A byte is a sequence of eight bits and data items are normally stored in whole numbers of bytes, i.e. in multiples of 8 bits.

Humans tend to think in terms of higher level forms of data, for example as text or numbers, and a *data type* represents a way of interpreting a set of bits in terms of some higher level representation.

In other words, a pattern of 1s and 0s arranged over a set of bytes can repre-

sent different things *depending on how they are interpreted* and this underlies what it means to have different data types. For example, consider a sequence of bits in a byte set to the following pattern: 01000001. If we interpret this as a integer, it can be viewed as the number 65 (because  $2^6 + 2^0 = 64 + 1$ ). We can, however, also interpret the same pattern as a character. Using the ASCII<sup>6</sup> code, the character corresponding to this pattern is ‘A’. That is, we have a correspondence between the number 65 and the character ‘A’. The correspondence continues along the alphabet, for example ‘B’ matches with the same bit pattern that would be used for 66, ‘C’ matches with 67 etc. The lower case letter characters ‘a’, ‘b’, … correspond to the sequence 97, 98, .... A space character matches 32 and other symbols also have similar correspondences.

## 5.2 Numeric Types

We briefly discussed the fundamental data types offered by MATLAB, including numeric types, in Section 1.5. To recap, we distinguish between two broad numeric types: those that are used to represent integers and those for representing floating point numbers (i.e. numbers with a fractional part). Types used for integers can be signed (i.e. can be used for both positive and negative numbers - as well as zero!) or unsigned (i.e. for non-negative numbers only).

The number of bytes used to store an integer can also be varied according to which data type is used. Using more bytes it is possible to represent larger integers. The number of bytes used to store a floating point number can be varied as well: if more bytes are used, as well as being able to represent larger floating point numbers, we can also represent the fractional part more precisely.

The names of the different types that can be used for integers are given in shorthand below. They reflect whether they are signed/unsigned and how many bits each uses:

- `uint8` , `uint16` , `uint32` , `uint64`
- `int8` , `int16` , `int32` , `int64`

This shows how the different types are prefixed with a u if they are unsigned

---

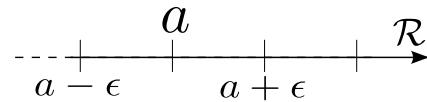
<sup>6</sup>American Standard Code for Information Interchange

and use 1, 2, 4 or 8 bytes depending on how many bits are indicated at the end of the name (8, 16, 32 or 64).

The types that can be used in MATLAB to represent non-integer numbers are called `single` and `double`, which are short for *single precision floating point numbers* and *double precision floating point numbers*; the first uses single precision arithmetic and the second has double precision arithmetic and can represent fractional parts with greater precision. See below for further details of the precision of floating point numbers.

### 5.2.1 Precision for Non-integer (Floating Point) Numeric Types

The precision of a floating point number affects the smallest distinguishable decimal place that can be used. For a given number contained in the computer's memory, the precision can be represented by the difference between a number and the next nearest representable number (above or below).



In the above diagram, a number represented in memory by  $a$  would have two adjacent representable numbers  $a - \epsilon$  and  $a + \epsilon$ . For high precision, the value of  $\epsilon$  is very small, i.e. adjacent representable floating point numbers are very close.

Because of the way floating point numbers are represented in memory, *the size of  $\epsilon$  depends on the size of the number  $a$* . If the value of  $a$  is large, then there will be larger gap to the next representable number than there would be if the value of  $a$  were small.

The built-in MATLAB function `eps` (short for epsilon) can be used to find out the value of  $\epsilon$  for a given number and data type.

If we call it with the name of the data type as an argument, it will return the value of  $\epsilon$  for the number 1 when represented in that data type.

```
>> eps('double')
ans =
2.2204e-16
```

```

>> eps('single')

ans =
1.1921e-07

```

In each case, we obtain the gap between 1 and the adjacent representable numbers when we use `double` or `single` data types.

If we call the `eps` function with a number as its argument, it will return the value of  $\epsilon$  for that number when using the `double` type. For example

```

>> eps(1000)

ans =

1.1369e-13

```

shows that the gap between 1000 (stored as a `double`) and the next representable number is  $1.1 \times 10^{-13}$  which is significantly larger than the gap for 1 ( $2.2 \times 10^{-16}$ , as calculated earlier).

### 5.2.2 MATLAB Defaults to Double Precision for Numbers

MATLAB will default to using double precision for numeric data. For example typing `a = 3` at the command window followed by a call to the `whos` function gives the following output:

```

>> whos
  Name      Size            Bytes  Class       Attributes
    a         1x1              8  double

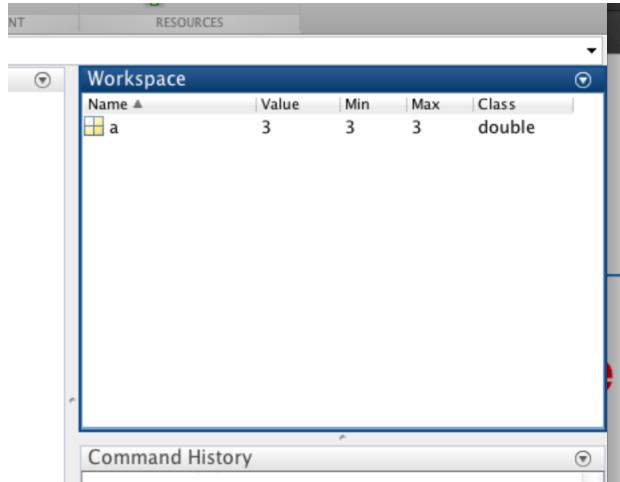
```

Recall that the built-in `whos` function was introduced in Section 1.5. It gives a summary description of the variables in our workspace. In this example, there is only one variable, `a`, in our workspace and the `whos` function tells us

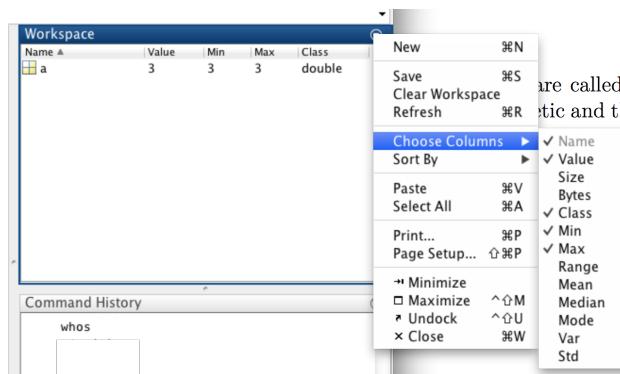
- that MATLAB has used the `double` data type to represent our variable (under the heading `Class`).
- that MATLAB views our variable implicitly as an array of size  $1 \times 1$ . MATLAB defaults to this view of variables as even a scalar value can be viewed as an (admittedly small) array.

- the amount of memory used to represent the variable. In this case 8 bytes are used (i.e. 64 bits).

Similar information is shown by the workspace browser in the MATLAB environment.



We can choose the columns displayed in the workspace window by clicking the small triangle in its top bar to open its context menu (see screenshot above). This allows information such as the number of bytes used for a variable to be shown (see screenshot below).



### 5.2.3 Take Care when Working with Numeric Types other than Doubles

As discussed above, MATLAB will use the double data type for numeric data by default. Most of the time it is fine that MATLAB defaults to use double precision for our variables (even if we only want to work with integers, say) and *the best advice is to leave the default behaviour*.

This default behaviour is only a problem if we want to work with *very* large amounts of data. In such cases, we need to explicitly force MATLAB to use one of the integer types above, or the single type if we still need to work with floating point data. This is to use fewer bytes to store our data. For example:

```
>> a = int16(3);
>> whos
  Name      Size            Bytes  Class       Attributes
    a         1x1                 2  int16
```

One warning here is that the variable will no longer work properly with functions that expect a double as their argument(s) and there are quite a few of these. For example, a call to the square root function such as the following will generate an error:

```
>> a = int16(3);
>> b = sqrt(a);

Undefined function 'sqrt' for input arguments of type 'int16'.
```

whereas if we had simply set `a=3`, the call to `sqrt` would have worked because the variable `a` would have been a double precision floating point number and suitable for passing to the `sqrt` function.

### 5.2.4 Ranges of Numeric Types

Another difficulty that can be encountered when not using MATLAB's default `double` type for numbers relates to the ranges of values that different numeric types can store. We start below by looking at the range of some of the integer data types.

For example, an unsigned single byte integer (`uint8`) can only store values in

the range 0–255. We can check this using the built-in functions `intmax` and `intmin` which accept a character array describing the data type for which we would like to know the minimum and maximum values. For example,

```
>> intmax('uint8')
ans =
    255
>> intmin('uint8')
ans =
     0
```

This tells us that if we use eight bits to represent an unsigned integer, then we can represent the numbers from 0 to 255 inclusive. The number 255 is the decimal value that corresponds to the binary number 11111111, i.e. the one where all the eight bits are set to 1.

If we use 16 bits, then we can represent a larger range of unsigned integers, for example

```
>> disp( [ intmin('uint16') , intmax('uint16') ] )
0   65535
```

Note how the range changes if we want to represent *signed* integers, i.e. both positive and negative numbers:

```
>> disp( [ intmin('int16') , intmax('int16') ] )
-32768  32767
```

The number of different signed 16 bit integers that can be represented is the same as the number of unsigned 16 bit integers, but they have been shifted backwards so that zero is (almost) at the centre.

We need to take care when working with mixed numeric types because we can have unexpected behaviour if we want to represent a number outside the range of the data type we are using. This is shown by the following command window calls:

```
>> a = 250 + 10
a =
    260

>> b = uint8(250 + 10)
b =
    255
```

```
>> whos
  Name      Size            Bytes  Class       Attributes
  a         1x1              8  double
  b         1x1              1  uint8
```

In this code, the intention was to set both variables to a value of 260 but this failed in the case of variable `b` which was forced to be a one byte unsigned integer. Therefore, the assignment to `b` gave an incorrect result due to what is called an *overflow* error.

To re-iterate, if we leave all numeric values in MATLAB's default data type of `double` then the chance of such unexpected behaviour is reduced.

We can check the range of *floating point* numbers using the `realmax` and `realmin` functions. The following calls

```
>> realmax('single')

ans =
  3.4028e+38

>> realmax('double')

ans =
  1.7977e+308
```

show that the biggest single precision number we can represent is  $3.4028 \times 10^{38}$  while the biggest double precision number is  $1.7977 \times 10^{308}$ .

### 5.3 Infinity and NaN (Not a Number)

Two special MATLAB constructs are used to deal with cases where we encounter infinities or numbers that cannot be determined.

`Inf` is what MATLAB uses to represent infinite numbers. For example:

```
>> 1/0

ans =
  Inf
```

The function `isinf` can be used to test if a number is infinite. It can be applied to an array of numbers and will return 1 at those locations containing an infinite value. For example:

```
>> a = [1 2 3];
>> b = [2 0 1];
>> a ./ b

ans =
    0.5000      Inf      3.0000

>> isinf( a ./ b )

ans =
    0      1      0
```

`NaN` is used to represent indeterminate values or things that cannot be interpreted as a number. For example, the expression  $0 / 0$  cannot be interpreted by MATLAB as a number. `NaN` is also returned by functions that attempt to interpret character arrays as numbers in the case when they cannot be interpreted properly. For example, consider the following calls to the built in `str2double` function:

```
>> str2double('2.3')

ans =
    2.3000

>> str2double('i')

ans =
    0.0000 + 1.0000i

>> str2double('pi')

ans =
    NaN

>> str2double('blah')

ans =
    NaN
```

The function `isnan` can be used to test for `Nan` values in a similar way to the `isinf` function and it can be applied to arrays in the same way.

## 5.4 Characters and Strings

We sometimes need to use more than just numbers: Alternatives such as characters and character arrays are often used, for example to report output or communicate with a user. Single quotes are needed to enclose characters and character arrays. At its simplest, we can assign a single character to a variable with a command `s = 'A'`. A call to `whos` allows us to inspect how MATLAB represents this variable:

```
>> whos
  Name      Size            Bytes  Class       Attributes
  s          1x1              2    char
```

This shows us that MATLAB views the type as a size  $1 \times 1$  array whose single element has a character type (`Class` is `char`). We can also see that two bytes are needed to store this single character.

We can also assign an array of characters in exactly the same way, for example `s = 'ABC'`. A call to `whos` gives the following:

```
>> whos
  Name      Size            Bytes  Class       Attributes
  s          1x3              6    char
```

As we can see, we still have an array with elements of type `char` but this time it has size  $1 \times 3$  and takes up 6 bytes of space in memory (2 for each element).

The above illustrates that the sequence of characters '`ABC`' is represented by MATLAB as an array of character elements. We can make this explicit during assignment by using the command `s=['A', 'B', 'C']` which achieves exactly the same result as `s='ABC'`.

Generally, creating character arrays in this way results in an array with size  $1 \times N$  where  $N$  is the number of characters. This means that the arrays are row vectors. So, to concatenate two or more such arrays we can simply

combine them into one longer array.

**Example 5.1.** For example:

```
>> s1 = 'hello';
>> s2 = ' ';
>> s3 = 'world!';
>> s = [s1, s2, s3]

s =
hello world!

>> size(s)

ans =
    1      12
```

Concatenating character arrays row-wise in this way produces a single long character array.

Suppose that we want to have an array that contains three separate character arrays. One way to do this is to concatenate the arrays column-wise (using the semi-colon to separate entries). Doing this can however lead to an error. For example, using `s1`, `s2` and `s3` from the above code, we would get the following:

```
>> s = [s1 ; s2 ; s3]

Error using vertcat
Dimensions of matrices being concatenated are not consistent.
```

This is because MATLAB is trying to place three row vectors, one beneath the other, to create a 2-D array. This is only possible if all of the row vectors have the same length (i.e. they are consistent).

**Example 5.2.** We can modify the above example to make it work if we *pad* the arrays with spaces so that they all end up being the same length. For example:

```
>> s1 = 'hello ';
>> s2 = '      ';
```

```

>> s3 = 'world!';
>> s = [s1 ; s2 ; s3]
s =
hello
world!
>> size(s)
ans =
    3      6

```

where the extra spaces mean we end up concatenating three row vectors vertically, each of length six, to create a  $3 \times 6$  array of characters.

However, since MATLAB version 2017a, it is possible to use a new dedicated string type. This is done by using double quotes when defining variables rather than the single quotes used for character arrays.

**Example 5.3.** For example, consider the following code:

```

>> s1 = "hello";
>> s2 = " ";
>> s3 = "world!";
>> s = [s1 ; s2 ; s3]

s =
3x1 string array

"hello"
" "
"world!"

>> whos
  Name      Size            Bytes  Class       Attributes
  s          3x1             258   string
  s1         1x1              134   string
  s2         1x1              134   string
  s3         1x1              134   string

```

Note that the types of `s1`, `s2` and `s3` are now `string`. With the string

<code>newStr=erase(str,match)</code>	Creates copy of <code>str</code> with all occurrences of <code>match</code> removed
<code>newStr=lower(str)</code>	Creates a copy of <code>str</code> with all characters converted to lower case
<code>newStr=upper(str)</code>	Creates a copy of <code>str</code> with all characters converted to upper case
<code>strArray=split(str)</code>	Creates an array of strings formed from the words in <code>str</code> separated by white space characters

Table 5.1: MATLAB built-in string functions.

type it is possible to create an array `s` of such strings even though they have different lengths.

MATLAB features a number of built-in functions for manipulating strings, and some of the more useful ones are summarised in Table 5.1.

## 5.5 Identifying the Type of a Variable

Before we look at further data types, we will briefly consider how we can identify the type of variables in MATLAB.

As we have seen in Section 1.5, we can use the `whos` command to display a summary description of the variables that are in the workspace. This is useful, but sometimes we may have many variables in the workspace and we may not need to know about some of them. Also, the `whos` command gives information on the size of variables (in bytes) which we may not want.

The built in function `class` can also be used to identify the type of a variable. It produces a character array that describes the variable type.

**Example 5.4.** For example:

```
>> a = 1:5
a =
    1     2     3     4     5

>> b = 'Some text'
b =
Some text
```

```

>> class(a)
ans =
double

>> class(b)
ans =
char

```

*(Note: The use of the word ‘class’ in MATLAB to describe the data type of a variable is at odds with the use of the same word in other programming languages, such as C++ or Python, where the word ‘class’ is reserved for something more specific and distinct from the fundamental data types that the language uses.)*

Another way to identify the type of a variable is to use the built-in `isa` function.

**Example 5.5.** Using the same variables defined in the previous example

```

>> isa(a, 'double')
ans =
1

>> isa(a, 'char')
ans =
0

```

Further methods for testing data types use specific functions and do not need to have a type specified. For example, the function `ischar` can be used to test if a variable is a character or a character array, and the function `isnumeric` can be used to test if a variable’s data type belongs to one of the numeric types (`uint8`, `int64`, etc.).

**Example 5.6.** Again, using the same variables as in the previous example.

```

>> ischar(b)
ans =
1

>> isnumeric(b)

```

```
ans =  
0
```

A full list of the specific functions for testing the type of a variable can be found in the MATLAB help under *Language Fundamentals* → *Data Types* → *Data Type Identification*.

## 5.6 The Boolean Data Type

In programming, we use the term ‘Boolean’ to refer to data and operations that operate with the two values ‘true’ and ‘false’. As we saw in Section 1.5, in MATLAB, the data type that is used for this is known as the `logical` data type. MATLAB accepts the key words `true` and `false` and it represents these key words internally using the numbers 1 and 0 respectively.

**Example 5.7.** Here, we assign a value of `true` to a variable, display it and ask MATLAB what class it is:

```
>> v = true;  
>> disp(v)  
1  
  
>> class(v)  
ans =  
logical
```

Note how it is displayed as the number 1 even though it is a logical (Boolean) data type.

A common way in which logical *arrays* are created is through the use of comparison operators.

**Example 5.8.** For example, we can generate a list of random numbers between 0 and 1 and test which of them is greater than 0.5:

```
>> a = rand([1,5])  
a =  
0.1576    0.9706    0.9572    0.4854    0.8003  
  
>> b = a > 0.5
```

<code>transpose(a)</code>	Return the transpose of the matrix <code>a</code>
<code>inv(a)</code>	Return the inverse of the matrix <code>a</code>

Table 5.2: MATLAB built-in functions for common matrix operations.

```
b =
0      1      1      0      1
```

A call to `whos` gives a description of these variables, where we can see that variable `b` is of the logical data type:

```
>> whos
  Name      Size            Bytes  Class       Attributes
  a         1x5             40  double
  b         1x5              5   logical
```

## 5.7 Matrices

We already introduced the concept of a *matrix* (i.e. a 2-D array) in Section 1.8. We will now briefly return to the concept of matrices in MATLAB and discuss some of the linear algebra operations they allow. The following code illustrates the use of some of the built-in MATLAB matrix functions, which are summarised in Table 5.2.

```
c = a * b
d = a + b
e = inv(a)
f = transpose(b)
```

Here, the ‘`*`’ and ‘`+`’ operators automatically perform matrix multiplication and addition because their arguments are both matrices. If you want to perform *element-wise* arithmetic operations instead you can add a ‘`.`’ before the operator, e.g. note the difference between the following two commands,

```
c = a * b
d = a .* b
```

Note that element-wise addition/subtraction are the same as matrix addition/subtraction.

The distinction between element-wise and matrix operations is an important one, and a common source of errors in programs. Be especially careful about the following potential mistakes (which extend the list given in Section 4.5):

- Matrix multiplication versus *element-wise* multiplication. For example, in Section 4.5 we saw the following code

```
x = [1 2 3];
y = [6 7 8];
z = x .* y;
```

which multiplies  $x$  and  $y$  *element-wise* to give a result of [6, 14, 24]. If we had omitted the '.', i.e.  $z = x * y$ , this would lead to an error as the matrix sizes are not compatible. In other cases, there might be no run time error because matrix multiplication is possible - this would lead to a logic error which can be hard to find. For example:

```
a = [1 2 ; 1 3]; % 2 by 2 array
b = [2 4 ; 6 7]; % 2 by 2 array
% Matrix multiplication but element-wise intended!
z = x * y; % Leads to unpredictable behaviour.
```

- Other operations can also be applied in a matrix sense or in an element-wise sense and care should always be taken to make sure that the correct use is made for the code. These include:
  - Division:  $x / y$  versus  $x ./ y$ . See `help rdivide` for a description of the first.
  - Exponentiation: For a matrix  $x$ , the operation  $x^2$  is a matrix operation (i.e. multiply the whole matrix by itself) while  $x.^2$  squares each element individually.
- Getting the shapes of matrices and vectors wrong during multiplication, e.g.

```
x = [1 2 3];
a = magic(3); % 3x3 magic square.
a * x;
```

gives an error because  $x$  contains a row vector ( $3 \times 1$ ) so pre-multiplying it by matrix  $a$  is not possible. The following would be more correct

```
x = transpose([1 2 3]);
```

```
a = magic(3); % 3x3 magic square.  
a * x;
```

## 5.8 Cell Arrays

Sometimes it can be useful to be able to easily store a number of with different types in a single array. This can be done using a MATLAB-specific data type known as a *cell array*.

We can construct a cell array using curly brackets: ‘{’ and ‘}’. Recall that we normally use the square brackets '[' and ']' to construct an ordinary array (e.g. in Example 5.1).

**Example 5.9.** Here, we initialise a cell array to contain three character arrays with different lengths.

```
a = {'Hi', ' ', 'World!'}
```

A call to `whos` will give some details of what is stored:

```
>> whos  
Name      Size            Bytes  Class       Attributes  
a          1x3              354   cell
```

We say that the array contains three *cells* and each cell contains a character array. The arrays themselves are of lengths 2, 1 and 6. As arrays of type `char`, these could require only 9 or 18 bytes of memory for storage (depending on whether we use one or two bytes per character). Putting these arrays into a cell array, however, leads to 354 bytes being used in this case. Cell arrays are much less efficient in terms of memory usage, but as long as we are not seeking to store too much data, the convenience of a cell array outweighs the extra memory needed.

There are two ways in which we can now access the data in the cell array. One is where we view its elements as cells or sub-arrays of cells. The other way is where we view the elements in terms of their ‘native’ data type - which in our example was the `char` array data type.

**Example 5.10.** In a slight extension of the previous example, we look at access to elements of a cell array. We start by putting three character arrays into a cell array called `a` which is constructed using curly brackets

```
>> a = {'Some text', 'Another text', 'And one more'};  
>> s = a(1);  
>> t = a(2:3);  
>> u = a{3};
```

After constructing the array, different ways of accessing the data in `a` are then illustrated. Variable `s` is assigned a single cell, the first cell in the array `a`. Variable `t` is assigned a sub-array of cells from `a` (the second and the third).

Both variables `s` and `t` are assigned using the round brackets that MATLAB normally uses for accessing array elements: ‘(’ and ‘)’. Variable `u` is defined by accessing the array data in a different way. This time we use curly brackets, ‘{’ and ‘}’, to *access the data contained in a cell* (the third one). That is, `u` is assigned a `char` array rather than a cell (as was the case for `s` and `t`). We can check the status and types of our variables using the `whos` command.

```
>> whos  
  Name      Size            Bytes  Class       Attributes  
  a         1x3             410   cell  
  s         1x1             134   cell  
  t         1x2             276   cell  
  u         1x12            24    char
```

Note how `s` and `t` are both cell arrays ( $1 \times 1$  and  $1 \times 2$  respectively), whereas `u` is a character array of length 12.

It is important to be clear about the use of the different types of brackets (round, square and curly). The different types of brackets can be used to do one of two things:

- Create an array
- Access an element (or elements) in the array.

Not all combinations are possible. For example, in MATLAB we do not use the square brackets to *access* elements in an array, they are only used for *constructing* arrays.

### 5.8.1 Cell Arrays can Contain Mixed Data Types

We have seen above how a cell array can be useful for storing a set of character arrays with varying lengths. In fact, a cell array can be viewed as an indexed list of containers, and each container can contain variables of *any* data type.

**Example 5.11.** Here, we define a cell array with a mixture of types:

```
>> mixedData = {'text', uint8(10) , true, [22, 23, 24, pi]}
mixedData =
    'text'      [10]      [1]      [1x4 double]
```

We can use the curly brackets to access the elements inside the cell array and use the `class` function to identify their types:

```
>> class( mixedData{2} )
ans =
uint8

>> class( mixedData{3} )
ans =
logical
```

The ability of a cell array to contain a mixture of data types may be of interest but is rarely useful in practice.

Of course, a cell array can still have a uniform data type, i.e. all of its cells can contain data of the same type. We have already seen this in the case of the cell array of character arrays above.

**Example 5.12.** Another example is

```
a = {1, 2, 3}
a =
```

```
[1]      [2]      [3]
```

which produces a cell array of doubles.

### 5.8.2 The Different Kinds of Bracket: Recap

To recap, we can carry out two basic operations with an array

- Creating an array
- Accessing elements in an array

and, depending on whether we have a cell array or one that contains another data type, we distinguish the different behaviour of each of the types of bracket: square, curly and round.

#### Creating an Array

- Square brackets ‘[ ]’:  
Creates an array where all entries have the same type, e.g. numeric or char. For example,

```
numArray = [5 62 3];
charArray = ['a', 'b', 'c'];
```

- Curly brackets ‘{ }’:  
Creates a cell array which can contain any type we like in each cell. The cells in the array can contain items of the same type or different types. For example,

```
sameTypeCells = {'Once', 'upon', 'a', 'time'}
diffTypeCells = {'Number', 1}
```

#### Accessing Elements

- Round brackets ‘( )’:  
These can be used to access one or more elements from any type of array. The elements returned always have the same type as the array, i.e. for an array of doubles the round bracket will return doubles, for an array of cells they will return cells. For example, using the arrays created above:

```
>> numArray(2:3) % Second and third elements.
```

```

ans =
62 3

>> y = sameTypeCells([1, 3]) % first and third
y =
'Once'    'a'

>> class( sameTypeCells(2) ) % Compare with below
ans = cell

```

- Curly brackets ‘{ }’:

For accessing elements, these are used specifically with cell arrays. They can be used to obtain the contents of a cell *in its native data type*, i.e. not in the form of a cell. For example,

```

>> disp( sameTypeCells{2} )

upon

>> class( sameTypeCells{2} ) % Compare with above
ans = char

```

## 5.9 Converting Between Types

Now that we have considered some of the fundamental data types, we can look at converting between data types. The simplest way to do this is to use the functions associated with each type.

The simplest everyday conversions are

- Between numbers and characters
- Between numbers and logicals

Some conversions, such as from character to logical, are not allowed.

### 5.9.1 Converting Between a Number and a Character

We stick to the default numeric type `double` in this example. The function we use to convert a number to a character is the `char` function:

```

>> char(66)

ans =

```

B

To go the other way, i.e. convert a character to a number, we can use the *function* named `double`

```
>> double('B')  
  
ans =  
    66
```

These examples show that the conversion is based on the numeric codes used for the characters.

### 5.9.2 Converting Between a Number and a Logical Type

We can convert a number to a logical type using the `logical` function

```
>> logical(1)  
ans =  
    1  
  
>> logical(3.5)  
ans =  
    1  
  
>> logical(-12)  
ans =  
    1  
  
>> logical(0)  
ans =  
    0
```

These show that the conversion from a number to a boolean (logical) type will always return `true` (i.e. 1) if the number is non-zero. It will only produce a `false` result if the number is zero (last example above).

Converting a logical type to a number can again be carried out using the `double` function. There are of course only two cases to check:

```
>> double(true)  
ans =  
    1
```

```
>> double(false)
ans =
0
```

which give the results we expect for converting the `true` and `false` logical values into numbers.

### 5.9.3 Converting Arrays

Converting arrays of data is also possible. For example, the function `double` will attempt to convert any argument it is given to a double precision floating point. If the argument is an array then it should give an array of doubles. For example, assume that an array `arr` contains a list of unsigned 8 bit integers.

```
>> arr = uint8(5:9)
arr =
    5     6     7     8     9

>> class(arr)
ans =

uint8
```

Passing `arr` as an argument to the `double` function gives a conversion

```
>> arrTwo = double(arr)
arrTwo =
    5     6     7     8     9

>> class(arrTwo)

ans =

double
```

We can convert from numbers to characters using the `char` function. For example,

```
>> arr = [104    101    108    108    111];
```

```
>> char(arr)

ans =
hello
```

In other words, the list of numbers initially put into the array, when converted to characters individually, produce a `char` array containing the characters, '`'h'`', '`'e'`', '`'l'`', '`'l'`', '`'o'`'.

On some occasions, MATLAB will convert values automatically if this is required. This can happen, for example, when a mixture of data types are put into an ordinary array<sup>7</sup>. In this case, based on a set of rules, one of the data types contained in the array will get converted to one of the others. For example, if we initialise an array as follows:

```
>> x = 70;

>> s = ['This is number seventy: ' x];
```

we can see that we are putting a character array and a numeric type together in the array. In this case, MATLAB will automatically try to convert the numeric variable `x` to a character array and concatenate it with the first character array.

We need to take care with this as the result may not always be what is expected. If the above character array is displayed, we get the following:

```
>> disp(s)
This is number seventy: F
```

This shows that the conversion has taken place according to the ASCII codes.

The programmer may not have wanted to do this and might in fact have intended to simply print a text representation of the numeric value 70 to the screen. In this case, MATLAB has a specific function for this kind of conversion, `num2str`, which can be used as follows:

```
>> y = num2str(x)
```

---

<sup>7</sup>Not a cell array.

```

y = 70

>> disp( class(x) )
double

>> disp( class(y) )
char

```

This shows that the variable `y` contains the characters required to represent the number 70. In particular, it contains two characters: a '`7`' and a '`0`'. This can be used to modify the displayed text above by writing

```

>> str = ['This is number seventy: ' num2str(x)];

>> disp(str)
This is number seventy: 70

```

Conversely, the built-in function `str2double` will take the character array representation of a number and convert it into a numeric value. For example, compare the different behaviours in the following two command window calls which lead to very different results:

```

>> '3.142' + 1

ans =
    52     47     50     53     51

>> str2double('3.142') + 1

ans =
    4.1420

```

In the first call, the `char` array `['3', '.', '1', '4', '2']` was automatically converted to numeric values based on the ASCII codes of each character, then a value of 1 is added to each. In the second call, the character array `'3.142'` is re-interpreted (converted) directly as the numeric value that is close to pi ( $\pi$ ), then a value of 1 is added.

**Terminology: Casting:** The process of converting from one data type to another is often called *casting*. We say that a variable is cast from one type to another. This cast can be *explicit*, using a function, or *implicit* or automatic.

**Terminology: Strong and Weak Types:** Different programming languages vary in how strict they are regarding the type of a variable. Some languages, such as C++, require the type of variables to be explicitly stated when a variable is defined, for example

```
double x = 3.14;
```

Also, a language may allow little or no conversion of the type of a variable. In this case we say that the language is *strongly typed*.

The examples we have looked at all use the MATLAB language which is fairly permissive about converting types of variables, with many conversions taking place implicitly (invisibly). Therefore, we say that MATLAB is a *weakly typed language*.

**Good Advice:** In general, it is always safer to convert between data types using explicit function calls (`double`, `char`, etc.) rather than relying on automatic (invisible) conversions. This leads to clearer code that is easier to debug.

A full description of the conversions between data types can be found in the MATLAB help under *Language Fundamentals* → *Data Types* → *Data Type Conversion*.

## 5.10 Advanced Data Types

### 5.10.1 Structures

A structure is a commonly used data type in a number of programming languages. It is used to contain a range of different types of information in a single entity. One way of thinking about them is as a record that contains some specific items of data that describe different aspects of an object.

For example, we might want to represent some information relating to tennis players as given in the following table

Player	Country	Born	Grand Slams
Venus Williams	USA	1980	7
Maria Sharapova	Russia	1987	4
Li Na	China	1982	2

Picking the first player in the list, we can build a structure to contain their details. We can start with a call to the built-in function `struct`

```
>> playerA = struct  
  
playerA =  
struct with no fields.
```

The function returns, telling us it has constructed an empty structure for us. Now we can add in the different fields (name, country, year of birth, grand slams won) putting in the details for our chosen player.

```
playerA.name = 'Venus Williams'  
playerA.country = 'USA'  
playerA.born = 1980  
playerA.grandSlams = 7
```

Notice the use of the dot to access or set each field in the structure. We can perform all of the above in a single line if we pass arguments in pairs to the `struct` function - in the form of `field, value, field, value ...`. This becomes:

```
playerA = struct('name', 'Venus Williams', ...  
                 'country', 'USA', ...  
                 'born', 1980, ...  
                 'grandSlams', 7)
```

where we have used dots to split the line for clarity. As described, we can use the dot operator to access the contents of the different fields in the structure

```
>> disp([ playerA.name ' is from ' playerA.country ])  
  
Venus Williams is from USA
```

In the same manner, we can create structs for the other players:

```
playerB = struct('name', 'Maria Sharapova', 'country', ...  
                 'Russia', 'born', 1987, 'grandSlams', 4)  
playerC = struct('name', 'Li Na', 'country', ...  
                 'China', 'born', 1982, 'grandSlams', 2)
```

MATLAB allows for multiple structs to be collected into a single array. One way to do this is to simply concatenate them within square brackets

```
>> players = [playerA playerB playerC]

players =
1x3 struct array with fields:

    name
    country
    born
    grandSlams
```

MATLAB reports that it has constructed a struct array of length three and tells us what fields are available. We can access the fields for the elements in the array using a combination of the round brackets and the dot operator. For example, to get the names of the first two structs in the array, we can call

```
>> players(2:3).name

ans = Maria Sharapova

ans = Li Na
```

and MATLAB prints out the required names one after the other to the command window. Similar calls can be made for the other available fields.

There are further ways in which structs can be created and manipulated, including conversion of data from cell arrays to arrays of structs. Details of this can be found in the MATLAB help under *Language Fundamentals → Data Types → Structures*.

### 5.10.2 Maps

The *Map* data type, also known as *associative arrays* or *dictionaries* in other programming languages, is a useful generalisation of the ordinary arrays we have seen so far.

We are already familiar with the idea of an array as a collection of data items that we can access via an index. We can construct a variety of arrays,

e.g.

```
>> a = [12, 2.4, 50] , b = {'red', 'blue', 'green'} ;
```

gives us two arrays, one numeric array and a cell array When we want to access an element, we use an integer (or an array of integers) as the index (or the indices). For example,

```
>> disp( a(2) )
2.4000

>> disp( b([1,3]) )
'red'    'green'
```

The point here is that *integers* are used when specifying elements in an array.

Another property of arrays is that the integers used to index the data are ordered successive integers. In MATLAB, the integers used for array indices start at 1 so, for example, the elements of the array *b* above can be indexed by the ordered set {1,2,3}. This property of starting array indices at 1 is called *1-indexing*<sup>8</sup>.

The **Map** data type is a generalisation of the array type. It still can be used to collect a number of variables in something like an array but we do not need to use successive integers to index elements.

A map relies on the use of *key-value* pairs. We say that each entry in the map has a unique *key*, and we can use the key to index an element in the map so that we can read or modify that element's *value*.

The key that we use as an index can be more than just the ordered integers: we can use a character array to access an element, a floating point value or a set of integers that are not ordered. The important thing is that each element in the map has a *unique* key - if two elements have the same key then we will not be able to look up individual elements successfully.

---

<sup>8</sup>It is actually fairly uncommon for programming languages to use 1-indexing. MATLAB is a little unusual in this regard and many languages, such as Java, Python, C++, use 0-indexing instead. This means that a length  $N$  array will use the index set  $\{0, 1, \dots, N-1\}$  for its elements.

**Example 5.13.** The following example is given as pseudocode, not in MATLAB code, to give the general idea of maps. A MATLAB specific example will be given afterwards. Three people have the following ages : Ahmed (21), Betty (23), Charley (21). We would like a map to contain the ages and we would like to look up the age of a person by giving their name. If we call such a map *ages*, then we would like to access data using something like the following:

*print ages( ‘Charley’ )* should give 21.

If Betty has a birthday and we would like to update her age, we would like to be able to write something like the following to do this

*ages( ‘Betty’ ) = ages( ‘Betty’ ) + 1*

If a new person arrives, Dharmintra say, who is 25, we would like to be able to add them to the map by simply writing

*ages( ‘Dharmintra’ ) = 25*

**Example 5.14.** Now we consider how the above pseudocode example can be carried out using MATLAB specific tools. Blank maps in MATLAB can be created by a call to the `containers.Map` function.

```
ages = containers.Map;
```

This creates an empty map object for us. If we ask MATLAB to display the map, it gives a brief description:

```
>> disp(ages)
Map with properties:
    Count: 0
    KeyType: char
    ValueType: any
```

This tells us that the count of stored elements is zero as expected. It also says that the map will use `char` arrays as the type for the keys. In other words, it will use character arrays to index its elements.<sup>9</sup> Finally, the description of

---

<sup>9</sup>The `containers.Map` function will use character arrays as the default type for its keys. We will see in the next example how to use a different type.

the map variable `ages` says that it will accept any type as a value. In our case, we will store numeric values for the ages of the people listed.

Now we can actually assign our data to the map. We can do this using round brackets with the names of the people as character arrays:

```
ages('Ahmed') = 21;
ages('Betty') = 23;
ages('Charley') = 21;
```

A call to `disp(ages)` will now give the same output as before but the count will be 3.

We can access the elements in our map and read their values:

```
>> disp(ages('Charley'))
21
```

We can also modify the elements in our map. For example, if Betty has a birthday:

```
>> disp(ages('Betty'))
23

>> ages('Betty') = ages('Betty') + 1;

>> disp(ages('Betty'))
24
```

Or we can add a new person, for example:

```
ages('Dharmintra') = 25
```

and the map will now contain four elements.

The map object behaves like a struct and we can ask for *all* the keys or values for the map by using the dot operator with the fields `keys` and `values`:

```
>> ages.keys
ans =
    'Ahmed'      'Betty'      'Charley'      'Dharmintra'

>> ages.values
```

```
ans =
[21]      [21]      [23]      [25]
```

Note that if we seek to index an element that is not present in the map then we get an error:

```
>> ages('Elijah')

Error using containers.Map/subsref
The specified key is not present in this container.
```

This is because the character array that we have used ('*Elijah*') is not present in the set of keys for the map. We can of course create an entry in the map with this key, in the same way that we did for '*Dharmintra*' above.

It is possible to create a map more efficiently by preparing the keys and corresponding values in advance.

```
>> keys = {'Ahmed', 'Betty', 'Charley', 'Dharmintra'};
>> values = [21, 23, 21, 25];
```

The above shows that the keys for the map are generated as a cell array of character arrays. The values for the elements are given as a numeric array. It is important that the order of the values array matches that of the keys array.

Now we can create our map in one line as follows:

```
ages = containers.Map(keys, values);
```

which uses the same `containers.Map` function but this time passes two arguments to assign the keys and values.

**Example 5.15.** Now we consider a map that uses a different data type for its keys (not a character array).

Suppose we have the same people with the ages given in the previous example. Now we want to build a map that will give us all the people for a specific age. That is, we want to use the numeric age as the key for indexing elements.

In this case, we can construct an empty map as before, but this time we can tell MATLAB specifically that we would like to use a numeric type for the key. We will use the `double` data type in case we ever need to use fractional ages (e.g. ‘Adrian’, 13.75 years).

When we specify the type of the key, we also need to specify the value type. Fortunately, we can be non-specific for the value type and can simply say ‘any’.

```
>> people = containers.Map('KeyType', 'double', ...
    'ValueType', 'any')

people =
Map with properties:
    Count: 0
    KeyType: double
    ValueType: any
```

Now we can populate the elements of our map as follows:

```
>> people(21) = { 'Ahmed' , 'Charley' };
>> people(23) = { 'Betty' };
>> people(25) = { 'Dharmintra' };
```

Note that we have used a cell array of character arrays for each of the values. This allows us to store the name of more than one person if we need to, although some of the cell arrays contain only one character array.

We can now access elements in the map by using a numeric key. For example, `people(21)` will return all those with age 21.

```
>> disp( people(21) )
'Ahmed'      'Charley'
```

A missing key will still generate an error:

```
>> people(40)

Error using containers.Map/subsref
The specified key is not present in this container.
```

because there is nobody aged 40 in the map.

Using a key of the wrong type will also generate an error. Here, we attempt to access an element in the `people` map using a character array

```
>> people('Fred')

Error using containers.Map/subsref
Specified key type does not match the type expected for
this container.
```

**Example 5.16.** Here, we extend the previous example to show how we can check if a map contains an element with a given key.

If we need to add a new person to the map, then what we need to do will depend on whether or not it already contains other people with the same age. We need to check if the key that we need to use (the person's age) exists in the keys of the map. If a new person (for example, Elijah, 30 years old) needs to be added to the map of the previous example, we can inspect the keys of the map using a built-in function to see if there is a key corresponding to their age:

```
>> people.keys

ans =
[21]      [23]      [25]
```

There is no key for age 30 so we can simply create a new entry in the map:

```
>> people(30) = { 'Elijah' }
```

If another person arrives and their age matches one we have already seen, then we need to be careful not to over-write the person or persons already listed for that age. Say Fiona is 21, so we need to append her name to the cell array of character arrays that contains all those already listed under that age. We can do this using the square brackets to append her name to the cell array as follows:

```
>> disp( people(21) )
'Ahmed'    'Charley'

>> people(21) = [ people(21) 'Fiona' ]
```

```
>> disp( people(21) )
    'Ahmed'      'Charley'      'Fiona'
```

There is a built-in function that tells us if a particular key is being used or not: it is called `isKey`. We can use it to test whether or not a key is present when we want to add new data items as in the above example.

The following gives a very brief outline of the structure of some code that could be expanded to add a new item to a map where we want to append values to existing ones rather than overwrite them. In our case, the variable `someAge` is the age (key value) of the person we would like to include.

```
if ( people.isKey(someAge) )
    % Code to append new value to existing values for this key
else
    % Code to add a new key-value pair to the map
end
```

The comments in the `if` clause indicate where code needs to be written to either introduce a new value or append a value to an existing set of values.

## 5.11 Summary

In this section we have looked at how we need to apply a data type to interpret sequences of ones and zeros in the computer's memory. The same pattern of ones and zeros will be interpreted differently depending on which data type we use, whether it is a numeric, character, or a more sophisticated data type.

As well as looking at the fundamental numeric, character and Boolean data types, we have looked at more advanced types such 1-D and 2-D arrays, cells and structs. We have seen that for all of these, the basic data type can be collected into an array and indexed with (in MATLAB's case) integers starting from one (1-indexing).

For the numeric types we have considered the range of the values it is possible to represent and how this varies with the number of bytes used. We have also looked at the precision with which we can represent floating point numbers depending on whether we use single or double precision floating point numeric

values. We have also introduced the special `Nan` and `Inf` constructs to represent ‘Not a Number’ and ‘infinity’ respectively.

We have looked at some of the possibilities for converting a variable between data types and how, for numeric types, MATLAB defaults to using doubles. Accepting this default behaviour is generally the best advice as many functions in MATLAB expect to use doubles.

As we have seen, arrays are widely used in MATLAB and, depending on the data type of elements in the array, we have looked at the different operations that can be carried out, using different forms of bracket, with regard to accessing elements in an array or creating one.

We have also looked at map data types and how they generalise arrays by allowing us to index elements using other types than 1-indexing integers. We have looked at structs and how they can be used to collect a set of details about an entity or object into a single data type.

## 5.12 Further Resources

As described already, the MATLAB help is fairly comprehensive. Open the help from the GUI, navigate to *MATLAB → Language Fundamentals → Data Types* and choose from the sections it contains.

For further descriptions of the ASCII and Unicode encodings for characters, the Wikipedia entry is a good place to start.

## 5.13 Exercises

1. (a) Initialise your own  $1 \times 3$  array of numbers. Try and choose them so there is at least one integer that matches the ASCII code for an alphanumeric character.  
(b) Make an explicit conversion of the array to a character array. If some of the characters appear to be missing, why should this occur?  
(c) Make an explicit conversion of the array to the `logical` type.  
(d) Make an *implicit* conversion of the array’s contents to a character. (Hint: we can use concatenation).

2. (a) Make up a one row character array containing '`some string`' using square brackets and individual characters
- (b) Make the same character array using single quotes:
- (c) Make a two row character array containing '`some`' and '`string`' on separate rows. What do we need to do to make sure these two words can be fitted into a two row array?
- (d) Make an explicit conversion of the character array of the last part to a numeric type.
- (e) Make an implicit conversion of the character array to a numeric type.
- (f) For each of the following character arrays, try and generate a corresponding numeric value using the `str2double` function. Give the reason why some of these conversions can fail and how we can test for it.

```
'2.3'
'e'
'0.7'
'X.3'
'1.3e+02'
```

- (g) Make an array of string types containing the strings '`Baa`', '`Baa`', '`Black`' and '`Sheep`' on separate rows. Write MATLAB code using a `for` loop to create a new string array which is a copy of the first array but with all characters converted to lower case.
3. Write a MATLAB *m*-file to define the matrices  $A$ ,  $B$  and  $C$  as follows:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 3 & 2 & 1 \\ 3 & 2 & 4 \end{pmatrix}, B = \begin{pmatrix} 4 & 4 & 4 \\ 2 & 2 & 2 \\ 1 & 1 & 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 0 & 1 \\ 2 & 1 & 3 \\ 1 & 0 & 1 \end{pmatrix}.$$

Now modify the script to compute the following:

- (a)  $D = (AB)C$
- (b)  $E = A(BC)$
- (c)  $F = (A + B)C$

(d)  $G = A + BC$

Save all variables to a MATLAB *MAT* file.

4. Systems of linear equations such as

$$\begin{aligned}5x_1 + x_2 &= 5, \\6x_1 + 3x_2 &= 9,\end{aligned}$$

can be solved by expressing the equations in matrix form, i.e.

$$\begin{pmatrix} 5 & 1 \\ 6 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \end{pmatrix}$$

and then rearranging to solve for  $x_1$  and  $x_2$ :

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 & 1 \\ 6 & 3 \end{pmatrix}^{-1} \begin{pmatrix} 5 \\ 9 \end{pmatrix}$$

Write a MATLAB *m*-file that uses matrices to solve the system of equations given above.

5. The `intmax` function can be used to show that the largest unsigned sixteen bit integer is 65535. Show the numeric calculations that prove this is the largest possible unsigned sixteen bit integer.
6. A set of numeric variables are defined as follows:

```
a = pi
b = uint8(250)
c = int32(100000)
d = single(17.32)
```

- (a) Give the full name of the type for each of the variables above.

- (b) An array is created by concatenating the variables above:

```
X = [ a b c d ]
```

Identify the type of the resulting array

- (c) Create the array by concatenating different subsets of the elements in different orders:

```

X = [a d]
X = [d a]
X = [a d b]
X = [a d c]
X = [a d c b]
X = [a d b c]

```

Check the type of each result each time.

Identify the rule that MATLAB uses for deciding on the type of the output:

- when the array contains one or more integer types.
- when the array contains a mixture of floating point types and no integer types.

- (d) A fifth variable is assigned a character: `e = 'M'`. What is the data type of the resulting array if variable `e` is included in any of the concatenations above?
7. (a) Make a cell array called `c` that contains the following eleven character arrays
- ```

'the'
' '
'cow'
' '
'jumped'
' '
'over'
' '
'the'
' '
'moon'

```
- (b) Write code at the command line that gives the cell at index 7 in the array `c`. Make sure you use the correct type of brackets. Use the expression `class(ans)` to confirm that a cell is returned.
- (c) Write code that gives the cells with odd indices in the array `c`, i.e. the cells at index 1, 3, 5, etc. How many cells are returned?
- (d) Write code that gives the cells with even indices in the array `c`.

- (e) Write down code to return the character array *contained* in the cell at index 5 of the array c
  - (f) Write code that returns the last character of the character array contained in the cell at index 5 in array c
  - (g) Write code that creates a new cell array c2 that is a copy of c but with all instances of the letter ‘e’ removed from words with length greater than 3.
  - (h) Make a new cell array d that contains at least three different data types.
8. Here are the names of five animals: Aardvark, Lion, Lemur, Camel, Elephant.
- (a) By padding the names with spaces where necessary, show how you can construct a 2-D character array to contain all the five names in 1-D character arrays. Assign this to a variable called a.
  - (b) Now show how we can put the five names into a cell array of character arrays. Assign this to a variable called c.
  - (c) Write the code that is needed to replace the Lemur with a Dog in the 2-D char array a.
  - (d) Write the code that is needed to replace the Lemur with a Dog in the cell array c.

The questions beyond this point are on struct and map data types.

9. A list of animals (vertebrates) and their classes are given below:

| animal | class   |
|--------|---------|
| zebra  | mammal  |
| gecko  | reptile |
| mouse  | mammal  |
| robin  | bird    |
| heron  | bird    |
| cobra  | reptile |
| hyena  | mammal  |

- (a) Show how a map can be constructed so that the name of each animal is a key and its class is the corresponding value. Assign the map to a variable called `classes`.
- (b) Write the code for adding a horse to the map `classes`.
- (c) Write code for iterating over the keys of the map. Hint: assign the keys to a variable and use the built-in function `numel` to find out how many elements it has.
10. Chemical elements can be described by their *name* or their *symbol*. Each element also has an *atomic number* (indicating the number of protons in its nucleus).
- (a) Show how a struct can be created to represent these three pieces of information for an element. Use the italics above for the field names. Give an example for an element of your choice.
- (b) Repeat the previous part for two more elements, and show how you can collect them into a single array of structs.
- (c) Show how we can access
- the name of the second element in the array
  - the symbols of the first and last elements in the array
11. This question is a continuation of the map one in Example 5.16. Write a function to add a new person with a given age to the map. The function should accept three arguments. The first argument is a map that is assumed to have the correct key and value data types (numeric and cell array of character arrays). The second argument represents the key, consisting of the age of the person to add. The last argument is the name of the person to add. The code will need to check whether

the key is already used. A basic structure of the code is given at the end of Example 5.16.

Test your function by adding new persons with and without ages that are already contained in the map.

## **Notes**

# 6 File Input/Output

*At the end of this section you should be able to:*

- Use built-in MATLAB functions to save and load workspace variables
- Use a range of built-in MATLAB functions to read data from external text or binary files
- Use a range of built-in MATLAB functions to write data to external text or binary files

## 6.1 Introduction

In Section 1.6 we introduced some MATLAB commands that can be used to perform loading and saving of data from/to external files. This type of operation is very common, particularly when we are dealing with complex programs that need to process large amounts of data. In this section we will look in more detail at the issue of *file input/output*.

## 6.2 Recap on Basic Input/Output Functions

First, let us recap on what we know already. Section 1.6 introduced the following simple file input/output functions:

- `save`: Save data to a MATLAB *MAT* file;
- `writematrix`: Write data to a delimiter-separated text file;
- `load`: Load data from a MATLAB *MAT* file or delimiter-separated text file.

Therefore, we have seen two different types of external file so far: *MAT* files and text files. *MAT*-files are MATLAB specific and are used to save parts of the MATLAB workspace. When using *MAT* files the `save` and `load` functions are all we need to know. For the rest of Section 6 we will focus in more detail on functions we can use when dealing with other types of file.

## 6.3 Simple Functions for Dealing with Text Files

We have already seen in Section 1.6 that the `writematrix` command can be used to write numeric data to text files. Text files are files which consist of a sequence of characters, normally stored using the ASCII coding system.

We saw that it was possible to change the delimiter used to separate data values in the file, for example as follows,

```
>> f = [1 2 3 4 9 8 7 6 5];
>> writematrix(f, 'testtab.txt', 'Delimiter', 'tab');
>> writematrix(f, 'testspace.txt', 'Delimiter', '');
```

The full list of values that can be supplied along with the '**Delimiter**' argument to `writematrix` is given in Table 6.1.

| Argument        | Delimiter    |
|-----------------|--------------|
| ', '<br>'comma' | Comma        |
| ' '<br>'space'  | Space        |
| 't'<br>'tab'    | Tab          |
| '; '<br>'semi'  | Semi-colon   |
| ' '<br>'bar'    | Vertical bar |

Table 6.1: Delimiters that can be specified for writing data to text files using `writematrix`.

For reading data from text files, although the `load` function can be used to read numeric data, a more flexible function is `readmatrix`.

**Example 6.1.** We illustrate the use of `readmatrix` using the following example. Physiological data has been exported from an MR scanner to a file called *scansphyslog.txt* in the format shown below.

```
-33 -53 25 -6 0 -1815 0 0 0 0000
-35 -56 24 -9 0 -1815 0 0 0 0000
-37 -59 22 -12 0 -1815 0 0 0 0000
-39 -61 20 -15 0 -1840 0 0 0 0000
-40 -61 18 -18 0 -1840 0 0 0 0000
-40 -60 17 -20 0 -1840 0 0 0 0000
-39 -58 15 -21 0 -1840 0 0 0 0000
```

```

-36 -55 15 -21 0 -1840 0 0 0 0000
-32 -50 15 -20 0 -1871 0 0 0 0000
-26 -44 17 -17 0 -1871 0 0 0 0000
-20 -37 19 -13 0 -1871 0 0 0 0000
-11 -30 23 -8 0 -1871 0 0 0 0000
-1 -22 29 -2 0 -1871 0 0 0 0000

```

The 6<sup>th</sup> column of this data file represents a signal acquired by a respiratory bellows that is sometimes used for respiratory gating of MR scans. The code example shown below will read the data from the file and then extract the bellows signal.

```

data = readmatrix('scanphyslog.txt', 'Delimiter', ' ');
bellows = data(:, 6);

```

Note that we specify the delimiter to use when reading the data (a space in this case) using the '**Delimiter**' argument followed by the delimiter, similar to `writematrix`. The `readmatrix` function will automatically work with files that use commas as the delimiter, but if we wish it to use an alternative delimiter we must specify it in this way.

In this example, the `readmatrix` function will return a  $13 \times 10$  2-D array of the physiological data. The second line of code extracts the 6<sup>th</sup> column, which represents the bellows data.

## 6.4 Reading from Files

The functions described in Section 6.3 can be very useful but they are quite limited. One important limitation is that they only work if the data consist of purely numeric values. For more complex data files these functions will not be enough for our needs. In this section we will introduce some of the more flexible functions provided by MATLAB for file input/output.

Before we proceed, a key concept to understand is the need to *open* a file before using it, and to *close* it after use. You can think of this as being similar to using a real, physical file: in this case it is obvious that the file should be opened before looking at or changing its contents. It's the same with computer files: there are specific functions that we can use to open or close external files and these must be used correctly. (Note that the simple functions described in the previous section do not need the file to be opened

and closed - this is performed automatically.)

**Example 6.2.** Let us illustrate this concept with an example. The following piece of code is from the MATLAB documentation for the `while` statement and it illustrates the concept of opening and closing a file as well as introduces some useful new functions.

```
% open file
fid = fopen('test.m', 'r');

% initialise count
count = 0;

% loop until end of file
while ~feof(fid)

    % read line from file
    tline = fgetl(fid);

    % is it blank or a comment?
    if isempty(tline) || strncmp(tline, '%', 1)
        continue;
    end

    % count non-blank, non-comment lines
    count = count + 1;
end

% print result
fprintf('%d lines\n', count);

% close file
fclose(fid);
```

This code will count the number of lines in the file *test.m*, skipping all blank lines and comments. The `fopen` and `fclose` statements open and close a file respectively. For most MATLAB file input/output functions it is essential that we open a file before use and it is good practice to close it afterwards. The statement `feof` checks to see if the end of file has been reached. Recall that the *tilde* operator (`~`) means *not* in MATLAB (see Table 2.1). The `fgetl` function reads one line from the file. This function returns the line as an array of characters. Therefore, the `while` loop will continually read lines

from the file so long as the end of file is not reached. When it is reached, execution passes to after the `while` loop's `end` statement.

Next, two new built-in functions are used together with an `if` statement:

- `isempty(x)`: returns whether or not the array `x` is empty;
- `strcmp(x,y,n)`: returns `true` if the character arrays `x` and `y` are identical up to the first `n` characters;

These two functions are used to check if the current line either: (a) is empty, or (b) starts with the MATLAB comment symbol, `%`. If either of these conditions is true (recall that `||` means *or* - see Table 2.1) then the `continue` statement will cause execution to pass to the beginning of the `while` loop again. If both of these conditions are not true then 1 is added to the `count` variable and execution passes to the beginning of the loop.

Finally, the `fprintf` function is used to display the number of lines to the command window. We first introduced `fprintf` in Section 3. It can be used as an alternative, and more flexible version, of the `disp` statement. Here it is used to display some text and a variable (`count`) to the command window. However, it can also be used to write to external files, as we will see later (Section 6.5).

So, to summarise, we have learnt the following new MATLAB functions for file input/output:

- `fid = fopen(filename, permission)`: Open a file called `filename` for reading/writing, giving it the file identifier `fid`. The `permission` argument is a character specifying how the file will be used, e.g. '`r`' for reading, '`w`' for writing or '`a`' for appending;
- `fclose(fid)`: Close a file with file identifier `fid`;
- `line = fgetl(fid)`: Read a single line from the text file with identifier `fid`, putting the result into the character array variable `line`;
- `feof(fid)`: Return a Boolean value indicating whether the end of the file with identifier `fid` has been reached.

Look at the MATLAB documentation for full details on any of these functions.

Now let us consider another example. Example 6.2 illustrated how entire lines could be read from a text file as 1-D arrays of characters. However, it is often the case that a line of data contains a number of individual values

which should be stored separately for further processing. In such cases a more flexible file input function is needed.

**Example 6.3.** In this example we would like to read some data about patients' blood pressures from a text file. The file *sys\_dia\_bp.txt* contains the following data:

```
BP data
51 85
88 141
67 95
77 111
68 115
99 171
80 121
```

First, there is a header line which just contains the text "BP data". After this, in each row of the file, the 1<sup>st</sup> column represents the systolic blood pressure of a patient and the 2<sup>nd</sup> column represents the diastolic blood pressure of the same patient. The following piece of MATLAB code can be used to read all of the data from the file into a single 2-D array called data.

```
% open file
fid1 = fopen('sys_dia_bp.txt', 'r');

% skip header line
line = fgetl(fid1);

% read systolic and diastolic blood pressure
data = fscanf(fid1, '%d %d', [2 inf])

% close file
fclose(fid1);
```

Note that, as before, we open the file before reading any data, and close it after we have finished reading. We use the `fgetl` function that we saw in Example 6.2 to read (and discard) the text header line. Next, the `fscanf` function is used to read all rows of data from the file in a single statement. Three arguments are provided to `fscanf`:

- the identifier of the file from which the data should be read: `fid1`;

| Specifier | Field Type                             |
|-----------|----------------------------------------|
| %d        | Signed decimal integer                 |
| %u        | Unsigned decimal integer               |
| %f        | Floating point number                  |
| %c        | Character                              |
| %s        | String (i.e. a sequence of characters) |

Table 6.2: Field specifiers for use in format strings of file input/output functions.

- a *format string* that specifies the format of the data fields to be read: %d means a signed decimal number, so fscanf will read pairs of decimal numbers from the file at a time;
- the dimensions of the output array returned by fscanf: [2 inf] means that there will be two rows (representing the pairs of decimal numbers) and an unlimited number of columns, i.e. the function will read pairs of decimal numbers until it reaches the end of the file.

In this example, seven pairs of decimal numbers will be read in, as there are seven lines of numeric data in the input file. Therefore, the output variable data will be a  $2 \times 7$  array.

The *format string* that we saw in the above example was first introduced in Section 3.3. As well as reading decimal numbers, format strings enable fscanf to be used to read in a range of other data types. A summary of some of the more common *field specifiers* that can be used in the format string is provided in Table 6.2. Note, however, that fscanf cannot be used to read *mixed* data types from a single file. For example, it is not possible to read a character array followed by a decimal integer using fscanf. The reason for this is that fscanf always returns an array data structure, and arrays cannot contain elements with different data types. It is possible to read multiple numeric data types (e.g. a signed decimal and a floating point) but in this case fscanf will perform an automatic conversion of one of the types to ensure that the resulting array contains only a single type (e.g. it will convert the signed decimals to floating point numbers).

In fact, Example 6.3 could also have been implemented using readmatrix, as the following code illustrates.

```
data = readmatrix('sys_dia_bp.txt', 'NumHeaderLines', 1);
```

The final two arguments of `readmatrix` indicate that the file contains one header line, which will be ignored when reading the numeric data.

However, `fscanf` is more flexible than `readmatrix` and can be used to read a wider range of data formats, as the following example illustrates.

**Example 6.4.** Suppose now that the file `sys_dia_bp2.txt` contains the following data:

```
BP data
Sys 51 Dia 85
Sys 88 Dia 141
Sys 67 Dia 95
Sys 77 Dia 111
Sys 68 Dia 115
Sys 99 Dia 171
Sys 80 Dia 121
```

This is the same numerical data as we saw in Example 6.3 but with extra text between the numerical values. In biomedical engineering it is common to have to deal with data exported from a range of devices with different data formats, so this example is quite realistic. It would not be straightforward to use `readmatrix` to read this data because each row of data has mixed data types, i.e. character arrays and numbers. However, with `fscanf` the solution is quite simple.

```
% open file
fid = fopen('sys_dia_bp2.txt');

% skip header line
line = fgetl(fid);

% read systolic and diastolic blood pressure
data = fscanf(fid, '%*s %d %*s %d', [2 inf])

% close file
fclose(fid)
```

All we have done here is to modify the format string of the `fscanf` function: we have added in two extra character array fields before each decimal integer field (these correspond to the `Sys` and `Dia` text in the data file). Note the

\* symbol between the `%` and `s`: this causes MATLAB to read the specified field but not to include it in the output array. This is necessary because, although it can read files containing mixed data types, `fscanf` can only return an array of values of a single type.

Sometimes, however, it is desirable to read in data of mixed types and store it all in a single variable. To do this, we need an alternative file input function: `textscan`. The following example illustrates its use.

**Example 6.5.** Suppose now that the file `sys_dia_bp3.txt` contains the patients' names as well as their blood pressure data, and that we wish to read in all of the data, i.e. names and blood pressures:

```
Joe Bloggs 51 85
Josephine Bloggs 88 141
Zhang San 67 95
Anders Andersen 77 111
Erika Mustermann 68 115
Ola Nordmann 99 171
Jan Kowalski 80 121
```

It is not possible to read all of this data using `fscanf` because the resulting array could not contain both character arrays and numeric data. However, `textscan` returns a *cell array* rather than a normal array (see Section 5.8), and cell arrays can contain mixed data types. Therefore, it is possible to read this data using `textscan`. Examine the code shown below.

```
% open file
fid1 = fopen('sys_dia_bp3.txt', 'r');

% read systolic and diastolic blood pressure
data = textscan(fid1, '%s %s %d %d');

% close file
fclose(fid1);
```

Note that `textscan` has a similar format string to `fscanf`. However, in this case it is permitted to mix up character array data fields (`%s`) with decimal number fields (`%d`) and all will be included in the returned data. The data type of the variable `data` returned by `textscan` will now be a  $1 \times 4$  cell array. The first and second elements of `data` contain arrays of the first and

second names of the patients respectively. In fact, since the lengths of the character arrays vary these first two elements are themselves cell arrays. The third and fourth elements of data are normal arrays of integers representing the systolic and diastolic blood pressures of the patients. Try entering this code into a MATLAB *m*-file, running it and looking at the data variable.

The `textscan` function is quite flexible and can also be used to read in data of mixed types and with multiple delimiters.

**Example 6.6.** For example, let's change the format of the data, in the file `sys_dia_bp4.txt`:

```
Joe Bloggs;51;85
Josephine Bloggs;88;141
Zhang San;67;95
Anders Andersen;77;111
Erika Mustermann;68;115
Ola Nordmann;99;171
Jan Kowalski;80;121
```

If we were to use the code shown in Example 6.5 to read in this data it would read it incorrectly because the default delimiter for `textscan` is white space, i.e. spaces, tabs or newline characters. As a result the 2<sup>nd</sup>, 3<sup>rd</sup> and 4<sup>th</sup> data fields (i.e. the last name, systolic blood pressure and diastolic blood pressure) would be read as a single character array. The following code shows how we can change the delimiter(s) for `textscan`.

```
% open file
fid1 = fopen('sys_dia_bp4.txt', 'r');

% read systolic and diastolic blood pressure
data = textscan(fid1, '%s %s %d %d', 'delimiter', ' ;');

% close file
fclose(fid1);
```

Note that the multiple characters in the single character array after the '`'delimiter'`' argument (i.e. a space and a semicolon) specify that either of these two characters can be used as the delimiter. This allows the data fields to be read in correctly as before.

So far all of the examples we have seen have read data from, or written data to, text files. In computing, it is common to distinguish between text files and *binary* files. In fact, strictly speaking, all computer files are binary files since the data is always stored as a series of 1s and 0s (known as binary digits, or *bits*). However, it is common to refer to files as text files if the sequence of bits can be interpreted as character data using a coding system such as ASCII<sup>10</sup>. Any other file is referred to as a binary file. Binary files tend to make more efficient use of disk space than text files, but can be more problematic to read. Whatever their pros and cons, it is a fact that both types of files are in common use so we need to know how to read/write data from/to both types.

MATLAB permits us to write data to and read data from binary files. The main function to know about for reading data from binary files is `fread`. The use of `fread` is illustrated in the following example.

**Example 6.7.** Suppose that the file `heart_rates.bin` is a binary file containing the following heart rate data stored as a series of 16-bit integers: [65 73 59 101 77 90 68 92]. How can we write code to read these values into a MATLAB array? Consider the program listing shown below.

```
% open file
fid1 = fopen('heart_rates.bin', 'r');

% read data from binary file as 16-bit integers
hr = fread(fid1, inf, 'int16')

% close file
fclose(fid1);
```

In this example, the `fread` function takes three arguments:

- the identifier of the binary file from which the data should be read: `fid1`;
- the number of data values to be read: `inf` means read until the end of the file;
- the data type of the elements to be read: `int16` refers to 16-bit integers (see Section 1.5).

---

<sup>10</sup><http://en.wikipedia.org/wiki/ASCII>

Note that if we didn't specify `int16` as the data type `fread` would assume that each byte (8 bits) represented a number and so the data would not be read correctly. Generally, when reading data from binary files it is important to know and specify the type of the data we are reading since different data types use different numbers of bytes on the computer's hard disk.

## 6.5 Writing to Files

Now we will introduce some MATLAB functions that can be used for writing data to binary or text files. For binary files, the main function for writing data is `fwrite`, the use of which is illustrated in the following example.

**Example 6.8.** The code shown below can be used to generate the binary file that was read in using `fread` in Example 6.7.

```
% define data array
hr = [65 73 59 101 77 90 68 92];

% open file
fid1 = fopen('heart_rates.bin', 'w');

% write data to binary file as 16-bit integers
fwrite(fid1, hr, 'int16');

% close file
fclose(fid1);
```

In this example, we provide three arguments to `fwrite`:

- a file identifier: `fid1`;
- an array containing data to be written to the file: `hr`;
- the data type to use when writing: `int16`.

Note that there is no argument for specifying how much of the array should be written - the entire array will be written. If we only want to write part(s) of the array we need to modify the value of the second argument ourselves, or use multiple `fwrite` commands. Note also that, using a single `fwrite` command, only values of a single data type can be written. If we want to write mixed data to a binary file we can use multiple `fwrite` commands.

The main function for writing data to text files is `fprintf`, which we saw

earlier in Example 6.2. The `fprintf` function is actually very powerful and flexible, and can be used to write data of a range of different data types, as well as specify the formatting of that data. Consider the following example.

**Example 6.9.** The code shown below can be used to generate the mixed data file that was read in using `textscan` in Example 6.5.

```
% define cell array of names
names = {'Joe Bloggs','Josephine Bloggs', ...
          'Zhang San','Anders Andersen', ...
          'Erika Mustermann','Ola Nordmann', ...
          'Jan Kowalski'};
```

```
% define array of blood pressure values
bps = [51 85
       88 141
       67 95
       77 111
       68 115
       99 171
       80 121];
```

```
% open file
fid1 = fopen('sys_dia_bp.txt', 'w');
```

```
% write data to file
for x=1:length(names)
    fprintf(fid1, '%s %d %d \n', names{x}, bps(x,1), ...
            bps(x,2));
end
```

```
% close file
fclose(fid1);
```

Here we use a cell array called `names` to store the names of the patients. Each element of `names` is a normal 1-D array of characters. Because the names have different lengths we can't store them using a normal 2-D character array. The blood pressure values are stored using a normal 2-D array of numbers. After opening the file for writing, we use a `for` loop to iterate through all patients (determined by finding the length of the `names` cell array). In each iteration, the `fprintf` function is used to display a character array (the patient's name), followed by two decimal values (the systolic and diastolic

blood pressures). We supply five arguments to `fprintf` in this example:

- the file identifier: `fid1` - if this is omitted the data is displayed in the command window as we saw in Example 6.2;
- the format string, in this case containing three fields (a character array and two decimals);
- the remaining three arguments contain the data for each field in the format string - so `names{x}` is used for the character array field, `bps(x,1)` for the first decimal and `bps(x,2)` for the second decimal.

Finally, after the loop has finished executing, the output file is closed.

When specifying a format string for `fprintf`, we can make use of the same field specifiers as we can when using `fscanf` (see Table 6.2). In addition, we can provide extra information to alter the display format of the fields by adding extra characters in between the `%` and the data type identifier. For example, we can specify the field width and precision of numeric values, e.g.

```
fprintf(fid1, '%10.3f \n', x);
```

This code will display the value of the variable `x` as a floating point number using a field width of 10 characters and a precision of 3 (i.e. 3 numbers after the decimal point). In addition, a number of flags can be added before the field width/precision values. Adding the `-` character will cause the displayed number to be left-justified (the default is right-justified). Adding the `+` character will cause a plus sign to be displayed for positive values (by default only a minus sign is added for negative values). Adding a space or `0` will cause the number to be left-padded with spaces or zeroes up to the field width. For example, the following statement,

```
fprintf('%010.3f \n', x);
```

is the same as that shown above except that the number will be padded to the left with zeroes to make the field width 10 characters.

See the MATLAB documentation for `fprintf` for more details on format strings.

| Function    | File Type | Operation                                                                 | Open/close file? |
|-------------|-----------|---------------------------------------------------------------------------|------------------|
| load        | MAT       | Read MATLAB variables                                                     | N                |
| save        | MAT       | Write MATLAB variables                                                    | N                |
| readmatrix  | Text      | Read delimiter-separated numeric data from a file into a 1-D or 2-D array | N                |
| writematrix | Text      | Write 1-D/2-D array of numeric data to a delimiter-separated file         | N                |
| fread       | Binary    | Read data of single type from file into 1-D array                         | Y                |
| fscanf      | Text      | Read data of single type from file into 1-D/2-D array                     | Y                |
| textscan    | Text      | Read file containing mixed data types into 1-D/2-D cell array             | Y                |
| fgetl       | Text      | Read line of data into a 1-D array of characters                          | Y                |
| feof        | Text      | Check for end-of-file condition                                           | Y                |
| fwrite      | Binary    | Write array of data of single type to file                                | Y                |
| fprintf     | Text      | Write array(s) of data of mixed data types to file                        | Y                |

Table 6.3: Summary of file input/output functions

## 6.6 Summary

MATLAB provides a range of file input/output functions for interacting with external files. Each function can be useful in different situations, depending on the data type(s) of the data, its format and whether we are dealing with a binary or text file. Table 6.3 summarises the key characteristics of the functions we have discussed in this section.

## 6.7 Further Resources

- MATLAB documentation on text file functions:  
<http://www.mathworks.co.uk/help/matlab/text-files.html>
- MATLAB documentation on general file input/output functions:  
<http://www.mathworks.co.uk/help/matlab/low-level-file-i-o.html>

## 6.8 Exercises

1. Write a MATLAB *m*-file to create the following variables in your MATLAB workspace, and initialise them as indicated:

- a character variable called `x` with the value ‘a’;
- an array variable called `q` containing 4 floating point numbers: 1.23, 2.34, 3.45 and 4.56;
- a Boolean variable called `flag` with the value `true`.

Now save only the array variable to a *MAT* file, clear the workspace, and load the array in again from the *MAT* file.

2. Write the same 1-D array of floating point numbers that you defined in Exercise 1 to a text file using the `writematrix` function. Then clear the workspace and read the array in again using `readmatrix`.
3. Again, working with the array from Exercise 1, again write it to a text file using the `writematrix` function, but this time use the semicolon character as the delimiter. Then clear the workspace and read the array in again using `readmatrix`.
4. The file `patient_data.txt` (available from the KEATS system) contains personal data for a group of patients taking part in a clinical trial. There is a row in the file for each patient, and the five columns represent: patient ID, age, height, weight and heart rate. Write a MATLAB *m*-file that uses the `readmatrix` function to read in this data.
5. Continuing with Exercise 4, write code to input a single integer value `n` from the keyboard, and display the height and weight data for the `n` oldest patients.
6. In the KEATS system you will also be able to access a second patient data file called `patient_data2.txt`. This also contains patient data (patient ID, age, height, weight, heart rate) but for some different patients in the trial. Modify the *m*-file you wrote in Exercise 4 to also read in this data and combine it with the data from the first file. There were five pieces of data for each of nine patients in the first file and the same data for an additional five patients in this second file, so your final data should be a  $14 \times 5$  array. Notice that this second file has no line break characters so you will not be able to use `readmatrix`, because this uses line breaks to separate the rows of data.
7. The equation for computing a person’s body mass index (BMI) is:

$$\text{BMI} = \frac{\text{mass}}{\text{height}^2} \quad (6.1)$$

where the mass is in kilograms and the height is in metres. A BMI of less than 18.5 is classified as underweight, between 18.5 and 25 is normal, more than 25 and less than 30 is overweight, and 30 or over is obese. In the KEATS system you will be provided with two text files: *bmi\_data1.txt* and *bmi\_data2.txt*. These both contain data about patients' height and weight (preceded by an integer patient ID), but in different formats. Write a MATLAB *m*-file to read in both sets of data, combine them, and then report the patient ID and BMI classification of all patients who do not have a normal BMI.

8. When MR scans are performed a log file is typically produced containing supplementary information about the scan's operation. An example of such a file (*logcurrent.log*) is available to you through the KEATS system. One piece of information that this file contains is the date and time at which the scan started. This information is contained in the 2<sup>nd</sup> and 3<sup>rd</sup> words of the line that contains the text "Scan starts". Write a MATLAB *m*-file to display the start date and time for the scan which produced the *logcurrent.log* file.

(Hint: look at the MATLAB documentation for the *strfind* and *strsplit* functions.)

9. The file *names.txt* (available through KEATS) contains the first and last names of a group of patients. Write a MATLAB *m*-file to read in this data using the *textscan* function and display the full names of all patients with the last name "Bloggs".

(Hint: Look at the MATLAB documentation for the *strcmp* command.)

10. The file *bp\_data.txt* (available through KEATS) contains data about patients' systolic blood pressure. There are four fields for each patient: patient ID, first name, last name and systolic blood pressure. Write a MATLAB *m*-file to read in this data, identify any 'at risk' patients (those who have a blood pressure greater than 140) and write only the data for the 'at risk' patients to a new file, called *bp\_data\_at\_risk.txt*.

11. Whenever a patient is scanned in an MR scanner, as well as the log file mentioned in Exercise 8, the image data is saved in files that can then be exported for subsequent processing. One format for this image data, which is used on Philips MR scanners, is the PAR/REC format. With PAR/REC, the image data itself is stored in the REC file whilst the PAR file is a text file that contains scan sequence details which

indicate how the REC file data should be interpreted. An example of a real PAR file (*patient\_scan.par*) is available to you through the KEATS system. Write a MATLAB *m*-file to extract from the PAR file and display the following information: the version of the scanner software (which is included at the end of the 8<sup>th</sup> line), the patient name and the scan resolution.

12. The file *cholesterol.bin* (available through KEATS) is a binary file. The first data element in the file is an 8-bit integer specifying how many records the following data contains. Each subsequent record consists of an 8-bit integer representing a patient's age followed by a 16-bit integer representing the same patient's cholesterol level. Write a MATLAB *m*-file to read the age/cholesterol data into two array variables.
13. Measuring blood sugar levels is an important part of diabetes diagnosis and management. Ten patients have had their post prandial blood sugar level measured and the results were 5.86, 8.71, 4.83, 7.05, 8.25, 7.87, 7.14, 6.83, 6.38 and 5.77 mmol/L. Write a MATLAB *m*-file to write this data to a binary file using an appropriate data type and precision. Your program should then clear the MATLAB workspace and read in the data again.
14. Write a MATLAB *m*-file to read in a text file and write out a new text file that is identical to the input file except that all upper case letters have been converted to lower case.
15. Write a MATLAB *m*-file to read in another *m*-file, strip away any blank lines or comments, and write the remaining statements to a new *m*-file. Note that a comment is any line in which the first non-white space character is a %.  
*(Hint: the MATLAB command `strtrim` can be used to remove leading and trailing white space from a character array.)*

## Notes

# 7 Program Design

*At the end of this section you should be able to:*

- Use structure charts and pseudocode to perform a top-down stepwise refinement design of a complex problem
- Convert a top-down design into working MATLAB code
- Use an incremental testing approach to program development and verification, making appropriate use of test stubs
- Maximise code reuse when designing and developing software

## 7.1 Introduction

So far we have learnt how to make use of a range of MATLAB programming features to develop more and more sophisticated programs. In Section 3 we introduced how to define our own functions for performing specific tasks. However, as we start to address more and more complex problems it is unlikely that they will be easily solved by writing one big function. Rather, an efficient and elegant implementation is likely to consist of a number of functions that interact by making calls to each other and passing data as arguments. *Program design* refers to the process of deciding which functions you need to write and how they should interact.

It is generally accepted that, when tackling larger programming problems, it is not a good idea to start coding straight away: time invested in producing and documenting a high quality structured design will produce significant benefits in the long run, primarily by reducing time spent writing code and maintaining it. This section will introduce some tools that will be useful when designing programs to tackle larger, more complex, problems.

We will also return to the important topic of program testing. Testing, or verifying, that a program meets its requirements is an essential part of the development process. In Section 4 we saw how an incremental approach to writing programs, with testing of the incomplete program after each stage, can expedite the development process. In this section we will discuss how this philosophy is affected by the need to combine it with a structured design process.

In computer science, it is common to identify two distinct approaches to program design: *top-down* design and *bottom-up* design. In this section we

will also discuss the meanings of these two concepts. However, in procedural programming (see Section 1.1.1), which is the subject of this module, top-down design has been the traditional approach, so we will focus mainly on this, illustrating it with a simple example.

## 7.2 Top Down Design

With a top-down design approach, the emphasis is on planning and having a complete understanding of the system before writing code. Top-down design is also often referred to as *stepwise refinement*. The basic idea is to take a hierarchical approach to identifying program modules. We start off by breaking down a complex problem into a number of simpler sub-problems. Each sub-problem will be addressed by a separate program module, which will have its requirements separately specified, including how it should interact with the main program (i.e. the top level of the hierarchy). The interaction is typically specified in terms of what data is passed to the module from the main program, and what data is passed back from the module to the main program. Next, each identified module is broken down further into sub-modules, which will also have their behaviour specified, including what data is passed between the new sub-modules and their parent modules. This process continues until the sub-modules are sufficiently simple as to be implemented relatively easily.

The result of this *stepwise refinement* is a hierarchy of modules and sub-modules that represents the structure of the program to be implemented. This, together with the specification for each module, represents the program design. Normally, no coding will begin until a sufficient level of detail has been reached in the design of at least some part of the system. Often, the complete structured design will be produced before any coding takes place.

We will illustrate this process with an example.<sup>11</sup>

**Example 7.1.** We are required to design a program to report if a given integer read from the keyboard is a *perfect number*. An integer is a *perfect number* if it is equal to the sum of its integer divisors (e.g.  $6=3+2+1$ ,  $28=14+7+4+2+1$ ).

---

<sup>11</sup> Adapted from: <http://courses.cs.vt.edu/cs1044/summerI2006/Notes/C02.ProgramDevelopment.pdf>

We start off by breaking the problem down into simpler sub-problems. In software design, this is known as *factoring*, and the first time you apply it to the overall problem it is known as *first level factoring*. We will illustrate our first level factoring using a commonly used graphical technique known as *structure charts*.

### Step 1 - First Level Factoring

Figure 7.1a shows the structure chart after first level factoring has been performed. The main program (which we have called *Perfect*) is at the top level of the hierarchy. Below this, we have broken down our initial problem into three sub-problems. We are saying that the requirements of our program can be met by executing three modules one after the other: first we get the number to be tested (*Get Number*), then we determine if it is perfect (*Is Perfect?*), and finally we display the answer (*Display Answer*). The arrows linking these three modules to the parent module are annotated with the data that needs to be passed between them. Empty circles with an arrow indicate data flowing in the direction of the arrow. For example, the *Get Number* module should pass back the number that was read,  $N$ , to the *Perfect* program. This number is then passed to the *Is Perfect?* module for testing, which returns a Yes/No result. This result is passed to the *Display Answer* module.

Currently, none of these three new modules is simple enough to be implemented easily, so we proceed by breaking down one of them (*Is Perfect?*) still further. This is known as *further factoring*.

### Step 2 - Further Factoring

Figure 7.1b shows the additional structure chart for the further factoring of *Is Perfect?*. Now, we are saying that the problem of determining if a number is perfect or not can be solved by first determining and summing its divisors, and then checking to see if this sum is equal to the original number. Note the data flowing between the parent module (*Is Perfect?*) and its two sub-modules.

### Step 3 - Further Factoring

Now we proceed with further factoring of the *Get Number* module identified in **Step 1**. Figure 7.1c shows the structure chart that represents the operation of this module. This specifies how the problem of reading a positive number from the user can be solved: first, an appropriate prompt is displayed, then a number is inputted, and finally the program stops if the number entered was negative. This time there is some new notation in the structure chart. In addition to the empty circles with arrows representing data, we also have a filled circle. A filled circle with an arrow in a structure chart indicates *control* information. Control information is normally a binary value that has some effect on future program execution. For example, in Figure 7.1c the control information called *Flag* is passed back from the *Check if -ve* sub-module: this will be true if *N* is negative and false otherwise. This value is then used to decide whether or not to execute the next sub-module, *Stop*. The *conditional execution* of the *Stop* sub-module is indicated by the diamond symbol at the top of the line joining *Get Number* and *Stop*.

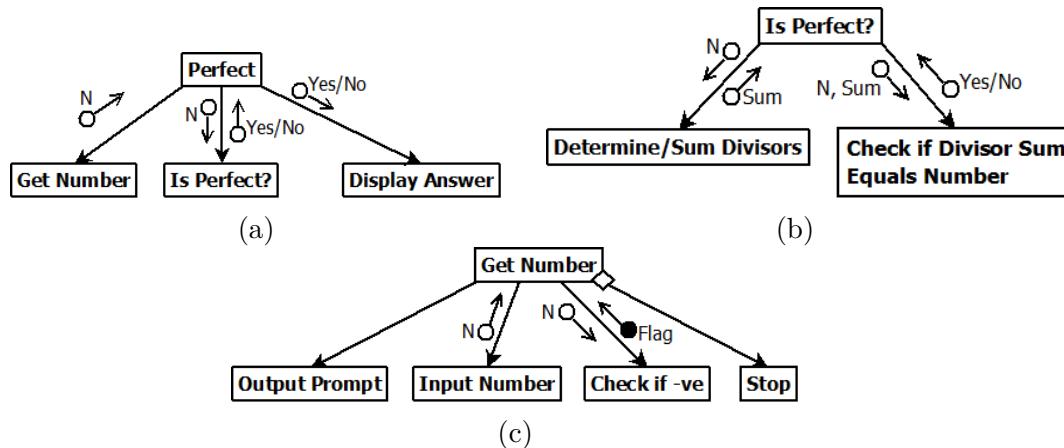


Figure 7.1: Structure charts representing the design of the perfect number program: (a) **Step 1** - first level factoring; (b) **Step 2** - further factoring of *Is Perfect?*; (c) **Step 3** - further factoring of *Get Number*. Empty circles represent data, filled circles represent control. The diamond in (c) represents conditional execution.

As an aside, as well as conditional execution (which corresponds to conditional statements in procedural languages such as MATLAB - see Section 2), structure charts can also represent *iteration*. For instance, suppose that we

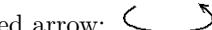
| Symbol                                                                                                     | Meaning                                                                                                 |
|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Box:                      | Module performing task x - see Figure 7.1                                                               |
| Arrow joining boxes:      | One module calls another - see Figure 7.1                                                               |
| Annotated empty circle:   | Data flow, i.e. data x flows between modules in direction of arrow - see Figure 7.1                     |
| Annotated filled circle:  | Control flow, i.e. control information x flows between modules in direction of arrow - see Figure 7.1c  |
| Diamond:                  | Conditional execution, i.e. execution of called module depends on control information - see Figure 7.1c |
| Curved arrow:             | Iteration, i.e. called modules covered by arrow are executed multiple times - see Figure 7.2            |

Table 7.1: Summary of structure chart notation

wanted our program to test if multiple numbers were perfect or not, i.e. the *Get Number*, *Is Perfect?* and *Display Answer* modules together should be iteratively executed. Figure 7.2 shows how the structure chart shown in Figure 7.1a should be modified to indicate this iteration. The curved arrow covering the three modules indicates that they should be executed multiple times.

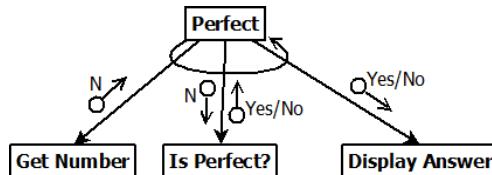


Figure 7.2: Structure chart showing iteration of sub-modules.

Table 7.1 provides a summary of the notation we have introduced for structure charts.

#### Step 4 - Write Pseudocode

To return to our design example, we now proceed by examining which modules might need further elaboration. Looking at Figure 7.1c we might conclude that this structure chart has enough detail already to proceed to implementation. However, there are three other modules that do require further analysis: *Display Answer* from the first-level factoring, and the two new sub-modules from Step 2 (*Determine/Sum Divisors* and *Check if Divisor Sum*

*Equals Number*). Although it would be possible to continue to use structure charts to further break down these problems, we will now introduce a new way of representing program structure: *pseudocode*. Pseudocode is simply an informal high-level description of the steps involved in executing a computer program, often written in something similar to plain English. Pseudocode is commonly used when we get down to the lower-level details of how modules work, rather than the interactions of higher-level operations commonly represented by structure charts.

For example, the pseudocode below shows more detail on how the *Determine/Sum Divisors* module works (which was identified in the structure chart shown in Figure 7.1b).

*PSEUDOCODE - Determine/Sum Divisors:*

```
Set Sum to 1
Set D to 2
While D is less than or equal to N/2
    If D is a divisor of N
        Add D to Sum
    Add 1 to D
```

This pseudocode indicates the steps involved in determining the divisors of a positive integer and summing them. We first initialise the divisor sum (denoted by *Sum*) to 1, because 1 is a divisor of all positive integers. The current integer to be tested (denoted by *D*) is then initialised to 2. A loop tests all integers up to *D*/2, since no divisor can be larger than this. For each integer tested, if it is a divisor it is added to *Sum*. Then the current integer *D* is incremented to move on to the next number to be tested. Note that this pseudocode is starting to look more like computer code, with even variables and simple operations on them being defined. However, we haven't yet written any code, so our design could be implemented in MATLAB or another language.

Pseudocode for the remaining two modules of our design is shown below.

*PSEUDOCODE - Check if Divisor Sum Equals Number:*

```
If Sum equals N
```

```
Set answer to true
Else
    Set answer to false
```

*PSEUDOCODE - Display Answer:*

```
If answer is true
    Output: "N is perfect"
Else
    Output: "N isn't perfect"
```

A typical top-down design process would involve first level factoring (e.g. Figure 7.1a), followed by further factoring (e.g. Figures 7.1b-c) as needed until a certain level of detail is reached. Then some of the bottom level modules in the structure charts would be expanded by writing pseudocode. The decision about when to switch from structure charts to pseudocode is to a certain extent subjective and a matter of personal preference: you may prefer to use entirely structure charts, or entirely pseudocode, but it is common to use a mixture of the two as we have in this example. Either way, once the design is complete and documented, coding would begin.

### 7.2.1 Incremental Development and Test Stubs

We now have a design that can be used as the starting point for producing an implementation. Regardless of the programming language used, we need a general approach for producing our implementation. In Section 4.2 we introduced the concept of *incremental development*, in which software is developed in small pieces, with frequent testing of the developing program. In Example 4.1 a program for interest calculation was developed incrementally, and the entire program was tested after each new piece of code was added. However, this was for quite a simple example in which each part of the overall program consisted of just a few lines of code. Now that we are tackling larger, more complex problems, it is likely that our operations will be functions that perform quite sophisticated tasks. So this begs the question: how can we perform incremental development and testing when implementing a top-down design of a larger program?

The answer to this question lies in the concept of a *test stub*. A test stub is a

simple function or line of code that acts as a temporary replacement for more complex, yet-to-be-implemented code. For instance, in Example 7.1 we might use test stubs for the *Get Number* and *Is Perfect?* modules whilst we are developing and testing the code for the *Display Answer* module. The test stub for *Get Number* could, for example, always set the number to the same value. The test stub for *Is Perfect?* could always return a true value, regardless of the number being tested. The code listings shown below illustrate such a program under development.

*perfect.m:*

```
% script to read integer and check if it is perfect
% (i.e. equal to sum of its divisors)

% Get Number (test stub)
n=10;

% Is Perfect? (test stub)
yes_no = is_perfect(n);

% Display Answer
...
```

*is\_perfect.m:*

```
function answer = is_perfect(num)
% Usage:
%   answer = is_perfect (num)
%   num      : Number to be tested
%   answer   : Output, Boolean value indicating whether num
%              is perfect

answer = true;

end % function
```

Once the *Display Answer* module has been developed and tested, the test stub for *Get Number* could be replaced with an implementation and tested. Finally, the test stub for *Is Perfect?* (i.e. the `is_perfect` function) could be replaced with its implementation. Final testing would then verify the operation of the complete program.

### 7.3 Bottom-Up Design

Although top-down design is the most common approach when developing procedural programs, an alternative approach does exist: *bottom-up design*. The philosophy of bottom-up design can be best understood by drawing an analogy with children’s building blocks. When building a structure from building blocks we would probably have a general idea of what we wanted to make to begin with (e.g. a house), but it is common to start off by building some small components of the overall structure (e.g. the walls). Once enough components are complete, they will be fitted together. It’s the same with bottom-up software design: we would begin with a general idea of the system to be developed (but not specified in detail), and then start off by developing some small self-contained modules. Once enough modules are complete, they would be combined to form larger subsystems. After each combination, the developing program would be tested. Like the top-down approach discussed in Section 7.2.1, this can be thought of as a form of incremental development (see Section 4.2): the program is developed module-by-module, with some increments consisting of combining existing modules to form larger modules. This process is repeated until the complete system has been produced.

A key feature of development using bottom-up design is early coding and testing. Whereas, with top-down design, no coding is performed until a certain level of detail is reached in the design, with bottom-up design coding starts quite early. Therefore, there is a risk that modules may be coded without having a clear idea of how they link to other parts of the system, and that such linking may not be as easy as first thought. On the other hand, a bottom-up approach is well-suited to benefitting from *code reuse*. Code reuse refers to making use of previously developed and tested, modular pieces of code. It is generally considered to be a good thing in software development, since it reduces the amount of duplicated effort in writing the same or similar code modules, and also reduces the chances of faults being introduced into programs (since the reused modules have been extensively tested already).

To further illustrate the concept of bottom-up design, let’s return to the perfect numbers program we introduced in Example 7.1. To develop this program using bottom-up design, we would first consider what types of function/module we might need to write in order to develop a program to report if an integer is perfect or not. An initial obvious choice might be a module to

determine and sum the divisors of an integer. So we would start off by specifying the requirements for this module, designing it using structure charts and/or pseudocode, and then writing the code and testing it. Next, we might choose to expand this module to make a module to test if a number is perfect or not. This would involve specifying/designing/coding/testing a module to check if the sum of the divisors is equal to the original number. This new module would then be combined with the existing module to make a perfect number testing module. Next, we would proceed to specify/design/code/test other modules, such as getting the input number and displaying the result. Once all modules were completed and tested, they would be combined to form the final program.

## 7.4 A Combined Approach

Although top-down and bottom-up design represent opposing philosophies when it comes to software design, most modern software design techniques combine both approaches. Although a thorough understanding of the complete system is usually considered necessary to produce a high quality design, leading theoretically to a more top-down approach, most software projects attempt to maximise code reuse. This reuse of existing code modules gives designs more of a bottom-up flavour. You may find it useful to adopt such an approach when designing your own software, i.e. take a predominantly top-down approach to the software design process (e.g. using structure charts and pseudocode), but always keep in mind the possibilities for code reuse. Where possible, try to include modules in your design that can be reused from other programs you have written in the past, or from freely available code on the internet.

## 7.5 Summary

There are two opposing philosophies to approaching the software design process: *top-down* design and *bottom-up* design.

Top-down design is also known as *stepwise refinement* and involves successively breaking down problems into simpler sub-problems, stopping when the sub-problems are simple enough to tackle on their own. The process of identifying sub-problems and the interactions between them is referred to as *factoring*. Top-down designs can be documented using a graphical technique

such as *structure charts*, and/or text-based techniques such as *pseudocode*. Top-down designs can be incrementally developed by making use of *test stubs*, which are simple functions or lines of code that act as a temporary replacement for the full, yet to be implemented code.

Bottom-up design involves starting off with a general idea of the system to be developed, but not documenting it in detail. Rather, small, self-contained modules that are likely to be useful are identified, their requirements specified, a design produced and code written and tested. These low-level modules are successively combined to form higher-level modules. This process continues until the final program has been formed and tested. With bottom-up designs there is a risk that modules may be coded without having a clear idea of how they might link to other parts of the system. On the other hand, bottom-up approaches are well-suited to the concept of *code reuse*, (making use of previously written and tested code modules).

Modern software design approaches generally feature a combination of top-down and bottom-up design: a predominantly top-down approach is taken but care is taken to identify modules that can be reused from previous implementations. As a general rule, for larger programs, it's a good idea not to begin coding straight away. It's almost always a good idea to plan your design on paper first using tools such as structure charts and pseudocode.

## 7.6 Further Resources

- Top-down vs. bottom-up design: [http://en.wikipedia.org/wiki/Top-down\\_and\\_bottom-up\\_design](http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design)
- Test stubs: [http://en.wikipedia.org/wiki/Test\\_stub](http://en.wikipedia.org/wiki/Test_stub)
- Structure charts: [http://en.wikipedia.org/wiki/Structure\\_chart](http://en.wikipedia.org/wiki/Structure_chart),  
<http://www.freetutes.com/systemanalysis/sa6-structure-charts.html>,  
[http://en.wikibooks.org/wiki/A-level\\_Computing/AQA/Print\\_version/Unit\\_1#Structure\\_charts](http://en.wikibooks.org/wiki/A-level_Computing/AQA/Print_version/Unit_1#Structure_charts)

## 7.7 Exercises

1. Write a MATLAB implementation of the perfect number top-down design given in Example 7.1. You will need to consider whether to implement each module using a sequence of statements in the main

script *m*-file or as a separate MATLAB function. Use an incremental development approach and test stubs when developing your solution.

2. Modify your solution to Exercise 1 so that your program reads multiple numbers from the user, stopping only when they enter a negative number. (This corresponds to the structure chart shown in Figure 7.2.) When a negative number is entered the program should exit with no error message.
3. Modify your solution so that, if the number entered is not perfect, the program will report the nearest perfect number to the number entered. If no perfect number is found within a predefined range of the number (say, 100 either side) then a suitable message should be displayed. You may need to make changes to the structure charts, pseudocode and implementation.
4. A prime number is one which has no divisors apart from 1 and itself. Therefore, much of the processing required to determine if a number is perfect or prime is the same. Modify your solution so that, for the number being tested, it also reports whether or not it is prime. Again, you should make any necessary changes to the structure charts, pseudocode and implementation. (For the purpose of this exercise, try not to use the built-in MATLAB function `isprime`.)
5. The file *patient\_data.txt* (available from the KEATS system) contains data for a group of patients registered at a GP surgery (patient ID, name, date of birth). A second file (*patient\_examinations.txt*, also available through KEATS) contains data about physical examinations that patients at the surgery have undergone (patient ID, age, height, weight, systolic/diastolic blood pressure, cholesterol level). There may be multiple examinations for each patient, or none.  
The GP has decided to implement a programme in which ‘high risk’ patients are invited for further tests. The initial definition for ‘high risk’ patients is those who have an age that was greater than or equal to 75 at their most recent examination.

Design and implement a program to read in the data from these files and display to the screen the patient IDs and names of all high risk patients. You should use a top-down design approach, utilising structure charts and pseudocode to design your program before starting to write code.

Use test stubs and incremental development whilst writing and testing your code.

6. The GP now wishes to expand her definition of a ‘high risk’ patient by also including those patients who have a body mass index (BMI) of less than 18.5 or greater than 25 at their most recent examination. Recall that a patient’s BMI can be calculated using the formula

$$\text{BMI} = \text{mass}/\text{height}^2.$$

where the height is specified in metres and the weight in kilograms. Modify your program design and implementation from Exercise 5 to meet this new requirement.

7. Modify your program design and implementation from Exercise 6 so that the program also reports the reason for a patient being high risk, i.e. either because of their age or their BMI or both.
8. Modify your program design and implementation from Exercise 7 so that the program also reports a patient as high risk if their cholesterol level is greater than 200mg/dL. Again, the reason for being reported as high risk should be displayed.
9. Modify your program design and implementation from Exercise 8 so that the program will also consider the patients’ systolic and diastolic blood pressures. The rules should be:
  - If the systolic blood pressure is  $\geq 120\text{mmHg}$  but  $< 140\text{mmHg}$ , or the diastolic blood pressure is  $\geq 80\text{mmHg}$  but  $< 90\text{mmHg}$ , the patient is classified as having *pre-high blood pressure*. These patients should only be high risk if another one of the risk criteria is met. The pre-high blood pressure should also be reported as a reason for being at risk.
  - If the systolic blood pressure is  $\geq 140\text{mmHg}$ , or the diastolic blood pressure is  $\geq 90\text{mmHg}$ , the patient is classified as having *high blood pressure*. These patients should be reported as high risk even if no other criteria are met.

## **Notes**

# 8 Visualisation

*At the end of this section you should be able to:*

- Use MATLAB to visualise multiple plots
- Use MATLAB to produce 3-D plots
- Use MATLAB to read/write/display/manipulate imaging data
- Access handles to graphical objects using MATLAB

## 8.1 Introduction

MATLAB is a powerful application for numerical processing and visualisation of complex datasets. So far in this module we have seen how to create data using variables (Section 1) and programming constructs (Sections 2 and 3), and also to read data from external files (Section 6). In Section 1 we also introduced the basics of data visualisation in MATLAB. In this section we will build on this knowledge and introduce some built-in MATLAB functions for performing more sophisticated visualisations of complex datasets, including images.

## 8.2 Visualisation

In Section 1 we saw how to use the MATLAB `plot` function to produce line plots. We also introduced a number of MATLAB commands to annotate plots produced in this way (e.g. `title`, `xlabel`, `ylabel`, `legend`, etc.). Most of these annotation commands are applicable to other types of plot, as we will see later in this Section. However, `plot` is only suitable for relatively simple data visualisations, such as visualising the relationship between two variables or plotting a 1-D signal. In this Section we will introduce some more built-in MATLAB functions that allow us to have more flexibility in producing data visualisations.

### 8.2.1 Visualising Multiple Datasets

Often it is useful to be able to view multiple visualisations at once, for example to enable us to compare the variations of different variables over time. There are a number of ways in which we can do this using MATLAB.

**Example 8.1.** The easiest thing to do is to simply display multiple plots,

either separately or using the same figure. For example, in Section 1.7 we saw how to display multiple plots on the same figure using the `plot` function:

```
x = 0:0.1:2*pi;
y = sin(x);
y2 = cos(x);
plot(x,y, '-b', x,y2, '--r');
legend('Sine', 'Cosine');
```

The `plot` function can take multiple sets of three arguments (i.e.  $x$  data,  $y$  data, line/marker style), each of which represents a different line plot in the figure.

**Example 8.2.** Alternatively, if we have a large number of plots to display, perhaps within a loop in our program, it may be more convenient to use multiple `plot` commands to produce our figure. This is made possible using the MATLAB `hold` command, as the following example illustrates.

```
x = 0:0.1:2*pi;
y = sin(x);
y2 = cos(x);
plot(x,y, '-b');
hold on;
plot(x,y2, '--r');
title('Sine and cosine curves');
legend('Sine', 'Cosine');
```

The command `hold on` tells MATLAB to display all subsequent plots on the current figure without overwriting its existing contents (the default behaviour is to clear the current figure before displaying the new plot). This behaviour can be turned off using the command `hold off`.

**Example 8.3.** Sometimes, displaying too many plots on the same figure can make visualisation more difficult, so it may be preferable to use multiple figures. We can achieve this in MATLAB through the use of the `figure` command, e.g.

```
x = 0:0.1:2*pi;
y = sin(x);
```

```

y2 = cos(x);
plot(x,y);
title('Sine curve');
figure;
plot(x,y2);
title('Cosine curve');

```

Here, the `figure` command tells MATLAB to create a new figure window (keeping any existing figure windows intact) and to make it the current figure. Any subsequent plots will be displayed in this new current figure.

**Example 8.4.** We can collect the output from the `figure` command in order to control which figure to plot in or modify when we have more than one available. When we first call the `figure` function, without any arguments, it returns a *handle* to the figure it generates. We can access a figure by referring to its handle later in the code. We can do this by passing the handle as an argument to the `figure` function. Doing this causes the figure with that handle to become the focus of any subsequent plotting related commands.

Here is an example which is based on the previous one:

```

x = 0:0.1:2*pi;
y1 = sin(x);
y2 = cos(x);

% Calling figure with no arguments.
% Creates an empty figure window, returns a handle each time.
h1 = figure();
h2 = figure();

% Calling figure with a handle as the argument.
figure(h1)
plot(x,y1);

% Using a different handle to plot to another window.
figure(h2)
plot(x,y2);

% Switch back to first figure to add a title.
figure(h1)
title('Sine curve');

```

```
% Switch again to add title to second figure.
figure(h2)
title('Cosine curve');
```

We return to the concept of handles in Example 8.6 below.

**Example 8.5.** Creating multiple figures as in the previous examples can be useful, but generating too many figure windows can make it difficult to keep track of them. An alternative way of generating multiple separate plots that overcomes this weakness is the use of subplots. The MATLAB subplot command allows us to display multiple plots within the same figure, as this example illustrates.

```
x = 0:0.1:2*pi;
y = sin(x);
y2 = cos(x);
subplot(2,1,1);
plot(x,y);
title('Sine curve');
subplot(2,1,2);
plot(x,y2);
title('Cosine curve');
```

This code produces the output shown in Figure 8.1. From the code shown above, we can see that subplot takes three arguments. The first two are the number of rows and columns in a rectangular grid of subplots that will be created (in this case, 2 rows and 1 column). The third argument is the number of the current subplot within this grid, i.e. where subsequent plots will be displayed. MATLAB numbers its subplots using the ‘row major’ convention: the first subplot is at row 1, column 1, the second subplot is at row 1, column 2, and so on.

**Example 8.6.** Finally, there is occasionally a requirement to display multiple plots on the same figure which have different scales and/or units on their  $y$ -axes. The following example illustrates this. This is a program that visualises two different exponentially decaying oscillating functions. The output of the program is shown in Figure 8.2.

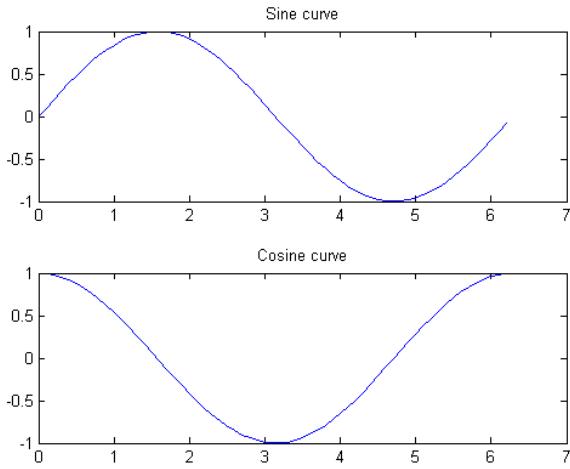


Figure 8.1: Example of the use of the `subplot` command to generate multiple plots in a single figure.

```
% generate data
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);

% produce plots
figure;
yyaxis left;
plot(x, y1)
ylabel('Low frequency oscillation');
yyaxis right;
plot(x, y2)
ylabel('High frequency oscillation');

% annotate plots
title('Exponentially decaying oscillations');
xlabel('Time');
```

The built-in MATLAB function `yyaxis` is used to display a single plot with two different  $y$ -axes. The scales of the  $y$ -axes are determined automatically by the ranges of the two datasets ( $y_1$  and  $y_2$ ). The two datasets share a common  $x$ -axis. Refer to the MATLAB documentation for more details on these and other commands we have covered.

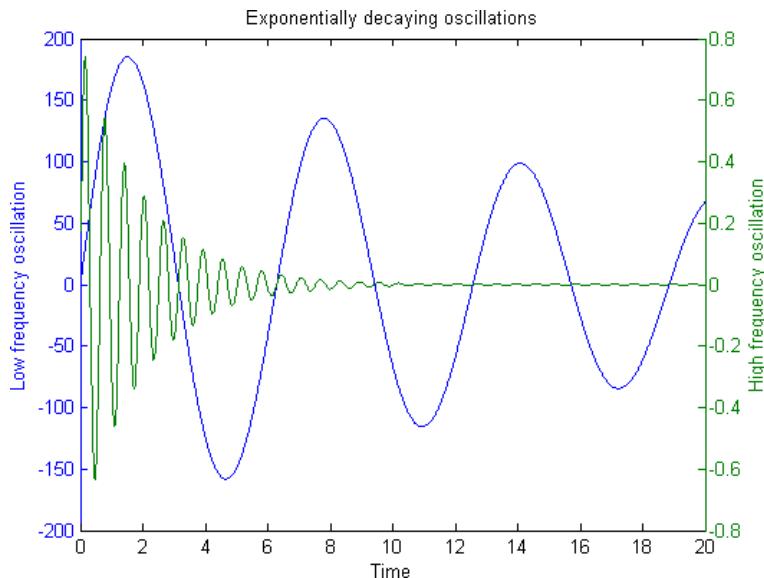


Figure 8.2: Example of the use of the `yyaxis` command to generate multiple plots in a single figure with different  $y$ -axes.

### 8.2.2 3-D Plotting

Often in biomedical engineering it is useful to be able to visualise and analyse data which have several different values per ‘individual’ datum. For example, we might want to visualise points in 3-D space: in this case the individual data are the points, and each point has three associated values (its  $x$ ,  $y$  and  $z$  coordinates). Alternatively, we might have a range of measurements related to hospital patients’ health (e.g. blood pressure, cholesterol level, etc.): in this case the individuals are the patients and there will be multiple measurement values for each patient. This type of data is called *multivariate* data. You will learn more about multivariate and other types of data in the Computational Statistics module later in the BEng programme, but for now we will introduce some built-in MATLAB functions for visualising multivariate data.

**Example 8.7.** The first command we will discuss is the `plot3` function. The following example (adapted from the MATLAB documentation) illustrates the use of `plot3`. This program will display a helix, and its output is shown in Figure 8.3.

```

% the z-coordinate of the helix
t = 0:0.05:10*pi; % 5 times around a circle

% the x-coordinate of the helix
st = sin(t);

% the y-coordinate of the helix
ct = cos(t);

% 3-D plot
figure;
plot3(st,ct,t, '.b')
title('Helix');
xlabel('X');
ylabel('Y');
zlabel('Z');

```

The first three (non-comment) program statements set up the array variables for storing the  $x$ ,  $y$  and  $z$  coordinates of the helix. These three arrays will be of the same length because each 3-D point has 3 coordinate values. The `plot3` function takes 4 arguments: the first three are these data arrays, and the fourth (optional) argument specifies the line/marker appearance, i.e. the same as for the `plot` function (see Section 1.7). Note that the same annotation commands (`title`, `xlabel`, etc.) can be used with figures produced by `plot3`, but that now we have an extra axis to annotate using `zlabel`.

As well as `plot3` for visualising multivariate data, MATLAB also provides two built-in functions for visualising data as *surfaces*: `mesh` and `surf`. An example of such a visualisation is shown in Figure 8.4a, which is the output of the program shown in Example 8.8 below. This type of plot is commonly used for visualising data in which there is a single  $z$  value for each point on an  $x$ - $y$  plane. (Note that this isn't true for the helix data we saw in Example 8.7 because there are multiple  $z$  values for a given  $x$ / $y$  value combination.)

**Example 8.8.** In this example (adapted from the MATLAB documentation) a mesh plot is produced to illustrate the *sinc* function (i.e.  $\sin(x)/x$ ).

```

% create arrays representing a grid of x/y values
[X,Y] = meshgrid(-8:.5:8, -8:.5:8);

```

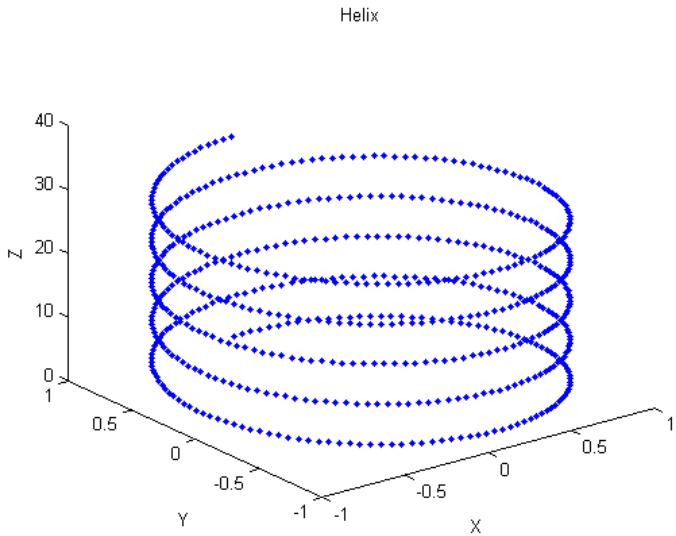


Figure 8.3: Output of the helix program shown in Example 8.7.

```
% define sinc function
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R) ./ R;

% mesh plot
mesh(X,Y,Z)
%surf(X,Y,Z)
title('Sinc function');
xlabel('X');
ylabel('Y');
zlabel('Z');
```

The built-in MATLAB `meshgrid` command creates two 2-D arrays of values. These define the  $x$ - $y$  grid over which the mesh is plotted. One array indicates the  $x$ -coordinate at each grid point and the other indicates the  $y$ -coordinate at the same grid point. Arrays of this form, together with a corresponding array of  $z$  values, are the arguments required by the `mesh` command.

The  $z$ -coordinate array is defined by computing the radial distance from the origin of each point, and using it as the input to the *sinc* function. Note the use of the `.^` operator to square the  $x$  and  $y$  coordinates: this is necessary to ensure that the `.^` operator is applied to each element of the array *individually*.

rather than performing a *matrix power* operation (see Section 1.4). The `eps` command in MATLAB returns a very small floating point number and is necessary here to avoid a division-by-zero error at the origin of the  $x$ - $y$  plane.

The `surf` command has the same form as `mesh`, as we can see from the commented-out command in the code above. The result of using `surf` is to produce a filled surface rather than a wireframe mesh (see Figure 8.4b).

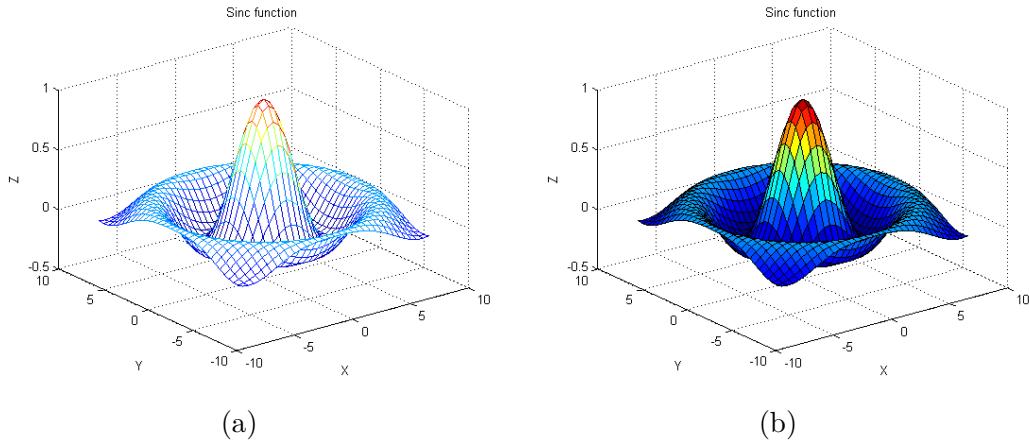


Figure 8.4: Output of the sinc function program shown in Example 8.8: (a) using `mesh` command; (b) using `surf` command.

### 8.2.3 The `meshgrid` Command

The `meshgrid` command is useful for 3D plotting of gridded data. This section describes how it works. The MATLAB function `meshgrid` can be used to generate grid-shaped arrays containing the coordinate components of all points on a regular grid. The syntax for making a 2-D grid is

```
[X, Y] = meshgrid(x, y)
```

The input arguments are two 1-D vectors containing the components required along each dimension, one for  $x$  and one for  $y$ . In this case, the output arguments, `X` and `Y`, are 2-D arrays with the same shape as the grid, containing *all* the grid points'  $x$  and  $y$  components. Each one is generated by replicating the input 1-D vectors, horizontally or vertically. An example call to make a small grid is as follows:

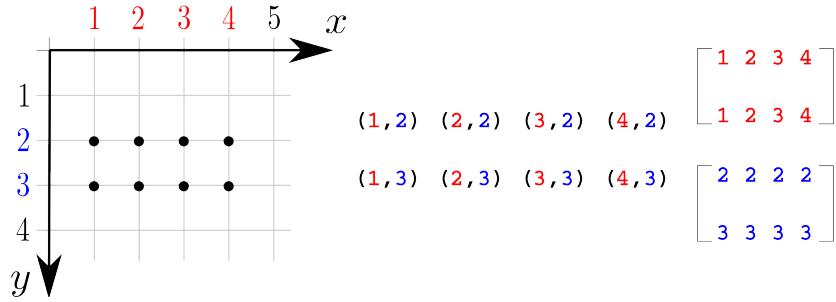


Figure 8.5: An illustration of the use of the `meshgrid` command. The grid required is shown by the black dots on the left-hand axes (with  $y$  pointing down). The grid ranges from  $x = 1$  to  $x = 4$  and  $y = 2$  to  $y = 3$ . The coordinates for all required grid points are shown in the middle. The output from `[X, Y] = meshgrid(1:4, 2:3)` are two separate arrays shown on the right, one containing the grid  $x$  coordinates and one containing the grid  $y$  coordinates.

```
>> [X, Y] = meshgrid(1:4, 2:3)
```

```
X =
    1     2     3     4
    1     2     3     4
```

```
Y =
    2     2     2     2
    3     3     3     3
```

Note that the  $y$  coordinates increase as we read *down* the array because MATLAB arrays are indexed starting from the top left. This is equivalent to viewing the  $x - y$  axes as shown in Figure 8.5 which shows how the matrices `X` and `Y` in the above call are generated.

#### 8.2.4 Imaging Data

Images are among the most common types of data you will encounter in biomedical engineering. Medical images can be acquired using modalities such as magnetic resonance imaging (MRI), X-ray computed tomography

(CT) and ultrasound (US). You will learn more about the physics of acquisition and the uses of these imaging modalities throughout the BEng programme. In this section we will introduce how to read, write and display imaging data in MATLAB.

**Example 8.9.** The following example illustrates the use of several built-in image-related MATLAB commands.

```
% read image
im = imread('fluoro.tif');

% display image
imshow(im);

% modify image
im(100:120,100:120) = 0;

% save image (saves 'im' array)
imwrite(im, 'fluoro_imwrite.tif');

% save image (saves figure as displayed)
saveas(gcf, 'fluoro_saveas.tif');
```

The `imread` command reads imaging data from an external file. The single argument is the file name, and the value returned is a 2-D array of pixel intensities.

Images are treated as normal 2-D arrays in MATLAB. The intensities can be of different types such as `int8`, `single` or `double`, but within a single image all must have the same type, as with all MATLAB arrays. Imaging data can be manipulated just like any other MATLAB array, as we can see from the line in the above example in which a rectangular block of pixels has its intensities set to 0.

The `imshow` command can be used to display a 2-D image in a figure window. The single argument is the 2-D array of pixel intensities.

This example shows two ways in which the image can be written to an external file. The `imwrite` command takes a 2-D array as its only argument, so the image data written to the file depends only on this array and not on any figure being displayed. The `saveas` command is used in this example to

save the data in the current figure (`gcf` means *get current figure*). Therefore, any changes made to the displayed figure will also be saved, but any changes made to the original array since it was displayed (as in the example above) will not be saved. Here, `saveas` saves the figure as an image, but it can also be used to save in the native MATLAB figure format, which has a “fig” file extension.

Both the `imread` and `imwrite` commands can handle a range of different imaging formats, such as `bmp`, `png` and `tif`. For a full list of supported formats type `imformats` at the MATLAB command window. Normally, just adding the extension to the file name is enough to tell MATLAB which format you want to use.

In medical imaging it is common for images to be 3-D datasets rather than 2-D (e.g. many MRI and CT images and some US images). Unfortunately, MATLAB doesn’t have any built-in functions for visualising 3-D images. However, several have been written by the MATLAB user community and been made available for free download from the Mathworks File Exchange web site. For example, the `imshow3D` function allows simple interactive slice-by-slice visualisation of 3-D image datasets. URLs are provided in the Further Resources at the end of this section.

### 8.3 Summary

MATLAB provides a number of built-in functions for performing more sophisticated data visualisations. The `plot`, `hold`, `figure`, `subplot` and `yyaxis` commands can be used to visualise multiple datasets at the same time. 3-D plots can be produced using the `plot3`, `mesh` and `surf` commands. Imaging data can be read in to 2-D array variables using `imread` and displayed using `imshow`. The array can be saved using `imwrite`, whilst the displayed figure can be saved using `saveas`. A range of 3-D image visualisation tools are available for free download from the Mathworks File Exchange web site.

### 8.4 Further Resources

- MATLAB documentation on 2-D and 3-D plots:  
<http://www.mathworks.co.uk/help/matlab/2-and-3d-plots.html>

- Mathworks File Exchange:  
<http://www.mathworks.co.uk/matlabcentral/fileexchange>
- `imshow3D` function: <http://www.mathworks.co.uk/matlabcentral/fileexchange/41334-imshow3d-3d-imshow>

In addition to these resources, you will learn more about data visualisation in the Computational Statistics module later in the BEng programme, so you are referred to the laboratory manual for that module for more information.

## 8.5 Exercises

1. The file `patient_data.txt` (available through the KEATS system) contains four pieces of data for multiple patients: whether they are a smoker or not ('Y'/'N'), their age, their resting heart rate and their systolic blood pressure. Write a program to read in these data, and produce separate arrays of age and heart rate data for smokers and non-smokers.

Using these data, produce plots of age against heart rate for smokers and for non-smokers,

- (a) on the same figure with a single use of the `plot` command;  
 (b) on multiple figures with the `figure` command;  
 (c) on the same figure with the `hold` command;  
 (d) on subplots with the `subplot` command.
2. A research team wish to evaluate a new algorithm for aligning 3-D medical images (this process of image alignment is known as *image registration*). To evaluate their algorithm the team have performed multiple experiments involving registering image pairs, and computed two measures of registration success for each experiment. The first is the value of the *normalised cross correlation* (a measure of image similarity) between the image pairs. The second is a *landmark error*, which was determined by observers manually clicking on corresponding anatomical landmarks in the image pairs and computing the distance between them. The values of these measures for 10 separate experiments are provided in the files `ncc.txt` and `points.txt` respectively

(available through the KEATS system). Both files contain 10 rows and two columns: the columns represent the experiment number and the respective evaluation measure.

Write a MATLAB program to read in these data and produce a single plot showing the values of both evaluation measures (on the  $y$ -axis) against the experiment number (on the  $x$ -axis). Annotate the  $x$  and  $y$  axes of your plot and add a suitable title. Your final figure should look like the one shown in Figure 8.6.

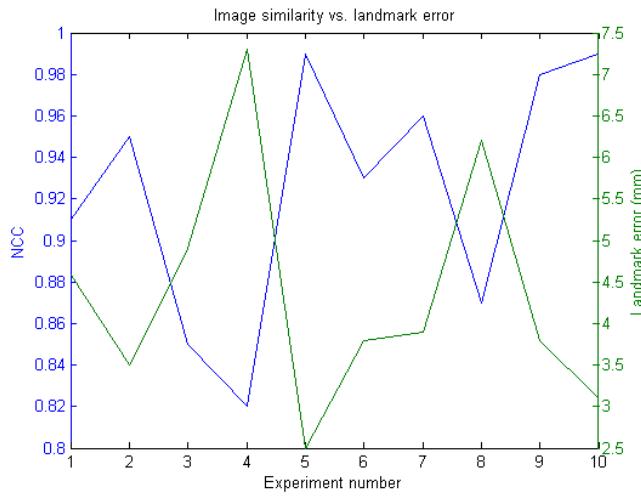


Figure 8.6: Plot of normalised cross-correlation and landmark error against experiment number.

3. Using the same *patient\_data.txt* data you used in Exercise 1, write a MATLAB program to produce a 3-D plot of the patient data. The three coordinates of the plot should be the age, heart rate and blood pressure of the patients, and the smokers and non-smokers should be displayed using different symbols on the same plot. Annotate your figure appropriately.
4. A (1-D) Gaussian (or *normal*) distribution is defined by the following equation:

$$\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (8.1)$$

where  $x$  is the distance from the mean,  $\mu$ , of the distribution. The

parameter  $\sigma$  is the standard deviation of the distribution, which affects how ‘spread out’ the distribution will be. An example of such a distribution is shown in Figure 8.7a.

- (a) Write a MATLAB function to display a figure of a 1-D Gaussian distribution. The function should take the following arguments: the standard deviation and mean of the Gaussian, and the minimum/maximum axis values.
- (b) Write another MATLAB function to display a 2-D Gaussian distribution. To make a 2-D distribution you should replace the  $(x - \mu)$  term in Equation (8.1) with a distance from a 2-D point which acts as the centre of the distribution (for example, the point  $(0, 0)$ ). See Figure 8.7b for an example. This function should take the following arguments: the standard deviation of the distribution, the  $x/y$  coordinates of the centre of the distribution, and the minimum/maximum axis values. You can experiment with different standard deviations but to begin with try a value of 3 and plot the distribution centred on  $(0, 0)$  and between the  $x/y$  coordinates of  $-10 \dots 10$ .

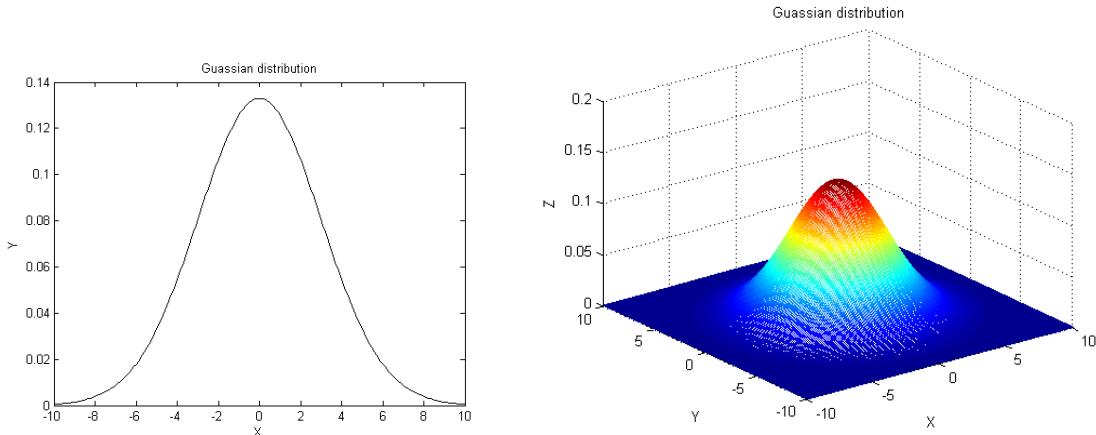


Figure 8.7: 1-D and 2-D Gaussian distributions.

5. Write a MATLAB program that will read in an image from a file and display it, and then allow the user to *annotate* the image (i.e. add text to it at specified locations). To perform the annotation the user should be prompted to enter a piece of text, then click their mouse in

the image to indicate where they want it to appear. The user should be allowed to add multiple annotations in this way, and the program should terminate and save the annotated image with a new file name when empty text is entered.

A sample image file, called *brain\_mr.tif*, which you can use to test your program, will be provided for you through the KEATS system.

(*Hint: look at the MATLAB documentation for the gtext command.*)

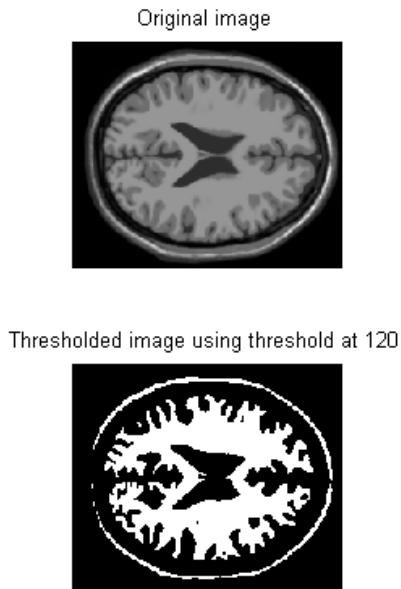


Figure 8.8: Image thresholding.

6. Write a MATLAB program to perform *image thresholding*. Image thresholding refers to creating a binary image from a greyscale image, where the binary image has a ‘high’ value (e.g. 1) where the corresponding original image intensity is greater or equal to a given threshold value, and a ‘low’ value (e.g. 0) otherwise.

Your program should first read in an image from a file. It should then display to the user the minimum and maximum intensities present in the image. The user should be prompted to enter a threshold value (which should be between the minimum and maximum values). A new thresholded image should then be created, with the value 255 for the

‘high’ pixels and 0 for the ‘low’ ones. A figure should be displayed showing both the original and thresholded images as subplots (as shown in Figure 8.8). Finally, the thresholded image should be written to a new external image file.

You can use the *brain\_mr.tif* file again to test your program.

7. Try visualising 3-D imaging data using the `imshow3D` function. The source code for `imshow3D` is available to you through KEATS, as well as a MATLAB *MAT* file containing some 3-D imaging data (called *MR\_heart.mat*).

## **Notes**

# 9 Code Efficiency

*At the end of this section you should be able to:*

- Explain the importance of programs running efficiently
- Explain the difference between time efficiency (speed) and memory efficiency (space)
- Explain the importance of a program running correctly as opposed to efficiently, i.e. that developing correctly running code is the first task and that optimising its performance is, in general, a later step
- Describe why recursive functions, if implemented naively, can lead to bad performance through repeated unnecessary calculation or through excessively deep levels of recursion
- Explain the fundamental ideas of dynamic programming that aim to address performance problems when running recursive programs

## 9.1 Introduction

When writing code that is more complex than a small or very simple program, it may become necessary to consider how efficient it is. There are two senses in which we can view the efficiency of code: in terms of the time it takes to run and in terms of the amount of computer memory it needs to run. In some cases, a program needs to be very time efficient, running and producing results quickly, for example in the software that is used by cars to manage the running of the engine or the software used in financial markets for automatic trading. In other cases, a program may need to be very economical with the amount of memory it uses, for example in programs running on devices where memory is limited such as an embedded processor in a domestic appliance or a mobile phone. This chapter will look at some aspects of efficiency, focusing on MATLAB specific examples, although some of the principles will be more generally applicable.

## 9.2 Time and Memory Efficiency

As a rule of thumb, when we want to assess the memory efficiency of a piece of code, we need to focus on the variables it uses, in particular on the largest arrays it uses. Recall that MATLAB defaults to double precision when assigning numeric values (see Section 5.2.2) which typically means we need eight bytes per element in an array containing numeric values.

When we want to assess how time efficient a piece of code is, we need to break it down into *fundamental instructions*. We make the simplifying assumption that these all roughly take the same amount of time each. Fundamental operations or instructions can include:

- Assigning a value to a variable, e.g. `a = 7;`
- Looking up a value of a particular element in an array, e.g. `myArray(3)`
- Comparing a pair of values, e.g. `if (a > 10) ...`
- Arithmetic operations like adding and multiplying, e.g. `a * 4`

**Example 9.1.** We consider a very simple example to illustrate some aspects of code efficiency. The following is code for a very trivial function that takes one argument (`N`) which is treated as an integer and adds up all the integers from 1 up to `N` and reports the result.

```
function sumOfIntegers(N)
% Usage: sumOfIntegers(N)
% Add up all the integers from 1 up to N and report
% the result.

a = 1:N;
total = 0;

for j = 1:N
    total = total + a(j);
end

fprintf('Sum of first %u integers = %u \n', N, total);
```

To keep things simple, we consider the first two assignments (to the variables `a` and `total`) as one assignment each<sup>12</sup>. Then there is a `for` loop which has `N` iterations. Within each iteration there is

- An array look-up: `a(j)`,
- A basic arithmetic operation: `total + a(j)`
- An assignment: `total = total + a(j);`

---

<sup>12</sup>Really, the assignment to variable `a` is an array assignment so requires allocating more memory than for a single numeric scalar value. This can add a time overhead which we ignore here.

Therefore, we have three operations per iteration which gives a total of  $3N$  operations for the entire loop. Counting in this way, we estimate that we need  $3N + 2$  fundamental operations for the whole function.

As we can see, the number of operations that the function will carry out depends upon the number passed in as the argument  $N$ . For large values, more operations are carried out.

### 9.2.1 Timing Commands in MATLAB

As we first saw in Section 2.7, there are two built-in commands in MATLAB that can be used to measure execution time. These are `tic` and `toc`. They can be used as follows:

```
tic  
% some  
% commands  
% here  
toc
```

If we just have a single command, we can use commas or semi-colons to separate them, for example:

```
>> tic; a = sin( exp(3) ); toc
```

or, using commas

```
tic, a = sin( exp(3) ), toc
```

The last version does not suppress the output so the value of  $a$  will be displayed to the command window. The important thing is that the time taken to execute the command will be reported:

```
>> tic, a = sin( exp(3) ), toc  
a =  
0.9445  
Elapsed time is 0.000355 seconds.
```

which shows that the above example took 355 microseconds. The same example, repeated on a different machine, can take a different amount of time because of differences in the hardware and the operating system. Even repeating the same operation on the same machine can lead to different amounts of time being taken because the tasks that the operating system is carrying out can change with time.

**Example 9.2.** Returning to the `sumOfIntegers` function from Example 9.1, we can measure how long the function takes to execute for various values of  $N$ . Let us try values of one thousand and one million for  $N$ . We'll focus on the main loop in the function. We can add a `tic/toc` pair around the loop so it becomes

```
% ...
tic
for j = 1:N
    total = total + a(j);
end
toc
% ...
```

Now, every time we run the function, it reports the amount of time the loop takes. Running for  $N = 1000$  and  $N = 1000,000$  gives

```
>> sumOfIntegers(1000)
Elapsed time is 0.000015 seconds.
The sum of the first 1000 integers is 500500

>> sumOfIntegers(1000000)
Elapsed time is 0.014182 seconds.
The sum of the first 1000000 integers is 500000500000
```

The ratio of the two times is *approximately* 1 to 1000 as we expect. It will not be exactly the ratio we expect because of variations in the operating system demand and the possibility that MATLAB might optimise the code in the background in some way.

### 9.2.2 Assessing Memory Efficiency

We have already seen that we can use the `whos` function to identify which variables are available in the workspace and, in particular, how much memory they are using (i.e. see Section 1.5).

When writing a function, however, we need to adopt a different strategy. We can inspect the code directly to find which variables will require the most memory. In the case of the function `sumOfIntegers` in Example 9.1, the variable that requires the most memory is the variable `a` declared in the command

```
a = 1:N;
```

This statement declares and assigns an array of length `N` when the function is run. Therefore, the amount of memory used by the function depends upon the argument given to it. The larger the number passed in, the more memory is needed.

Another way to inspect the amount of memory used by a function is more direct. We can actually run the function and use the debugger to stop the execution and check memory use directly. If we place a break point on the last line and call the function with a value of one thousand

```
>> sumOfIntegers(1000)
```

then, when the code stops, a call to `whos` gives the following:

```
K>> whos
  Name      Size            Bytes  Class    Attributes
  N          1x1                  8  double
  a          1x1000           8000  double
  j          1x1                  8  double
  total      1x1                  8  double
```

This shows the four variables within the scope of the function (`N`, `a`, `j`, and `total`) and how many bytes each one uses.

**Example 9.3.** Returning once again to the trivial function `sumOfIntegers`,

an obvious way to avoid a memory overhead is to avoid the allocation of the array at all. In this simple example this is quite easy: the code can be re-written as

```
function sumOfIntegers(N)
% ... usage
%
total = 0;

for j = 1:N
    total = total + j;
end
%
```

This is an easy win - the code did not need an array at all. But not all examples are as straightforward as this.

#### Example 9.4. A very very memory inefficient piece of code

We often need to use the built-in `ones` or `zeros` functions to quickly initialise an array containing only 1's or 0's. By default, if only one argument is given to these functions, they assume that the user requires a square matrix (i.e. a 2-D array).

An easy mistake to make is when a user requires a *vector* of zeros, i.e. a 1-D array, say a vector that has a length of 100,000. The user may mistakenly assume that it can be obtained by a call such as

```
myVector = zeros(100000) % Bad! do not do this...
```

MATLAB interprets this as the user asking for a square matrix of size  $100,000 \times 100,000$ . Therefore, it tries to allocate enough memory for  $100,000 \times 100,000 = 10^{10}$  double values. Each double value requires eight bytes because that is the default numeric precision for MATLAB. This means that the user is asking for 80000000000 bytes which is a little over 74GB because

$$\begin{array}{cccccc} \text{bytes/KB} & & \text{KB/MB} & & \text{MB/GB} \\ 74 & \times & 1024 & \times & 1024 & \times & 1024 \\ & & & & & & = 79456894976 \text{ bytes} \end{array}$$

Most ordinary computers do not have 74GB of RAM available, so typing such a command into MATLAB is very inadvisable and likely to cause either

MATLAB or the operating system to crash.

## 9.3 Tips for Improving Time Efficiency

A couple of tips for improving time efficiency are given below. These are to pre-allocate arrays that are going to be used and to avoid loops by using vector style operations.

### 9.3.1 Pre-Allocating Arrays

MATLAB is fairly relaxed about the length of an array and will adjust it automatically if the user decides to assign a value to it that is beyond its current limits. For example, if we run the command:

```
x = randi(3, 1, 6)
```

this will generate a length 6 row vector of random integers from {1, 2, 3}, for example (3, 1, 3, 1, 2, 2). If we make an assignment beyond the end of the array, e.g.  $x(10) = 5$  then it is automatically resized to a size 10 array and fills the ‘gap’ between the end of the original array and the new array with zeros. This means the vector will become (3, 1, 3, 1, 2, 2, 0, 0, 0, 5).

As discussed in Section 2.7 this can be convenient and enables the user to avoid worrying about whether the element being written to is within the array (yet) or not. The problem is that it is a costly operation in terms of time - there is an overhead involved in resizing an array and this is made worse if it is done within a loop.

For example, if we want to find the first 10 cubic numbers, we could write

```
% Start off with an empty array
cubes = [];
for j = 1:10
    cubes(j) = j^3;
end
```

In each iteration, we are assigning a value to the array `cubes` that is one step beyond the end of the array, i.e. resizing the array. Each resize has a cost associated with it.

As we saw in Section 2.7, we can avoid this cost by *pre-allocating* the array to achieve much better time efficiency:

```
cubes = zeros(1, 10);
for j = 1:10
    cubes(j) = j^3;
end
```

### 9.3.2 Avoiding Loops by using Built-In Functions and Vector Operations

Probably one of the most time-consuming parts of any program are loops such as `for` or `while` loops. This is especially true if the loops are *nested*, for example:

```
for j = 1:N
    for k = 1:M
        % Some code carried out here
    end
end
```

In this case, the code within the central loop is executed  $MN$  times (unless a `break` command is contained in the central section of code and can be reached).

In other words, nesting loops has a multiplicative effect on the number of operations needed. This is why nesting loops should only be done when it is unavoidable, especially if the number of loops nested within each other exceeds two.

MATLAB also offers ways to avoid loops. One way is to use the built-in functions it provides. For example, in the previous version of the `sumOfIntegers` function (Example 9.3), we can replace the entire loop

```
for j = 1:N
    total = total + j;
end
```

with a simple call to the built in function `sum`:

```
total = sum(1:N);
```

where we have used the array initialisation code `1:N` as an argument (without actually assigning it to an array). This single line will run faster than the original loop.

**Example 9.5.** Another example of the avoidance of loops is to remember that many built-in functions can accept array arguments. The code below:

```
x = linspace(0, 2*pi, 50);
y = zeros(size(x));
for j = 1:50
    y(j) = sin(x);
end
```

constructs fifty points on the graph  $y = \sin(x)$  between  $x = 0$  and  $x = 2\pi$ . It tries to improve efficiency by pre-allocating the array `y` to be the same size as `x`. However, since the built in `sin` function in MATLAB accepts an array argument, we can simply replace the loop with a one-line command to generate the array for variable `y` directly. This is done by applying the `sin` function element-wise to array `x`:

```
x = linspace(0, 2*pi, 50);
y = sin(x);
```

Operating in this way directly on arrays, in order to avoid loops, is often called vectorisation or vectorised calculation. It can also be applied when we have multiple arrays to process rather than just one as in the previous example.

**Example 9.6.** Suppose we have a very simple example where we have collected the widths, lengths and heights of four cuboids and we want to calculate their volumes. Say the dimensions are collected into three arrays:

```
w = [1, 2, 2, 8];
l = [2, 3, 5, 3];
h = [1, 3, 2, 4];
```

We could use a loop to find the volumes:

```
v = zeros(1,4);
```

```

for j = 1:4
    v(j) = w(j) * l(j) * h(j);
end

disp(v)

```

which gives the output:

```
2      18      20      96
```

However, this code can be speeded up by vectorisation. This requires the use of element-wise multiplication of our dimension data which can replace the above loop:

```
v = w .* l .* h;
```

Remember, this is a toy example (there are only 4 data points in it for a start) so the speed-up will be slight. But if the data arrays were much larger then the time saved would be more significant.

**Example 9.7.** We saw in Example 8.8 a surface plot of the sinc function. This relied on the generation of two arrays  $x$  and  $y$  using the `meshgrid` function and the subsequent calculation of the height of the function in an array  $z$ . We repeat the code (which uses vectorisation) here as a reminder:

```

% create arrays representing a grid of x/y values
[X,Y] = meshgrid(-8:.5:8, -8:.5:8);

% define sinc function in a good way
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R). ./ R;

```

The input arrays are both  $33 \times 33$  arrays and the resulting  $z$  array is also the same size because the element-wise operations ensure this. A bad way to achieve the same result as the above would be to use a loop, i.e.

```

% define sinc function in a really bad way
Z = zeros(33, 33);

%
```

```

for m = 1:33
    for n = 1:33
        r = sqrt(X(m,n)^2 + Y(m,n)^2) + eps;
        Z(m,n) = sin(r) / r;
    end
end

```

This code is inefficient because it uses two nested loops (bad) when it could have used element-wise operations instead.

### 9.3.3 Logical Indexing

Another way to speed up code and avoid loops is to use logical indexing which is described below in an example. It can be useful when we want to modify the contents of an array based on whether they meet a condition.

**Example 9.8.** The data we look at don't really matter. In this example, we start with an array of numbers created by the built-in `magic` function. This returns a square array of numbers with the ‘magic’ property that the rows, columns and diagonals add up to the same value.

```

>> magic(6)
ans =
    35     1     6    26    19    24
      3    32     7    21    23    25
    31     9     2    22    27    20
      8    28    33    17    10    15
    30     5    34    12    14    16
      4    36    29    13    18    11

```

The grid contains all the numbers from  $1$  to  $6 \times 6 = 36$  inclusive.

For the purposes of this example, we want to create a new array of the same size as `A` which has a value of one everywhere except at those locations where the element in `A` is divisible by three. We can do this by looping over all the elements in `A`, testing each one in turn and assigning the value to `B` accordingly. This can be done as follows:

```

N = 6;
% NxN magic square

```

```

A = magic(N);
% Initialise B to be the same size as A with ones
B = ones(N);

% Set B to one where A is divisible by 3
% Loop over all elements
for row = 1:N
    for col = 1:N
        % Get current element's value.
        value = A(row,col);
        % Is it a multiple of 3?
        if ( value / 3 == floor(value / 3) )
            B(row, col) = 0;
        end
    end
end

disp(B)

```

This code produces the array  $B$  as shown below. Note the pattern in the values of  $B$ , and confirm that it matches what you expect based on the values of the elements in  $A$ .

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |

Note how the code loops over all the elements, individually testing each one to see if it is divisible by 3. This is done with the test

```
if ( value / 3 == floor(value / 3) ) ...
```

which could also be done using the built-in modulo function called `mod` as follows

```
if ( mod(value, 3) == 0 ) ...
```

where the `mod` function returns the remainder after variable `value` is divided by 3.

There would be a little speed-up to be gained from replacing the test to use the `mod` function (because it would avoid repeatedly evaluating `value/3` for a start). But we can obtain a greater speed-up by using logical indexing, a vectorised operation that means we can discard the loops.

This can be done as follows:

```
N = 6;
A = magic(N);

B = ones(N);
indices = mod(A, 3) == 0;
B(indices) = 0;

disp(B)
```

Here, we see that the pair of nested loops in the original code is replaced by two lines. There is quite a lot going on in these two lines so we will break down what they do into smaller chunks.

The first line (`indices = mod(A, 3) == 0;`) identifies where in the array `A` we have multiples of 3. It starts with a call to `mod(A, 3)` - this applies the function `mod` *element-wise* to `A` to calculate the remainder after dividing by three.

Next, an equality test (`mod(A, 3) == 0`) compares the array of remainders returned by the modulo function with zero. This test is again applied element-wise and returns an array of *logical* values where *true* (i.e. logical 1) indicates that the remainder is zero and *false* (i.e. logical zero) indicates a non-zero remainder.

The array of logical values (with logical 1 indicating the elements of interest) is assigned to a variable `indices`. This is a logical indexing variable that we use in the second line.

The second line uses the `indices` variable directly with `B` and the round (indexing) brackets. It is an assignment to 0. All of the locations in the array `B` for which the `indices` variable has a logical true value will get assigned a value of 0.

We can make the code even more compact by avoiding the assignment to a separate `indices` variable. The two lines would then become a single line.

```
B(mod(A, 3) == 0) = 0;
```

As we have seen, there is quite a lot going on here. But the code is much faster than the original use of a pair of nested loops and it illustrates an important point:

**Code that has been optimised for efficiency  
tends to be harder to read**

Generally, it is better to start with code that may be slow and inefficient but where it is clear what the code does. Optimising the code should only be done a) if it is considered important and b) once we are sure the code is running correctly. This relates to a second key point:

**It is more important that code runs  
correctly than that it runs efficiently.**

These points may seem obvious but it is fairly common for them to be forgotten during routine programming.

#### 9.3.4 A Few More Tips for Efficient Code

These are adapted from the MATLAB help:

- Keep script files small. If a file gets large, then break it up and call one script from the other.
- Convert script files into functions. Functions will generally run a little faster than scripts. See Sections 3.2 and 3.4 for more information on function and script files.
- Keep functions as small and as simple as possible. If a function gets very long, look for parts of it that can be put into a new function and call it from the first one.
- Simple utility functions that can be called from lots of other functions are good. They can be put into their own script files and accessed easily (as long as they are in the MATLAB path (see Section 3.7). Otherwise, smaller functions can be included within the same file as the one containing the function that calls them.
- Avoid using names for variables and functions that are the same as built-in functions such as `sum`. Unfortunately the letter `i` is also a

built in variable, equalling the square root of -1, so be careful when using it if you also intend to use complex numbers.

- *To repeat:* Avoid loops if possible and try to use vectorisation and element-wise operations.
- *To repeat:* Pre-allocate arrays when it is known how large they will be. If it is not known then it can be better to pre-allocate an array to a size that is safely larger than what will be needed.

## 9.4 Recursive and Dynamic Programming

The definition of some problems can naturally lead to time consuming or memory demanding solutions when a program is written to solve them. Recursively defined problems are one such type of problem. For a recursive problem, it can be easy to write a program that naively ends up having a high computational cost. Dynamic programming (DP) is one way to try and reduce this cost. Before discussing this in more detail, we will recap on recursive programs and look at the number of calls that they can lead to.

**Example 9.9.** This is a continuation of Example 3.9 that looked at the factorial function. We will focus on the number of times that the function is called and with what arguments.

The code for evaluating the factorial  $n!$  of a number  $n$  is repeated here but we have changed the name slightly so that, for brevity, it is  $f(n)$ .

```
function result = f(n)
    % f(n) : return the factorial of n

    if (n<=0)
        result = 1;
    else
        result = n * f(n-1);
    end

end
```

If we want to evaluate  $f$  for  $n = 5$  say, then the call we make will be  $f(5)$  and this will in turn make a call to evaluate  $f(4)$  which in turn makes a call to evaluate  $f(3)$  and so on. The sequence of calls can be written

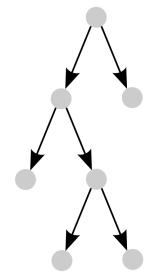
$$f(5) \rightarrow f(4) \rightarrow f(3) \rightarrow f(2) \rightarrow f(1) \rightarrow f(0)$$

The final call to  $f(0)$  is dealt with by the *ground case* in the `if` clause and no further recursive calls are made. In other words, after the original call to  $f(5)$  a further 5 calls to the function are made recursively before stopping. In general, evaluating the function  $f(n)$  will lead to  $n$  recursive calls to  $f$  in order to evaluate  $n!$ .

### Example 9.10.

**Full binary trees:** We define a *full binary tree* as one with set of nodes of which one is a *root* node and every node in the tree is either a *leaf* node or is the *parent* of exactly two *child* nodes. A leaf node is one with no children.

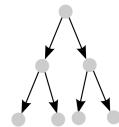
In the example on the right, there are seven nodes shown as grey circles. Each arrow connects a parent node at the top to a child node below. The root node is at the top of the whole tree, there are four leaf nodes and two intermediate nodes (neither a root nor a leaf).



A natural question to ask is *How many full binary trees exist for a given number of leaves?* The above example is one with four leaf nodes and we ask how many trees with four leaf nodes can be built?

One way to answer this question is to look at *sub-trees* below the root node. In the above example, the sub-tree below the root node on the left contains three leaf nodes and the sub-tree on the right contains the remaining single leaf node (i.e. the right-hand sub-tree consists of one node only).

In another tree with four leaf nodes, we could for example have two leaves in the left sub-tree and two leaves in the right sub-tree. An example is shown on the right.



Considering the sub-trees on either side of the root node provides us with a way to find the total number of trees for a given number of nodes. For the four leaf node example, we can have the following pairs of numbers of leaves assigned to each sub-tree beneath the root node:

$$(1, 3) \quad (2, 2) \quad (3, 1)$$

Let  $T_k$  be the number of trees with  $k$  leaf nodes. For each pairing above, we can work out the number of trees with that pairing by multiplying the number of possible subtrees on either side of the root. This is because, for any of the possible sub-trees on the left side of the root, we can match it with all the possible sub-trees on the right side of the root.

For example, if we make a  $(3, 1)$  split for the leaf nodes, then the total number of trees with this split will equal

$$\text{No. of trees with three leaves} \times \text{No. of trees with one leaf} = T_3 T_1$$

Adding such expressions up over all possible splits will then give us the total number of full binary trees with four leaves:

$$T_4 = T_1 T_3 + T_2 T_2 + T_3 T_1$$

Written this way we can see the recursive nature of a function that would evaluate the number of trees for a number of nodes. The value of  $T_4$  above depends on the previous values of  $T_k$  for  $k = 1, 2, 3$ .

We can write expressions for five and six etc. leaves similarly:

$$\begin{aligned} T_5 &= T_1 T_4 + T_2 T_3 + T_3 T_2 + T_4 T_1 \\ T_6 &= T_1 T_5 + T_2 T_4 + T_3 T_3 + T_4 T_2 + T_5 T_1 \\ &\dots \end{aligned}$$

A general formula for  $n$  leaves can be written as

$$T_N = \sum_{k=1}^{N-1} T_k T_{N-k}$$

We haven't actually calculated any values of  $T_k$  but we observe that if  $k = 1$ , i.e. when there is only one leaf node, we have a tree that consists of a single node only (i.e. the root and the leaf are the same node) so there is only one possible tree and  $T_1 = 1$ . This is the ground case.

**Example 9.11. A naive and inefficient recursive program for the number of trees**

We can see from the definition of  $T_N$  above that it is recursive and this leads to an obvious possible recursive program to evaluate it:<sup>13</sup>

```

function nTrees = T(N)

% Base case:
if N == 1
    nTrees = 1;
    return
end

% N > 1 here.

% Initialise value to calculate.
nTrees = 0;

% The main loop.
for k = 1:N-1
    nTrees = nTrees + T(k) * T(N-k);
end

end

```

The code has a ground case of  $N = 1$ , in which case there is a single leaf node. This would correspond to a single node in the whole tree and so there is only one possible tree. For larger values of  $N$ , recursive calls are made for smaller values ( $k$  and  $N - k$ ) and the results are multiplied together and accumulated.

If we consider the case  $N = 4$ , we can write out in full the calls that are made:

$$T(4) = T(1)T(3) + T(2)T(2) + T(3)T(1)$$

We see that  $T$  is called twice with the ground case ( $N = 1$ ), twice with  $N = 2$  and twice with  $N = 3$ .

- Each call with  $N = 3$  leads to calls  $T(1)T(2) + T(2)T(1)$ , i.e. two further calls with  $N = 1$  and  $N = 2$  each.
- Each call with  $N = 2$  leads to calls  $T(1)T(1)$ .
- Each call with  $N = 1$  returns the ground case value and stops the recursion.

---

<sup>13</sup>Note this code has been written compactly and without usage or many comments to save space. This is not an example of perfect programming.

A visualisation of the calls made when evaluating  $T(4)$  with this function is given on the right. The numbers along the top indicate the depth of the recursion for each call. Recursion depth 0 corresponds to the original call to the function which, in turn, makes calls to evaluate three pairs of terms at recursion depth 1 to find their products and add the results. These make calls at recursion depth 2 and so on. The calls when  $N = 1$  are marked in red as they are the ground case when each path of recursive calls stops.

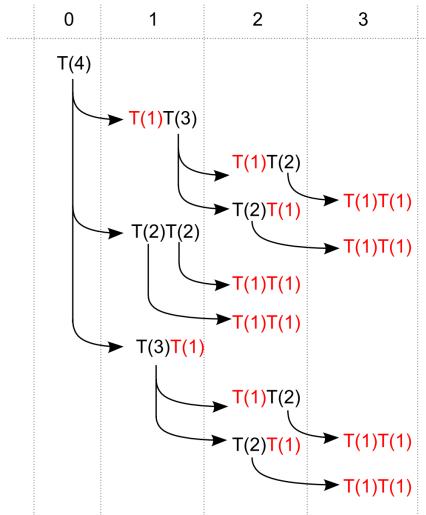
Adding up all these calls, including those within recursive ones, we see that the number of calls for each value of  $N$  below the original one ( $N = 4$ ) will be as follows:

| N               | 4 | 3 | 2 | 1  |
|-----------------|---|---|---|----|
| Number of calls | 1 | 2 | 6 | 18 |

As we can see, there are many repeated calls for smaller values of  $N$  that are really unnecessary because they are repeatedly calculating the number of trees for the same value of  $N$  (leaf nodes). This shows that the naive recursive implementation can lead to a very inefficient solution in which a lot of work is duplicated.

#### 9.4.1 A Note on Recursive Functions

We saw in the previous example that the call to evaluate  $T(4)$  led to recursive calls to a depth of three. In general, for this example, a call to evaluate  $T(N)$  will lead to a depth of  $N - 1$  in the final level of recursion. There are practical limits when running recursive functions and MATLAB has a built-in limit that stops the levels of recursion exceeding a certain number (typically 500). This is sensible because, theoretically at least, a recursive function has the potential to call itself to an infinite depth of recursion, which would use up all of the (finite) computer memory. This means that we should generally be careful when using recursive functions and avoid using them when the depth of recursion is likely to become high.



## 9.5 Dynamic Programming to Improve Performance

The above example that counts the number of full binary trees is a good example of one which can benefit from dynamic programming (DP). This is because it is computed by evaluating the same function on sub-problems of the original problem (in our case on sub-trees of the original tree).

The basic idea of DP is to store results that might need to be repeatedly calculated. This is known as *memoisation*. That is, one keeps a ‘memo’ of the results that have been already calculated and may be useful later and values of the function are only directly evaluated if they have not already been stored (memoised).

### Example 9.12. Using Dynamic programming for the tree counting example

In order to reduce the number of evaluations on the above tree counting example, we will write the code differently using ideas from dynamic programming.

We begin by writing the top-level function. This initialises an array for the memoised values and calls a helper function called `count`:

```
function nTrees = countFullBinaryTrees(N)

memoValues = -1 * ones(1,N);

nTrees = count(N, memoValues);

end
```

The helper function accepts two arguments, the value of  $N$  for which the tree count is required and the array of stored values. The stored value array (`memoVals`) is initialised so that every entry is -1, this indicates that the number of trees has not been calculated for any of the possible numbers of leaves from 1 up to  $N$ .

Now we can consider the helper function. The start of this function looks like this:

```
function [nTrees, memoVals] = count(N, memoVals)
```

```

% Helper function for counting the trees. Keeps
% a memoised list of values to prevent too many
% calculations.

% N > 1. Check if we have made the calculation for this ...
% value of N already.
if memoVals(N) > -1
    nTrees = memoVals(N);
    return
end

% ... continues below ...

```

Note how the function accepts the number of leaves  $N$  and the array for the stored values and returns two items: the number of trees and the array of stored values. The array of stored values also appears as a return value because it may be updated during a call.

The function begins by seeing if the required value has already been calculated. This is done by checking if the value stored at index  $N$  in `memoVals` array is greater than -1 (the initialisation value of each entry). If it is greater than -1, then the value was calculated earlier and we can assign it directly to the output argument `nTrees` and return. Job done.

Now let's look at the next part of the function:

```

% ...

% ground case:
if N == 1
    nTrees = 1;
    memoVals(N) = nTrees;
    return
end

% ... continues below ...

```

This part deals with the ground case when there is only one leaf in the tree. In this case there is only one possible tree and this is the count returned. Importantly, this count is stored in the `memoVals` array (at index 1) so that we can use it in any later repeated calls with  $N = 1$ .

Now we will look at the remainder of the function. This part is only evaluated

if  $N > 1$  and the number of trees for  $N$  has not yet been calculated and memoised.

```
% No recorded value. Need to calculate from scratch.  
% Initialise:  
nTrees = 0;  
  
for k=1:N-1  
    [nLeft , memoVals] = count(k, memoVals);  
    [nRight, memoVals] = count(N-k, memoVals);  
    nTrees = nTrees + nLeft * nRight;  
end  
  
% Store value we have just calculated so we don't  
% need to do this again.  
memoVals(N) = nTrees;  
  
end
```

Here, we make two recursive calls to the same function. Each time we collect both arguments, the count of the trees ( $nLeft/nRight$ ) and also the (possibly updated) `memoVals` array. After the loop, the number of trees is calculated and we do the important step of storing it in the array at the index  $N$ .

In the exercises, we will compare the speed of this memoising version of the tree counting function with the speed of the naive recursive implementation. It is sufficient to say here that the memoised version is much faster because it avoids a lot of unnecessary duplication of calculations for values which have previously been calculated and stored. We note that the use of a memoised array of previously calculated and cached results has led to an increase in memory use (i.e. the memory needed for the array itself). In the above case, this has been a small price to pay compared to the gain in speed but, in general, we need to monitor the memory used when seeking to adopt this approach to ensure that the memory usage of the program remains reasonable.

## 9.6 Summary

We have looked at some aspects and general tips to help improve code efficiency. These are generally useful and good advice but it is important to remain careful about good coding practice (See Section 4). The general

advice when coding is to proceed in two broad stages:

- Write code that is correct, clear, readable and tested
- Only when the above step is complete should we begin to think about performance and efficiency:
  - Decide whether or not improvements in efficiency are necessary
  - Only implement them if they are, re-writing appropriate parts of the code

The reason for this is that code that has been optimised to be as efficient as possible tends to be less readable and harder to debug when errors arise. Code that is possibly less efficient tends to take up more lines and is clearer and more easy to follow.

To stress this point: It is more important to have code that is *correct* than code that is fast or memory efficient. Once we are confident that the code is correct, we can consider changes for optimising efficiency.

In other words, it is okay to start off with code that works and is slow. We can then incrementally change it so that it becomes quicker, only changing small parts of the code at a time (in line with the incremental development model outlined in Section 4.2).

In the last part we looked specifically at recursive functions and how, if implemented naively, they can lead to a lot of unnecessary calculation. We took an idea from dynamic programming, in which we store or cache the results of calculations, to avoid such duplication. Generally this will lead to significant improvements in the calculation of recursive functions but we should remain aware that it can cause an increase in memory usage.

## 9.7 Further Resources

- The MATLAB built-in help has a number of tips on improving the performance of programs from a time or memory point of view. A good place to look is under *MATLAB → Advanced Software Development → Performance and Memory → Code Performance*
- Dynamic programming applies to a number of other areas
  - For a good general description of how it applies to a number of interesting problems follow the links from:  
[http://people.cs.clemson.edu/~bcdean/dp\\_practice/](http://people.cs.clemson.edu/~bcdean/dp_practice/).

- A more general tutorial can be found here:  
<http://www.codechef.com/wiki/tutorial-dynamic-programming>.
- A good list of challenge problems can be found at:  
[http://uva.onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=114](http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=114).
- Notes on memoisation are available at:  
<http://cs.wellesley.edu/cs231/fall01/memoization.pdf>
- The number of trees with a given number of leaves is an example of an application of Catalan Numbers which are interesting in their own right and have a diverse number of other applications:  
[http://en.wikipedia.org/wiki/Catalan\\_number](http://en.wikipedia.org/wiki/Catalan_number)

## 9.8 Exercises

1. (a) Calculate how much memory will be required by the following assignment

```
x = ones(1000)
```

- (b) Explain why a call to `z = 3 * zeros(1000000)` is likely to cause problems when it is run.
2. (a) Give two reasons why the following code is not efficient

```
myArr = 0;
N = 100000;

for n = 2:N
    myArr(n) = 2 + myArr(n-1);
end
```

- (b) Re-write the above code so that it runs more efficiently
- (c) Use the built-in timer functions to estimate the speed-up given when the above code is re-written
3. Re-write the following code so that it is more time efficient.

```
N = 20;
```

```
a = [];
for j = 1:N
```

```

a(j) = j * pi / N;
end

for k = 1:numel(a)
    b(k) = sin(a(k));
end

for m = 1:numel(a)
    c(m) = b(m) / a(m);
end

plot(a, b, 'r')
hold on
plot(a, c, 'k')
legend('sin', 'sinc')

```

4. (a) Use the built-in random function `rand` to generate a  $5 \times 6$  array of uniform random numbers, and assign this to an array called `A`.
  - (b) Use loops to inefficiently check the elements in `A`. If an element is less than 0.6, replace it with a value of 0.
  - (c) Use logical indexing to achieve the same result as in the previous part, i.e. by avoiding loops
  - (d) Use logical indexing to set any values in `A` that are greater than 0.7 to -1
  - (e) Use logical indexing to set any values in `A` that are greater than 0.7 *OR* less than 0.3 to -1.  
*(Hint: use the bitwise logical or operator `/`)*
5. (a) Write a MATLAB script to compute and display a 2-D array representing a times table for integers from 1 up to a specified maximum  $N$ . Use the `disp` statement to display the array. E.g. for  $N = 5$  the program should compute the following array:

|   |    |    |    |    |
|---|----|----|----|----|
| 1 | 2  | 3  | 4  | 5  |
| 2 | 4  | 6  | 8  | 10 |
| 3 | 6  | 9  | 12 | 15 |
| 4 | 8  | 12 | 16 | 20 |
| 5 | 10 | 15 | 20 | 25 |

For this implementation, concentrate on writing clear, understandable code that works.

- (b) Now write another implementation, this time optimised for speed. Run and time both implementations for values of  $N$  up to 200, and plot the execution times for both against the value of  $N$ . (Don't include the `disp` statement in the timed section.)
  - (c) Finally, write a third implementation that is optimised for memory.
6. Download from KEATS both versions of the code that counts the number of full binary trees with  $N$  leaves:
    - The naive recursive version (Example 9.11)
    - The version that uses memoisation (Example 9.12)
    - (a) Write a MATLAB script to measure the execution times of the two implementations for a single value of  $N$ .

Be careful when choosing the value of  $N$ . When using the naive version, a value of  $N$  greater than around 14 could lead to a long wait!
    - (b) Extend your script to compute the execution times for the two implementations for  $N=1 \dots 14$ , and plot both on the same graph against  $N$ .
  7. The Fibonacci sequence is

$$1, 1, 2, 3, 5, 8, \dots$$

and it is defined mathematically and recursively by the formula

$$f_1 = 1, \quad f_2 = 1, \quad f_{n+1} = f_n + f_{n-1}, \text{ for } n > 2$$

(Exercise 9 from Section 3 involved writing a recursive implementation of a function to compute Fibonacci numbers.)

- (a) If you haven't already, write the recursive function that implements the recursive Fibonacci formula. Call the function `fibonacciRecursive` and save it to a suitable `.m` file.

- (b) Each call to the function will make further calls recursively with smaller values of  $n$ . If the function is called originally (recursion depth 0) with a value of  $n = 5$ , work out how many calls there are for the function with values of  $n = 4, 3, 2$  and  $1$ , and how many calls in total.
- (c) Extend this to work out the number of all such calls to the function when we start with higher values of  $n$ , i.e.  $n = 6, n = 7$ , etc.
- (d) One way to avoid the overhead of multiple calls which evaluate the same value of the function in Exercise 7 is to avoid recursive calls altogether. One way to do this is calculate the values in a ‘bottom up’ way, starting with the values for the lowest values of  $n$  and ‘building up’. This way, we can visit each value of  $n$  once only.

By defining values for the current and previous terms in the sequence (or otherwise) write a function that loops upwards to the required value of  $n$ . Within each iteration, the following will need to be done:

- Calculate the value of the next term in the sequence
- Increment a counter that keeps track of how far we have moved along the sequence
- Update variables holding the previous and current terms based on the incremented counter  
*(Hint: use the value calculated in the first step).*
- Check if the loop has reached the required value of  $n$ , stopping if this is the case and returning the required value.

NB such *bottom-up evaluation* is another technique used in dynamic programming (as well as memoisation).

- (e) Estimate the speed up of the dynamic programming bottom-up evaluation of the Fibonacci code compared with the previous recursive version. How many times faster is the new code? It may be advisable not to call the previous version with a value much higher than 30 or so.

## **Notes**

# 10 Images and Image Processing

*At the end of this section you should be able to:*

- *Describe the basic concepts of how images are stored in a computer (including discrete locations (pixels), discrete intensities at pixels, data type and colour versus grey-scale images)*
- *'Read' or load an image into MATLAB using command-line expressions or some of the available interactive image tools*
- *Access the data for grey-scale images in the form of array variables*
- *Use MATLAB to get information about an image file, display an image and write an image to a file*
- *Describe some of the fundamental image processing steps and how we can carry these out using MATLAB commands*

## 10.1 Introduction

Processing images is an important task that can be carried out using programming. Specialist software can be developed to carry out a variety of tasks that manipulate and alter images in ways that can make them provide useful information. A very important area of image processing focuses on biological or medical images. This is because it is often the case that images are acquired in order to help clinicians make a diagnosis. These can range from microscopy images of individual cells to magnetic resonance (MR) images of the whole body. The examples below, from top-left, are a computed tomography (CT) image of the abdomen, an MR image of the head, a positron emission tomography (PET) image of the head, an ultrasound image of the abdomen and an X-ray of a hip.





This section will focus on some of the basic steps of image processing - simple techniques that are useful when processing biomedical images in particular and everyday images more widely. A few methods will be described in general terms and we will also show how they can be carried out in MATLAB.

## 10.2 Images on a Computer

All images, when stored in a computer, need to be represented in a way that the machine can understand. This means we need to have

- Discrete locations:
  - Computers cannot store continuous data. This means that the image data are stored for a finite number of discrete locations.
  - These locations are usually arranged on a regular grid, often called *pixels* (See Figure 10.1)
  - The image has a value (or intensity) associated with each pixel and we can visualise each pixel as a block of uniform colour or a block of uniform shading.
- Discrete intensities
  - As well as the locations being discrete, the intensity of the image at each pixel also needs to be discrete. This again is because computers cannot store continuous data.
- A data type
  - The most common way to store the intensity of each pixel is using an integer, often between 0 and 255.
  - This is because a number between 0 and 255 can be represented by an unsigned integer type in a single byte (8 bits), which makes it easier to store intensities for many pixels in large images.
  - It is also possible to use other data types such as floating point numbers or similar.

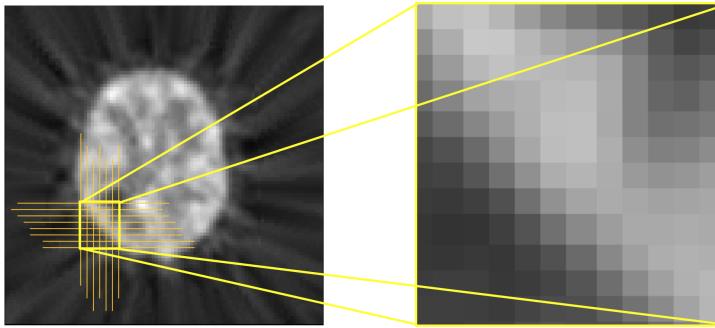


Figure 10.1: A PET image showing the individual pixels. These are the discrete locations at which we have a measurement.

### 10.2.1 Colour Versus Grey Scale Images

While most images in general use tend to be colour<sup>14</sup> many medical images tend to be grey-scale<sup>15</sup> images. A colour image typically uses three numbers to represent the red, green and blue intensities associated with each pixel. Grey-scale images only store a single value to represent a shade of grey between 0 (black) and a maximum value (white).

Storing the pixel data of a grey-scale image is easy. The simplest way is to store an unsigned one-byte integer with 0 representing black and 255 (the maximum value) representing white. An integer between 0 and 255 will be shown as an intermediate shade of grey.

It is also possible to have an image where each pixel is either black (minimum value) or white (maximum value). That is, we have no intermediate shades of grey. These are known as *binary* images.

Storing pixel data for a colour image can be done in more than one way. Perhaps the simplest is to repeat the same approach used for grey-scale images but with a separate number to represent the intensity of each of the red, green and blue (R, G and B) channels. In other words we can use 3 bytes, with one byte each for the red, green and blue intensities. For example, (0, 0, 255) would represent pure blue, (0, 255, 0) would represent pure green. A mixture such as (255, 0, 255) would represent magenta. A magenta colour with a lower intensity can be represented by (127, 0, 127) etc.

---

<sup>14</sup>color in the US

<sup>15</sup>gray-scale in the US

Using a numeric value for each of the R, G and B intensities is sometimes called a *true colour* representation.

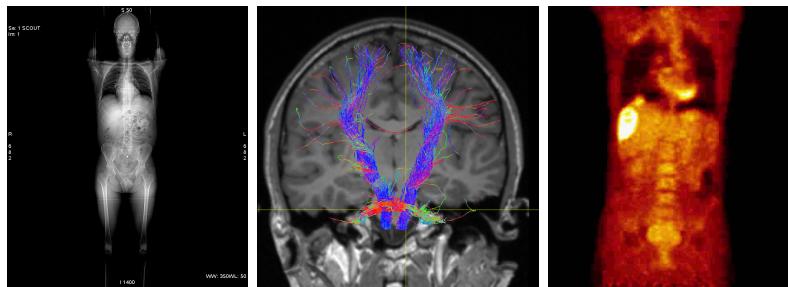
Another way to store pixel data for a colour image is to use a **colour map**. This is a list of triples representing RGB colours. In MATLAB, we can store a colour map as an array with three columns. The number of rows will need to be enough to capture all the colours in the image. Then, in order to represent the data for an individual pixel in an image, we only need to store a single integer which tells us which row in the colour map we should use for that pixel's colour. In other words, we need to know the row index to use for the pixel. This method of using a colour map and integers to look up rows in the map is called *indexed colour* representation and it is good for images where there are only a few colours.

## 10.3 Accessing Images in MATLAB

### 10.3.1 Accessing Information about an Image

We can use the built-in tool `imfinfo` to get information relating to an image file in MATLAB. We can just give the name of the file and a struct (see Section 5.10.1) is returned which contains information on different aspects of the image, for example its size. There are three example images provided via KEATS:

- `CT_ScoutView.jpg`: A view of a CT image of a body.
- `mr-fibre-tracks.png`: A slice of an MR image showing tracks of white matter in colour.
- `pet-image-liver.png`: A PET image showing metabolic activity in the liver.



We can use `imfinfo` to find out about each of these images. For example, the second image gives output like the following:

```

>> imfinfo('mr-fibre-tracks.png')

ans =
    Filename: '/PATH/TO/FILE/mr-fibre-tracks.png'
    FileModDate: '08-Sep-2014 10:48:42'
    FileSize: 192215
    Format: 'png'
    FormatVersion: []
    Width: 465
    Height: 464
    BitDepth: 24
    ColorType: 'truecolor'
% and some further information follows below ...

```

We can see that the size of the pixel grid is  $465 \times 464$ , and that the image is a true colour image. The struct has a `BitDepth` field which says that 24 bits are used for each pixel. In other words, three bytes are used for each pixel.

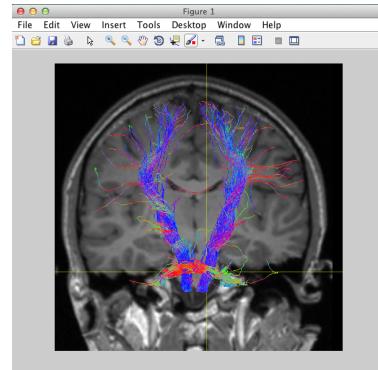
**Activity:** Use the `imfinfo` function to list the information for the other two images `CT_ScoutView.jpg` and `pet-image-liver.png`.

### 10.3.2 Viewing an Image

The simplest way to view an image from the command window is to use the `imshow` function, giving the name of the image file as an argument. For example,

```
imshow('mr-fibre-tracks.png')
```

The `imshow` function belongs to the Image Processing Toolbox. This toolbox is not part of the default MATLAB installation and needs a separate licence. If a call to `imshow` fails, it may be because the Toolbox is not licenced for your machine. The above call should result in the following display by MATLAB.



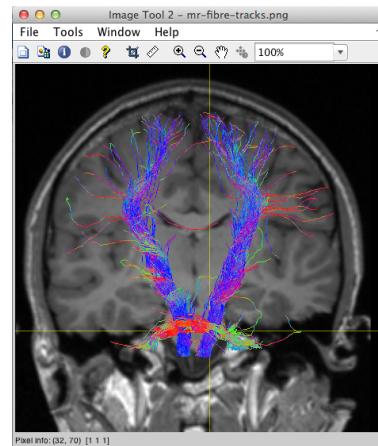
Here we see that `imshow` has loaded the image into a standard plotting window. The same tools that we can use for graph plots are also available in this window along the top.

Another way to view an image is to use the *interactive* image viewer provided by the Image Processing Toolbox. This provides more ways to control how an image is viewed and provides some tools for manipulating and editing an image.

We can start it by calling `imtool` with the name of the image file:

```
imtool('mr-fibre-tracks.png')
```

This command loads the image into an interactive viewer with a *Tools* menu that offers some basic ways of interacting with the image. For example, under *Tools* → *Image Information* we can access the same information as we obtained using `imfinfo` above.



It is also possible to start the viewer without loading an image by simply typing `imtool` at the command window. This will open the viewer with a blank screen and an image can be loaded by going to *File* → *Open...*

**Activity:** The images *CT\_ScoutView.jpg*, *mr-fibre-tracks.png* and *pet-image-liver.png* are available on KEATS. Try the different methods above for loading and viewing these images.

### 10.3.3 Accessing the Pixel Data for an Image

To keep things simple, we will assume from now on that all images being described are grey-scale and that we can simply represent each pixel's intensity by one number. Colour images are a little more complicated as we need to keep information for three separate channels.

The example image *CT\_ScoutView.jpg* is a single channel grey-scale image with an unsigned integer stored for each pixel. This means that we can store the intensities for all pixels in a single array. We can obtain the pixel data

using a call to the `imread` function (this is a standard MATLAB function, not part of the Image Processing Toolbox).

```
>> x = imread('CT_ScoutView.jpg');
>> whos
  Name      Size            Bytes  Class    Attributes
  x         1024x880        901120  uint8
```

Here we load the pixel intensities into the array `x`. A call to `whos` tells us the size of the array is  $1024 \times 880$  pixels and the data type stored is `uint8`, i.e. unsigned 8 bit (1 byte) integers.

As we saw in Section 8.2.4, we can use the `imshow` function to display the pixel data in array `x` directly (i.e. without using the image filename).

```
imshow(x);
```

#### 10.3.4 Viewing and Saving a Sub-Region of an Image

We do not have to view the whole image and can focus on a sub-region or ‘window’ inside the image. In the interactive viewer (`imtool`), we can use the zoom tool to adjust our view, zooming in and out as necessary.

We can also focus on a specific region using command window calls. After all, the image data is stored in an array and we can set ranges of row and column indices to decide on a view. The following can be done for example:

```
x = imread('CT_ScoutView.jpg');
imshow(x(25:350, 350:500))
```

which displays a region of the ‘scout’ CT image around the head and shoulders. Note that the rows specified (from 25 to 350) are counted starting from the top of the image. The columns chosen (from 350 to 500) are counted from the left. We can save our *cropped* version of the image data into a new image with the following command

```
imwrite(x(25:350, 350:500), 'ct-scout-crop.png')
```

This uses the built-in `imwrite` function that we saw in Section 8.2.4. This function can take two arguments: the array of image data to save and a

character array containing the file name to save it to.

**Activity:** Experiment with different crop regions and save or display the results. Use the command window and the tool built into the interactive `imtool` viewer. Can you choose a region which identifies the chest area?

## 10.4 Image Processing

Now that we have seen the ways in which we can load grey-scale image data, we will look at changing the pixel data using simple operations.

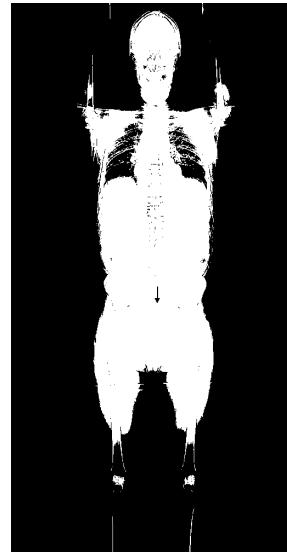
### 10.4.1 Binarising a Grey-Scale Image and Saving the Result

A very simple operation is to convert a grey-scale image to a binary one so that the resulting image has one of two values (e.g. ‘on’/‘off’, ‘true’/‘false’, 0/1). We can use logical tests to do this in MATLAB. For example, using the same CT image described earlier:

```
x = imread('CT_ScoutView.jpg');
y = x > 50 & x < 250;
```

This reads in the image pixel data to array `x`. The next line applies a pair of logical tests (element-wise) to `x` and applies the logical *and* (`&`) operator to the results. We have already seen the `&&` operator used for the logical *and* operation when using scalar values (see Section 2.3). When combining *arrays* of logical values as we are in this case we should use the `&` operator instead. Similarly, for the *or* operation on logical arrays we should use `|` rather than `||`. Therefore, this statement will produce an array (which is assigned to `y`) where a value of `true` or `1` is present for those pixels in the original image with an intensity between 50 and 250 (exclusive).

A call to `whos` shows that `y` is a logical array:



```
>> whos
```

| Name | Size     | Bytes  | Class   | Attributes |
|------|----------|--------|---------|------------|
| x    | 1024x880 | 901120 | uint8   |            |
| y    | 1024x880 | 901120 | logical |            |

We can view the result with a call to `imshow(y)` which is illustrated above. It is also possible to save the resulting array to a new image. This uses the `imwrite` function as in the previous section.

```
% Save our binary (logical) data
imwrite(y, 'ct-binary.png')
```

We can now use `imfinfo` to ask MATLAB to give us the general image information on the file we just saved. The salient parts of its output are given below:

```
>> imfinfo('ct-binary.png')

ans =

    Filename: '/PATH/TO/FILE/ct-binary.png'
    %
    % ...
    Format: 'png'
    Width: 880
    %
    % ...
    Height: 1024
    BitDepth: 1
    ColorType: 'grayscale'
    %
    % ...
```

Note how MATLAB has saved the file to a grey-scale image that uses one bit for each pixel. This is because one bit is sufficient to indicate whether a pixel in a binary image is off (0) or on (1).

**Activity:** Generate your own binary image and display it using your own values to decide which pixels are set to 0 and which are set to 1.

#### 10.4.2 Threshold-Based Operations

Threshold-based operations take an image and update each individual pixel depending on whether its value is above (or below) a user chosen *threshold value*.

As a simple example, we can set a threshold value  $tVal$  and set any pixel to zero in the new image if the old pixel value at the same location is less than  $tVal$ , otherwise we can leave the value the same. If we write  $old(i, j)$  and  $new(i, j)$  for the intensities of pixels in row  $i$  and column  $j$  in each of the old and new images, then this thresholding operation can be written:

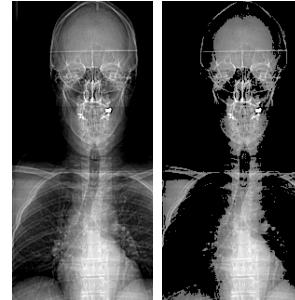
Threshold to zero (below), user gives  $tval$ :

$$new(i, j) = \begin{cases} 0 & \text{if } old(i, j) < tVal \\ old(i, j) & \text{otherwise} \end{cases}$$

We say the above operation ‘thresholds to zero’ (i.e. sets to zero any pixel below the threshold). In MATLAB we can easily do this with array operations.

```
x = imread('CT_ScoutView.jpg');
y = x;
tVal = 100;
y(x < tVal) = 0;
% View a sub-window of the ...
% input and output images.
imshow(x(25:350, 350:500))
imshow(y(25:350, 350:500))
```

The last two lines display a sub-window of each of the input and output (old and new) images so that we can see the effect of applying the threshold.



We can invert the way a threshold operation works, i.e. we can set to zero any pixel with a value *above* the threshold (rather than below). In other words:

Threshold to zero (above), user gives  $tval$ :

$$new(i, j) = \begin{cases} 0 & \text{if } old(i, j) > tVal \\ old(i, j) & \text{otherwise} \end{cases}$$

We can alternatively use the threshold to limit or truncate the values in an image, for example by setting a pixel to the threshold value if it previously was larger than the threshold:

Truncate to threshold (above), user gives *tval*:

$$new(i, j) = \begin{cases} tVal & \text{if } old(i, j) > tVal \\ old(i, j) & \text{otherwise} \end{cases}$$

The value assigned to a pixel above (or below) a threshold can be another choice for the user. In other words, as well as choosing a threshold value *tVal*, they can set a substitute value *newVal* to apply to pixels above (or below) the threshold. For example:

Reset pixels above threshold), user gives *tval*, *newVal*:

$$new(i, j) = \begin{cases} newVal & \text{if } old(i, j) > tVal \\ old(i, j) & \text{otherwise} \end{cases}$$

**Activity:** Make your own threshold-based filter based on one of the types shown above. Apply it to the CT image and save or view the result.

#### 10.4.3 Chaining Operations

We have only considered very simple operations so far but we can already build up more complex ones by chaining them together. For example, we can apply more than one threshold operation to an image where the output of one operation becomes the input to the next. Again, we start with the grey-scale CT image:

```
x = imread('CT_ScoutView.jpg');
y = x;
y(y < 50) = 0;
y(y > 200) = 0;
y(y > 0) = 255;
```

The first two lines read the image and copy it into an array variable *y*. The next three lines apply threshold operations and set to zero below and above, then assign a new value (255) to all positive pixels. Note how array *y* is updated on each line and modified in the next.

**Activity:** Run the code above and compare it with the output of the binarising code in Section 10.4.1. What do you notice?

This is a simple chain of operations and chains can be made as complicated as we like. A chain of operations is sometimes called a *pipeline*.

## 10.5 Summary

In this section we have looked at images and how we can read them and view them using MATLAB commands. In particular, we have looked at some of the properties of digital images (images that are stored on a computer) such as discrete locations and grey-scale versus colour images. Focusing on grey-scale images, we have introduced some of the basic steps that are used for *image processing* such as threshold-based operations. These operations rely mainly on the fact that, once an image is loaded into MATLAB, the data it contains can be represented by an array variable that can be manipulated to process the data. Some of the methods have used standard built-in MATLAB functions and some of the functions we have used are provided by the specialist Image Processing Toolbox. While these basic steps can be applied to images in general, they form the basis of the important field of *medical image processing* which has been used in a wide range of clinical settings since a variety of different types of medical image are now routinely available. The basic operations we have looked at can be ‘chained together’ with the output from one step forming the input for a later step. In this way image processing ‘pipelines’ can be built up incrementally.

You will learn more about medical image processing later in the BEng programme.

## 10.6 Further Resources

- The Image Processing Toolbox has a dedicated help section that is distinct from the main MATLAB help. It has sections including
  - *Import/export*: Reading images into MATLAB and saving them.
  - *Display*: Showing an image through the interactive viewer or by a command line call to a function such as `imshow`.

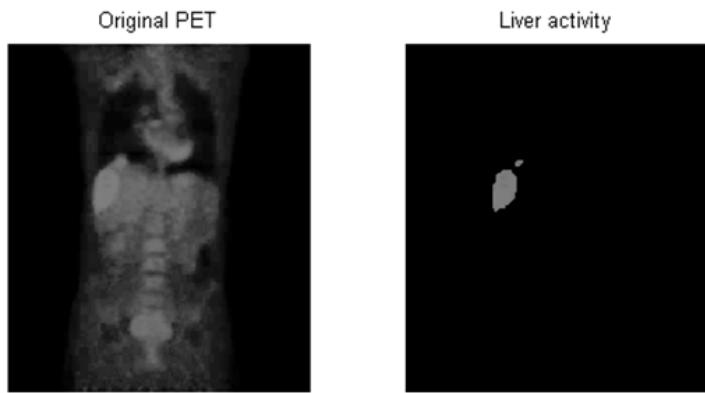
## 10.7 Exercises

These exercises involve use of a number of sample medical images which you will need to download from the KEATS system before proceeding.

1. Using the *CT\_ScoutView.jpg* image, produce, display and save a binarised image of the head using an appropriate threshold, i.e.



2. Write two functions called `thresholdLow` and `thresholdHigh`, which both take an image and a threshold as their two arguments, and produce a thresholded image as their result (the thresholded images should have zero intensity at any pixel that had a value below/above the threshold in the original image).
3. Write two functions called `truncateLow` and `truncateHigh`, which both take an image and a threshold as their two arguments, and produce a truncated image as their result (the truncated images should have the intensity of any pixel below/above the threshold set to the threshold).
4. Write a pipeline of operations to process *pet-image-liver.png* to highlight the area of high activity in the right liver, i.e. output of program should be:



(Also display a count of the number of pixels that are non-zero in the final image.)

5. Write a program to interactively help a user to apply some of the image processing operations we have seen in this section. It is up to you how to structure your code. The interaction with the user can be via the command window, for example through the `input` function, or via a GUI, for example by using a text box. As a minimum, your code should
  - Allow the user to specify an image file to load.
  - Allow the user to run a basic image processing operation
  - Allow the user to visualise or save the result

Preferably, the user should be able to chain operations together, applying one operation to the image that results from a previous one. It will be important to follow the advice on program design given in Section 7.

## **Notes**

# 11 Appendix

In this appendix, we discuss how to create graphical user interfaces (GUIs) using MATLAB. In the preceding chapters, we have been writing MATLAB programs that interact with the user by displaying text to the command window and reading data from the keyboard, but GUIs have an important role to play in making your programs more accessible and usable by a wider range of (non-technical) users. For the interested reader, an outline of how to build a simple graphical user interface (GUI) with basic components such as text boxes or axes is given using the built in `guide` tool. We also look at how to control the behaviour of GUI components using handles and callback functions

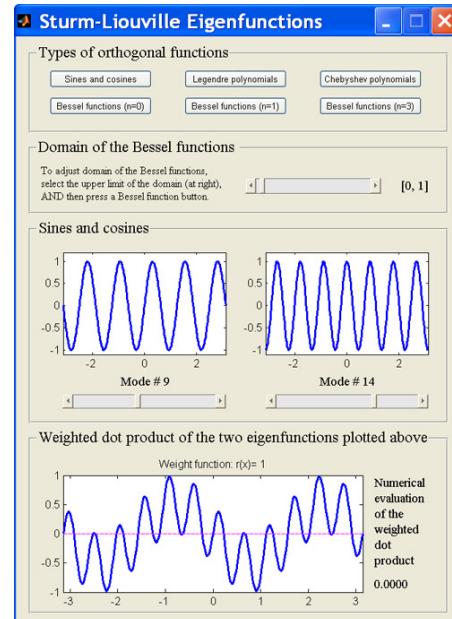
## 11.1 Graphical User Interfaces

Whilst a lot of interaction with a software environment can be carried out on a command line or by running a script, it can also be useful to use *Graphical User Interfaces* or GUIs.

In MATLAB, we can create our own custom-made GUIs, we can add objects including text-boxes for the user to enter text, pop-up menus for making choices, images or plots, and buttons for turning something on or off. We can customise the GUI, laying out the components and sizing them as we need.

When a user interacts with a component, such as clicking a button or entering some text, this is called an *event* and MATLAB executes a specific *callback function* that is associated with that event. We can edit the callback function to control the behaviour of the GUI depending on what the user did.

A single component can have more than one callback, one for each different



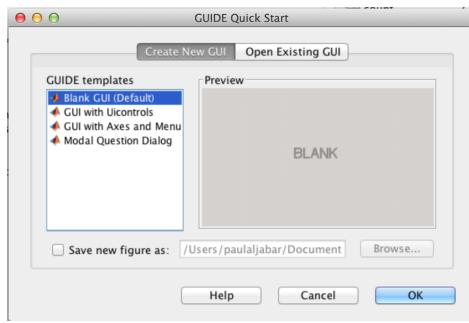
event (action by the user). For example, a set of axes (say one of those in the plots above) can have a callback function for when the user presses down the left mouse button and a different callback function for when the mouse simply moves inside the axes.

### 11.1.1 Building a GUI with the Guide Tool

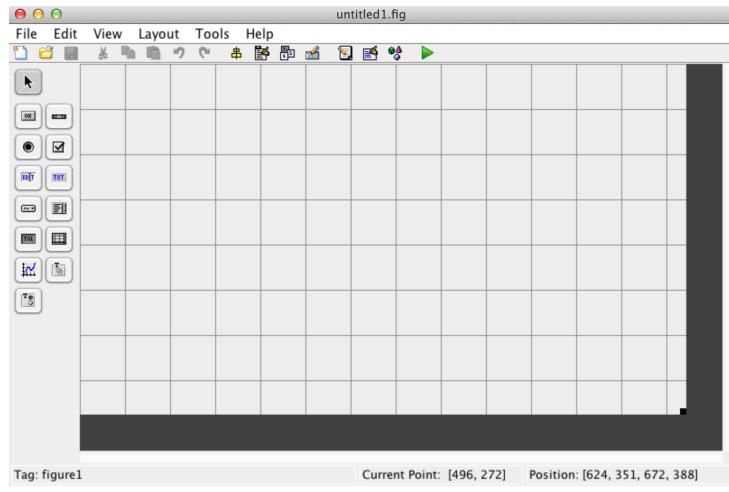
It is possible to create and build a MATLAB GUI entirely in code using scripts and functions. In practice, it is a lot easier to use the *Graphical User Interface Development Environment*, which is shortened to `guide`.

We will illustrate the use of the `guide` tool for building a GUI with a series of examples that build up a single GUI step by step. This GUI will carry out some basic plotting of a function that can be modified by the user.

#### Example 11.1. Creating a blank GUI and saving it

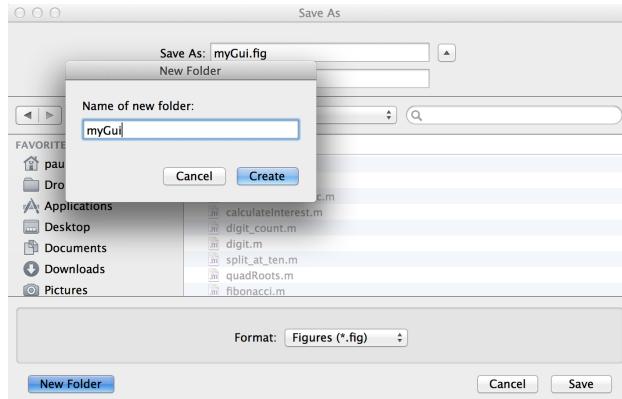


To start the tool, simply type `guide` at the command window. This causes a dialogue box to be displayed (see picture to the left) in which we can choose the type of GUI to build. Choose the default option of starting a blank GUI. This opens a blank canvas for our GUI which we can fill as we like (see picture below).



The blank canvas is overlaid with a grid to help us with the layout of the components (buttons, text-boxes, etc.) that are inserted. There is a set of buttons down the left hand side that we can use when editing. The button with the arrow icon at the top is used for selecting components and moving them around. Below that, there are buttons for various components that can be put in the GUI we are building. Hovering the mouse over a button gives some text to describe what the component is. For example, the button that has an icon showing a small graph of a function is used to insert a set of axes into the GUI for plotting graphs or displaying images.

Our GUI is still untitled so we will save it first before carrying on. Click *File* → *Save As* to open the save dialogue box. Because there will eventually be more than one file used by the GUI, it is a good idea to save the figure in a folder of its own where it can be kept together with any other files needed. For example, save it as *myGui.fig* to a new folder called *myGui* (see picture below).



After the save step, there will be two files in the folder: *myGui.fig* and *myGui.m*. The first is a binary file containing our GUI's window and components, and the second is a standard MATLAB *m*-file to contain our callback functions.

Also, after the save step, MATLAB opens an editor window with a lot of complex looking code in the *myGui.m* file. This code is automatically created by the guide.

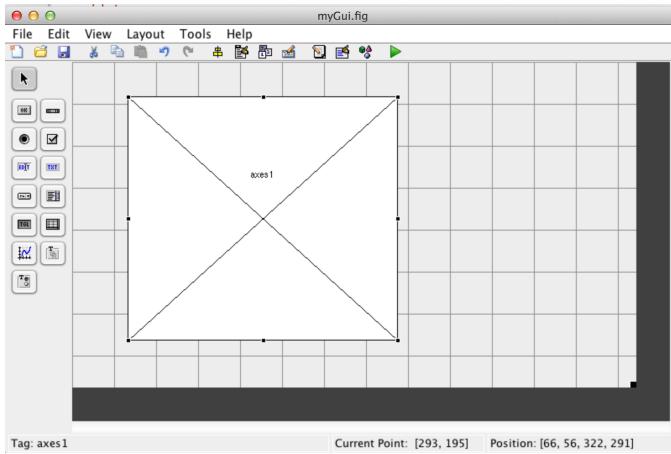
It is now possible to run the GUI we have just created, even though it does not contain any objects yet. This can be done by clicking the Run button at the top of the script editor window as we would for any *.m* file. MATLAB may complain about the script not being in the path, in which case, choose the option to *Change Folder* so the GUI can be run. This changes the MATLAB current directory to the location of *myGui*. When the GUI runs, as expected, we get a blank window that we cannot interact with yet. We can close the window to continue editing.

The GUI can also be run directly from the guide canvas screen which also has a green Run button at the top. Finally, it can also be run by typing *myGui* at the command prompt (as long as its folder is in the path or we have changed the working directory to that folder).

### **Example 11.2. Adding a component to the GUI**

Now we can add components to our GUI. We would like to plot, so we will start by adding a set of axes. Clicking on the button to insert axes makes the

cursor turn into a cross-hair: we can drag out a square or rectangle on the canvas to generate a set of axes. Our GUI canvas should now look something like this:



We can move and drag the corners of the axes until they are the size we want and where we want them to be. In the above example, we have left some space to the right of the axes for some components to be added later.

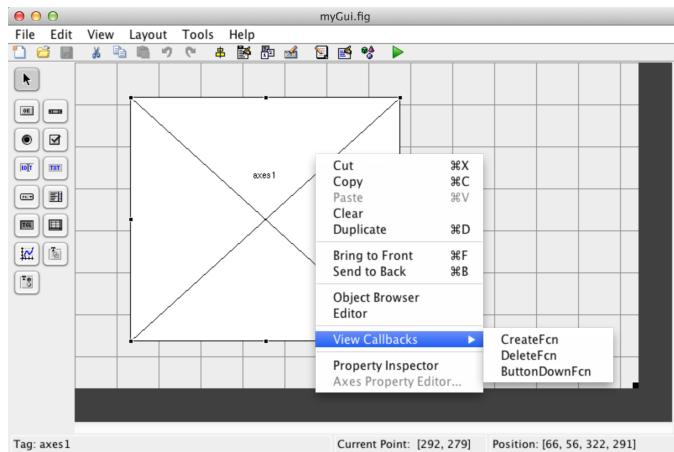
### 11.1.2 Controlling Components: Callback Functions and Events

Now that we have a component added to the GUI, we can use the code to control the components and how they behave. In MATLAB, and in some other programming languages that allow GUI programming, this can be done using *callback functions*. Callback functions are special functions that are associated with each component on the GUI. They are associated with *events* that occur when the user interacts with the GUI, e.g. by clicking on an item in the GUI ('ButtonDown'). Other less obvious events include the creation of each of the components as the GUI is starting up or their 'deletion' when the GUI closes.

When an event occurs, the system will execute code in the specific callback function associated with that event - it is in this function that we can write our own code to control the GUI's behaviour.

#### Example 11.3. Accessing the callback function for a component

Now that we have a component (some axes) on our GUI in the guide editor, we can have a look at what callback functions it has available. We can do this by right-clicking on our component (the axes) to bring up a context menu. As shown below, we can *View callbacks* in this menu. They can also be accessed from the *View* menu at the top of the guide.



In this case, we have three functions available as callbacks: CreateFcn, DeleteFcn and ButtonDownFcn. These are associated with the creation, deletion and button-down events for the axes.

If we click on the first of these in the menu (CreateFcn) then the guide takes us to some automatically generated code for this function:

```
% Executes during object creation, after setting all properties.
function axes1_CreateFcn(hObject, eventdata, handles)
% hObject
% handle to axes1 (see GCBO)
% eventdata
% reserved - to be defined in a future version of MATLAB
% handles
% empty - handles not created until after all CreateFcns called
% Hint: place code in OpeningFcn to populate axes1
```

This specially generated code is run when the axes are created as the GUI is being loaded. The name of the function reflects the fact that the axes we inserted have been given the label *axes1* and hence the function is named *axes1\_CreateFcn*. This would distinguish them from a second set of axes that could be inserted into the GUI.

The usage text for the function tells us that the first argument is an object

handle to our axes (`hObject`), and that the second argument can be ignored. It then tells us that the third argument (`handles`) is currently empty and not usable. We will look further at the `handles` variable later as we consider how to pass information between the various functions.

From this unpromising start, the last line of the usage gives us a hint to add code to a specific function in order to populate our axes. This is what we shall do next.

#### Example 11.4. Adding code to populate the axes

The hint in the usage text for the axes creation code above suggests that we add code to the GUI's opening function and this is what we will do here. The GUI's opening function is in the same file as the automatically generated code. The function we are looking for is naturally named `myGui_OpeningFcn` as it is associated with the whole GUI (*myGui*) and it is the opening function (i.e. just before it is made visible). Currently, this automatically generated code is mostly comments describing what the function does:

```
% ---- Executes just before myGui is made visible.
function myGui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata   reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to myGui (see VARARGIN)

% Choose default command line output for myGui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

We can inspect the function's behaviour by debugging the code. Place a break point on the last line of actual code (`guidata(hObject, handles);`) and press the *Run* button. The debugger should stop in the code before the GUI is made visible.

There is only really one variable of interest at this point, the `handles` variable. If we type its name at the command line and hit return as follows:

```
>> handles
handles =
figure1: 173.0077
```

```

axes1: 174.0077
output: 173.0077

```

we see that `handles` is a struct with three fields, `figure1`, `axes1` and `output`. Our axes are the second of these fields and we can plot to them by referring to their handle. To do this, we can use a form of the `plot` command where we specify the axes we want to plot to as the first argument.

We will use one of the functions from Figure 8.2 in this example. Stop the debugger and add code to the `myGui_OpeningFcn` function so that it looks like the following:

```

% Choose default command line output for myGui
handles.output = hObject;

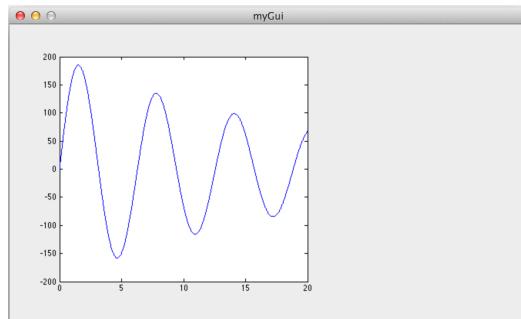
x = 0:0.01:20;
y = 200*exp(-0.05*x).*sin(x);
plot(handles.axes1, x, y)

% Update handles structure
guidata(hObject, handles);

```

The three lines in the middle give the code to insert. Note how the `plot` command has `handles.axes1` as the first argument. The function plotted represents an exponentially decaying oscillation with an amplitude of 200, a decay factor of 0.05 and a wavelength of  $2\pi$ .

Now when we run the GUI, our axes will contain a plot:



We can view the decay factor and amplitude of the function as parameters that we can vary and in the next example, we will use a text box to do this for one of them.

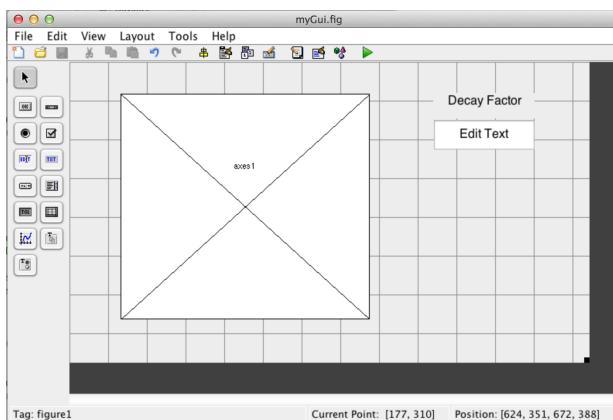
### Example 11.5. Using a text box to vary the plot

There are two tools (buttons) on the guide for inserting text components onto a GUI: *Edit Text* and *Static Text*. The first inserts a text box that the user can enter text into, including numeric values. The second cannot be modified by the user and simply provides information.

We can put both kinds of text onto our GUI: a text-box for entering a decay factor for the function and some static text to give the text box a label.

By default the Static Text component will just display the text ‘*Static Text*’ and we need to change this. Click to highlight the component and, from the *View* menu or the (right-click) context menu, choose *Property Inspector*. Find the *String* field and change it from ‘*Static Text*’ to ‘*Decay Factor*’.

Now highlight the Edit Text component and open its Property Inspector. Find the *Background Colour* field and change it to white to emphasise that it can be edited by the user. Then go to the *String* field and delete the ‘*Edit Text*’ that is there by default so that the box is empty. Select *File → Save* on the guide or click the Save icon. The GUI canvas should now look something like this:



### Example 11.6. Check what callbacks are available for text boxes

Now we can look at the available callback functions for both types of text-boxes. Both of them have the following callback functions:

- CreateFcn
- DeleteFcn
- ButtonDownFcn

The first and second callbacks relate to the creation and deletion of the component and the third is called when a mouse click takes place over the component.

The Edit Text box has two more callback functions with the following names:

- Callback
- KeyPressFcn

The second of these is called when the user presses *any* key inside the editable text box.

The first callback (with the imaginative name ‘Callback’ !) is the more useful one; it is called after the user has entered some text into the box and hit Enter. We will use this function.

### **Example 11.7. Getting user input from an edit text box with the callback function**

Hit save on the guide or highlight the Edit Text box and go to *View* → *View Callbacks* → *Callback* so that the guide automatically generates the starting code for this function.

The code for the *Callback* function of our edit text box contains the definition line and some comments on usage. These should include hints on how to use the function:

```
function edit1_Callback(hObject, eventdata, handles)
% hObject    handle to edit1 (see GCBO)
% eventdata   reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit1 as text
%         str2double(get(hObject,'String')) returns contents of edit1 as a double
```

The function name for the specific text-box is *edit1\_Callback* and the hints give an idea of how to access what the user enters into it. As before, the first argument to the function is a handle to our text box, which in this case is called *edit1*.

We can access what the user enters as a standard character array using:

```
get(hObject, 'String')
```

This uses the `get` function to ask for a property named ‘String’ that is possessed by the text box object.

Or, if we want to interpret what the user has entered as a number we can repeat the above and pass what `get` returns to the built-in `str2double` function to convert the character array to a double precision floating point number:

```
str2double(get(hObject, 'String'))
```

If the user enters text that cannot be interpreted as a number (e.g. ‘Sd!’) then `str2double` will return NaN (See Section 5.3).

We can put the following code into the callback for the edit text box to allow the user to modify the plot:

```
decayFactor = str2double(get(hObject, 'String'));

if isnan(decayFactor)
    return
end

x = 0:0.01:20;
y = 200*exp(-1*decayFactor*x).*sin(x);

plot(handles.axes1, x, y)
```

This code takes the user input text and tries to convert it to a number, if this is not possible (i.e. if `decayFactor` is NaN) then the function simply returns. Otherwise, the function re-calculates the new `y` values for the given decay factor and re-plots the figure.

Note how we have used the third argument to the callback function (the one named `handles`) to access the axes component on the GUI so that we can plot into it.

Run the code above and experiment by changing the decay factor in the edit text box. The box should start off empty but you can insert a number into

the box and hit Enter. The plot should be updated to reflect the values you put in.

Inserting a non-numeric value into the box, e.g. ‘blah’, should not affect the plot.

### 11.1.3 Maintaining State

In Example 11.4 we added the first code for plotting to the axes (in the function `myGui_OpeningFcn`).

In Example 11.7, we added code to the function `edit1_Callback` to update the plot if the user gives a decay factor.

This means that we now have two places in the code that plot to the axes. It would be good to avoid such unnecessary duplication of code so we will now make a single plot function and call it from the other parts of the code when it is needed.

In order to do this, we need to be able to pass information between the different parts of the code. We will do this in the next example by using the `handles` variable.

#### Example 11.8. Maintaining state and avoiding duplicated code

So we start by adding a function to the end of the script `myGui.m`

```
function myPlot(handles)
    % Plots the function with the current parameters to the axes.

    x = 0:0.01:20;
    y = 200*exp(-1* handles.decayFac .*x).*sin(x);

    plot(handles.axes1, x, y)
```

Note how this function expects to receive a `handles` structure and that it also expects that one of the fields of the structure is called `decayFac` that contains the decay factor to use.

We need to call our new plotting function from two places in the code:

- Inside the opening function `myGui_OpeningFcn`

- Inside the edit text box callback function `edit1_Callback`

For the first place, we need to remove the code that plots inside `myGui_OpeningFcn`, i.e. *delete* the following:

```
x = 0:0.01:20;
y = 200*exp(-0.05*x).*sin(x);

plot(handles.axes1, x, y)
```

and replace it with the following code:

```
handles.decayFac = 0.05;
myPlot(handles)
```

For the second place, we need to delete the following code from `edit1_Callback`

```
x = 0:0.01:20;
y = 200*exp(-1*decayFactor*x).*sin(x);
plot(handles.axes1, x, y)
```

and replace it with the following:

```
handles.decayFac = decayFactor;

myPlot(handles);

% Update handles structure
guidata(hObject, handles);
```

Note how in both places, we have set a field called `decayFac` in the `handles` structure. We do this in preparation for the call to the `myPlot` function which expects it.

Note also that in the second place, we make a call to an important function called `guidata` - this call was already present in `myGui_OpeningFcn` so we did not need to add it there. It is an important function because it ensures that the updates we make to the `handles` variable *persist*, i.e. they can be used by functions that use it as an argument later on.

If we add more code later that also adds information as fields in the `handles` variable, then we will also need to make a call to the `guidata` function at the end, using the same pair of arguments (`hObject` and `handles`).

Check that the code is running properly by testing it with a range of decay factors, including non-numeric values.

### Example 11.9. Ensuring the edit text box shows the correct value when starting the GUI

We saw in Example 11.5 how we can set what is displayed in a text box using the property inspector. Here we will control what is shown in the edit text box in code so that, as the GUI is being created and loaded, it displays the initial decay factor.

This can be done by adding one line in the GUI's opening function, `myGui_OpeningFcn`. This uses a call to the `set` function to adjust the text box's string property:

```
set(handles.edit1, 'String', '0.05')
```

Here, we simply put in the starting value for the decay factor.

Put this line in somewhere near the start of the function and test that the code is working properly, i.e. that the correct value is displayed when the GUI starts.

## 11.2 Summary

MATLAB provides a set of tools for creating Graphical User Interfaces (GUIs). We have seen how we may use the `guide` tool to design a GUI by positioning and sizing components such as text boxes and axes. Further components such as buttons and pop-up menus can also be added.

The `guide` provides automatically generated skeleton code for special functions called *callback functions*. These can be used to control the behaviour of the GUI when *events* occur, such as the user clicking inside a plot or entering text into an edit box.

## 11.3 Further Resources

- Mathworks File Exchange:  
<http://www.mathworks.co.uk/matlabcentral/fileexchange>

- The MATLAB help documentation has some examples of GUI building under *GUI Building* → *GUI Building Basics* → *Examples and Howto*. Online version at this [link](#). It would be a useful exercise to work through these as well as the examples in this chapter.

## 11.4 Exercises

1. Create a new blank GUI using the guide tool and carry out the following steps:
  - (a) Add two text boxes to the GUI: one static text box and one edit text box.
  - (b) Save the GUI in a folder called guiEx1.
  - (c) Run the GUI to make sure it works.
  - (d) Close the GUI and change the code in the edit text box callback function so that the text entered by the user is displayed in the other (static) text box.

*Hints:*

- The *handles* variable in the automatically generated code will have fields for each of the edit and static text boxes. These should be named *edit1* and *text1* (although the number might vary).
- Use the *set* and *get* functions to set and access the '*String*' properties in the text boxes.

2. This question is a continuation of the GUI examples in the main text (Examples 11.1 to 11.9).

Recall that the default amplitude of the exponential decay function is hard-coded in the script. It is set to a value of 200 in the *myPlot* function. This task is to allow the user to change the amplitude.

Add another pair of text boxes to the GUI using the guide tool: one static text box and one edit text box. Use the edit text box to take a number that is used for the amplitude of the function plotted. Set the string for both new text boxes in the script during the call to the opening function for the GUI.

3. This question continues from the previous question and uses the *myGui* plotting GUI.

Modify the code so that, if a user gives a non-numeric value in an edit text box, the value displayed is re-set to the correct number. That is, the decay factor edit text box should show the current decay factor value (instead of the non-numeric value that the user entered). Similar behaviour should be shown by the amplitude edit text box.

4. This question continues from the previous questions and also uses the *myGui* plotting GUI.

Insert a pop-up menu onto the GUI to control the colour of the plot. The choice should be between black, red, blue or green.

*Hints:*

- Use the *handles* object to store a field for the plot colour. This should be the single character that MATLAB uses for plot colours: '*k*', '*r*', '*b*', '*g*'.
- Set the '*String*' property for the pop-up menu in the main GUI opening function. Assign a cell array of character arrays to contain what is displayed to the user.
- Update the *myPlot* function so that it uses the plot colour field from the *handles* structure when it actually calls the *plot* function.

5. Use the *guide* to create a blank GUI. Save it as *guiEx2*.

Add two edit text boxes (their tags should be *edit1* and *edit2*) and a static text box (its tag should be *text1*).

- (a) Write code to set the edit boxes to show a (string) value of 1 at opening.
- (b) Write code so that the static text box shows the sum of the two numbers shown in the edit text boxes.

*Hints:*

- Use the *handles* structure to store numeric values for the contents of the edit text boxes.

- Use a separate function called `runOperation` to actually carry out the adding and the updating of the static text box.
- (c) Write code to guard against non-numeric input in the edit text boxes, i.e. ensure that all stored and displayed values are still numeric even if the user makes a mistake.
- (d) Use the `guide` to add a Button Group to the GUI. Add four Radio Buttons to it. We will use each button for one of the four basic arithmetic operations, so modify them so their strings are '`'add'`', '`'multiply'`', '`'subtract'`', and '`'divide'`'. Modify the string of the panel containing the buttons so that it says 'operation'.

Add code to the script and modify the `runOperation` function so that the correct operation is applied to the current values in the edit boxes.

*Hints:*

- Add a field to the handles to store the current operation.
- Use a `switch` statement in `runOperation` to select which operation to apply.

## 12 Exercise Solutions

## 12.1 Chapter 1 - Introduction to Computer Programming and MATLAB

Solutions are shown in framed text.

1. First, write a command to clear the MATLAB workspace. Then, create the following variables:
  - (a) A variable called `a` which has the character value ‘q’
  - (b) A variable called `b` which has the Boolean value true
  - (c) A variable called `c` which is an array of integers between 1 and 10
  - (d) A variable called `d` which is an array of characters with the values ‘h’, ‘e’, ‘l’, ‘l’, ‘o’

Now find out the data types of all variables you just created.

```
clear
a = 'q';
b = true;
c = 1:10;
d = 'hello';
whos
```

2. Create a list of all even integers between 32 and 55 and save it to a text file called `evens.txt`. Use the comma character as the delimiter.

```
a = 32:2:55;
writematrix(a, 'evens.txt');
```

3. Let `x = [5 2 4 7 9 1]`.
  - (a) Add 3 to each element.
  - (b) Add 4 to just the even-indexed elements.
  - (c) Create a new array, `y`, with each element containing the square root of the corresponding element of `x`.

- (d) Create a new array,  $z$ , with each element containing the cube of the corresponding element of  $x$ .

```

x = [5 2 4 7 9 1];
x = x + 3;
x(2:2:end) = x(2:2:end) + 4;
y = sqrt(x);
z = x.^3;

```

4. A class of students have had their heights in centimetres measured as: 159, 185, 170, 169, 163, 180, 177, 172, 168 and 175. The same students' weights in kilograms are: 59, 88, 75, 77, 68, 90, 93, 76, 70 and 82. Use MATLAB to compute the mean and standard deviation of the students' height and weight.

```

h = [159, 185, 170, 169, 163, 180, 177, 172, ...
168, 175];
w = [ 59, 88, 75, 77, 68, 90, 93, 76, 70, 82];
mean(h)
std(h)
mean(w)
std(w)

```

5. Let  $a = [1 \ 3 \ 1 \ 2]$  and  $b = [7 \ 10 \ 3 \ 11]$ .

- (a) Sum all elements in  $a$  and add the result to each element of  $b$ .
- (b) Raise each element of  $b$  to the power of the corresponding element of  $a$ .
- (c) Divide each element of  $b$  by 4.

```

a = [1 3 1 2];
b = [7 10 3 11];
b = b + sum(a);
b = b.^a;
b = b / 4; or b = b./4;

```

6. Write MATLAB commands to compute the *sine*, *cosine* and *tangent*

of an array of numbers between 0 and  $2\pi$  (in steps of 0.1). Save the original input array and all three output arrays to a single *MAT* file and then clear the workspace.

```
x = 0:0.01:(2*pi);
s=sin(x);
c=cos(x);
t=tan(x);
save('trig.mat','x', 's', 'c', 't');
clear;
```

7. Write a MATLAB *m*-file to read in a user's height (in metres) and weight (in kilograms) from a user, then compute and display the user's body mass index (BMI), according to the formula

$$\text{BMI} = \text{mass}/\text{height}^2.$$

(Hint: look at the MATLAB documentation for the *input* command.)

```
height = input('Enter height in metres: ');
weight = input('Enter weight in kilograms: ');
BMI = weight / (height^2)
```

8. Write a MATLAB *m*-file to read a floating point number from the user, which represents a radius  $r$ , and then compute and print the surface area and volume of a sphere of that radius, according to the following equations:

$$\text{Surface Area} = 4\pi r^2, \quad \text{Volume} = \frac{4}{3}\pi r^3$$

```
r = input('Enter radius: ');
sa = 4 * pi * r^2
v = 4/3 * pi * r^3
```

9. Write a MATLAB *m*-file to read in three floating point values from the user ( $a$ ,  $b$  and  $c$ ), which represent the lengths of three sides of a triangle. Then compute the area of the triangle according to the equation:

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

where  $s$  is half of the sum of the three sides.

```
a = input('Enter a: ');
b = input('Enter b: ');
c = input('Enter c: ');
s = (a + b + c) / 2;
area = sqrt(s * (s-a) * (s-b) * (s-c))
```

10. Let  $m = [1 \ 2 \ 3 \ 4 \ 5 \ 6]$  and  $n = [10 \ 1 \ 8 \ 5 \ 3 \ 0]$ . Plot  $m$  against  $n$ .

```
m = [1 2 3 4 5 6];
n = [10 1 8 5 3 0];
plot(m, n, 'o-b');
```

11. Write a MATLAB *m*-file to plot curves of the *sine*, *cosine* and *tangent* functions between 0 and  $2\pi$ . Your script should display all three curves on the same graph and annotate the plot. Limit the range of the *y*-axis to between -2 to 2.

```
x = 0:0.1:(2*pi);
s = sin(x);
c = cos(x);
t = tan(x);
plot(x,s,'k-', x,c, 'b-', x,t, 'r-');
title('Sine, cosine and tangent plots');
legend('Sine', 'Cosine', 'Tangent');
axis([0 (2*pi)-2 2]);
```

12. Write a single command that will create a  $3 \times 4$  matrix in which all elements have the value -3.

```
m = -3 * ones(3,4);
```

13. Write a single command that will create a  $4 \times 4$  matrix containing all zeros apart from 3s on the diagonal.

```
m = eye(4)* 3;
```

14. Write a MATLAB *m*-file to define the following matrices:

$$X = \begin{pmatrix} 4 & 2 & 4 \\ 7 & 6 & 3 \end{pmatrix}, Y = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \end{pmatrix}, Z = \begin{pmatrix} 2 & 0 & 9 \\ 1 & 5 & 7 \end{pmatrix}.$$

Now perform the following operations:

- Replace the coefficient in the second row and second column of Z with 1
- Replace the first row of X with [2 2 2]
- Replace the first row of Y with the second row of Z

```
X = [4 2 4; 7 6 3];
Y = [1 2 3; 1 0 0];
Z = [2 0 9; 1 5 7];
Z(2,2)= 1
X(1,:)= [2 2 2]
Y(1,:)= Z(2,:)
```

15. The injury severity score (ISS) is a medical score to assess trauma severity. Data on ISS and hospital stay (in days) have been collected from a number of patients who were admitted to hospital after accidents. The ISS data are [64 35 50 46 59 41 27 39 66], and the length of stay data are [8 2 5 5 4 3 1 4 6]. Use MATLAB to plot the relationship between ISS and hospital stay.

If you select *Tools* → *Basic Fitting* in the MATLAB figure window it is possible to fit regression lines between datasets. Using the Basic Fitting window, fit a linear regression line to these data and estimate what length of hospital stay would be expected for a patient with an ISS of 55.

```
iss = [64 35 50 46 59 41 27 39 66];
hs = [8 2 5 5 4 3 1 4 6]
```

```
plot(iiss, hs, 'x')
```

Select *Tools* → *Basic Fitting*, then under TYPES OF FIT check Linear. Then under INTERPOLATION/EXTRAPOLATION, enter 55 into the x box and click on the green arrow symbol.

16. The MATLAB script shown below is intended to read a matrix from a text file and display how many coefficients the matrix has. However, the code does not currently work correctly. Use the MATLAB code analyser and debugger to identify the bug(s) and fix the code.

```
clear
m = load('m.txt');
s = size(m);
n = s^2;
disp(['The matrix has ' num2str(n) ' elements']);
```

(This file *debug\_exercise.m* and the test file *m.txt* are available to you from within KEATS.)

The `size` function returns an array result so it is not correct to use the `^` operator on it. The correct solution should multiply the two elements of the output of the `size` function, which represent the number of rows and columns respectively. The corrected code should be:

```
clear
m = load('m.txt');
s = size(m);
n = s(1) * s(2);
disp(['The matrix has ' num2str(n) ' elements']);
```

17. Data has been collected of the ages and diastolic/systolic blood pressures of a group of patients who are taking part in a clinical trial. All data is available to you through the KEATS system in the files *age.txt*, *dbp.txt* and *sbp.txt*. Write a MATLAB *m*-file that reads in the data and creates a plot containing two subplots: one of age against systolic

blood pressure and one of age against diastolic blood pressure. Your script should annotate both subplots.

(Hint: look at the MATLAB documentation for the subplot command.)

```
clear
sbp = load('sbp.txt');
dbp = load('dbp.txt');
age = load('age.txt');
subplot(2,1,1);
plot(age, sbp, 'x');
title('Age vs. systolic blood pressure');
xlabel('Age (yrs)');
ylabel('Systolic blood pressure (mmHg)');
subplot(2,1,2);
plot(age, dbp, 'x');
title('Age vs. diastolic blood pressure');
xlabel('Age (yrs)');
ylabel('Diastolic blood pressure (mmHg)');
```

18. A short way of determining if an integer is divisible by 9 is to sum the digits of the number: if the sum is equal to 9 then the original number is also divisible by 9. For integers up to 90 this requires only a single application of this rule. Write MATLAB commands to verify the rule, i.e. create an array of all multiples of 9 between 9 and 90, and compute the sums of their two digits.

(Hint: look at the MATLAB documentation for the idivide and mod commands. These can be used to find the result and remainder after integer division, so if you use them with a denominator of 10 they will return the two digits of a decimal number. Because these two commands only work with integer values you will also need to know how to convert from floating point numbers to integers: type doc int16.)

```
nums = 9:9:90;
nums = int16(nums);
digit1 = idivide(nums, 10);
digit2 = mod(nums, 10);
```

digit1 + digit2

## 12.2 Chapter 2 - Control Structures

Solutions are shown in framed text.

1. Write a script that asks the user to enter a number, and then uses an `if` statement to display either the text “odd” or “even” depending on whether the number is odd or even.

(Hint: see the documentation for the `mod` function)

```
n = input ('Enter number: ');
if (mod(n, 2)==1)
    disp('Odd');
else
    disp('Even');
end
```

2. If  $a=1$ ,  $b=2$  and  $c=3$ , use your knowledge of the rules of operator precedence to predict the values of the following expressions (i.e. work them out in your head and write down the answers). Verify your predictions by evaluating them using MATLAB.

- (a)  $a+b * c$
- (b)  $a ^ b+c$
- (c)  $2 * a == 2 \&& c - 1 == b$
- (d)  $b + 1:6$

- (a) 7
- (b) 4
- (c) `true` (MATLAB displays Boolean values as either 1 for true or 0 for false)
- (d) [3 4 5 6]

3. Cardiac resynchronisation therapy (CRT) is a treatment for heart failure that involves implantation of a pacemaker to control the beating

of the heart. A number of indicators are used to assess if potential patients are suitable for CRT. These can include:

- New York Heart Association (NYHA) class 3 or 4 - this is a number representing the severity of symptoms, ranging from 1 (mild) to 4 (severe);
- 6 minute walk distance (6MWD) less than 225 metres; and
- Left ventricular ejection fraction (EF) less than 35%.

Write a MATLAB *m*-file containing a single `if` statement that decides if a patient is suitable for CRT, based on the values of three numerical variables: `nyha`, `sixmwd` and `ef`, which represent the indicators listed above. All three conditions must be met for a patient to be considered suitable. You can test your program by assigning values to these variables from the test data shown in Table 2.3.

```
if (nyha >= 3) && (sixmwd < 225) && (ef < 35)
    disp('Patient suitable for CRT');
else
    disp('Patient not suitable for CRT');
end
```

4. Not all patients respond positively to CRT treatment. Common indicators of success include:

- Decrease in NYHA class of at least 1;
- Increase in 6MWD of at least 10% (e.g. from 200m to 220m); and
- Increase in EF of at least 10% (e.g. from 30% to 40%).

Table 2.4 shows post-CRT data for patients 2 and 4 from Exercise 3 (who were considered suitable for CRT). Write a MATLAB *m*-file that determines if a patient has responded positively to CRT treatment. All three conditions must be met for a patient to be considered a positive responder. The program should use a single `if` statement which tests conditions based on the values of six numerical variables: `nyha_pre`, `sixmwd_pre`, `ef_pre`, `nyha_post`, `sixmwd_post` and `ef_post`, which represent the pre- and post-treatment indicators.

```

if ((nyha_pre - nyha_post)>= 1)&& ...
(((sixmwd_post-sixmwd_pre)/ssixmwd_pre)> 0.1)&& ...
((ef_post - ef_pre)> 10)
    disp('Responder');
else
    disp('Non-responder');
end

```

5. Write a MATLAB *m*-file to read a character from the keyboard, then read a number. Based on the value of the character the program should compute either the *sine* of the number (if the character was a ‘s’), the *cosine* (if it was a ‘c’) or the *tangent* (if it was a ‘t’). If none of these three characters was entered a suitable error message should be displayed. Use a *switch* statement in your program.

(*Hint: if you use the input command to read a character, by default it will try to ‘evaluate’ it as a variable. To prevent this, add a second argument ‘s’. See the MATLAB documentation for input for details.*)

```

op = input('(s)in, (c)os, (t)an:', 's');
n = input('Enter number:');
switch op
    case 's'
        disp(['Sin(' num2str(n) ')= ' num2str(sin(n))']);
    case 'c'
        disp(['Cos(' num2str(n) ')= ' num2str(cos(n))]);
    case 't'
        disp(['Tan(' num2str(n) ')= ' num2str(tan(n))]);
    otherwise
        disp('Invalid character');
end

```

6. The normal human heart rate ranges from 60-100 beats per minute (BPM) at rest. *Bradycardia* refers to a slow heart rate, defined as below 60 BPM. *Tachycardia* refers to a fast heart rate, defined as above 100 BPM. Write a MATLAB *m*-file to read in a heart rate from the keyboard, and then display the text “Warning, abnormal heart rate” if

it is outside of the normal range.

```
hr = input('Enter heart rate:');
if (hr < 60) || (hr > 100)
    disp('Warning, abnormal heart rate');
end
```

7. Now modify your solution to Exercise 6 to display one of two warning messages, depending on whether the heart rate indicates bradycardia or tachycardia. If the heart rate is within the normal range the text “Normal heart rate” should be displayed.

```
hr = input('Enter heart rate:');
if (hr < 60)
    disp('Warning, bradycardia');
else
    if (hr > 100)
        disp('Warning, tachycardia');
    else
        disp('Normal heart rate');
    end
end
```

8. In Section 1 Exercise 7 you wrote MATLAB code to compute a patient’s body mass index (BMI), according to the formula

$$\text{BMI} = \text{mass}/\text{height}^2$$

based on a height (in metres) and weight (in kilograms) read in from the keyboard. Modify this code to subsequently display a text message based on the computed BMI as indicated in Table 2.5.

```
height = input('Enter height in metres: ');
weight = input('Enter weight in kilograms: ');
BMI = weight / (height^2);
if (BMI <= 18.5)
    disp('Underweight');
else
```

```

if (BMI <= 25)
    disp('Normal');
else
    if (BMI <= 30)
        disp('Overweight');
    else
        disp('Obese');
    end
end
end

```

9. Write a MATLAB *m*-file that reads in two numbers from the keyboard followed by a character. Depending on the value of the character, the program should output the result of the corresponding arithmetic operation ('+', '−', '\*' or '/'). If none of these characters was entered a suitable error message should be displayed. Use a `switch` statement in your program.

```

x = input('Enter operand 1: ');
y = input('Enter operand 2: ');
op = input('Enter operation (+, -, *, /): ', 's');
switch op
    case '+'
        disp(['Result = ' num2str(x+y)]);
    case '-'
        disp(['Result = ' num2str(x-y)]);
    case '*'
        disp(['Result = ' num2str(x*y)]);
    case '/'
        disp(['Result = ' num2str(x/y)]);
    otherwise
        disp('Invalid operation');
end

```

10. Figure 2.5<sup>16</sup> shows how blood pressure can be classified based on the

---

<sup>16</sup>Adapted from <http://www.bloodpressureuk.org/BloodPressureandyou/Thebasics/Bloodpressurechart>

diastolic and systolic pressures. Write a MATLAB *m*-file to display a message indicating the classification based on the values of two variables representing the diastolic and systolic pressures. The two blood pressure values should be read in from the keyboard.

```

dia = input('Enter diastolic pressure:');
sys = input('Enter systolic pressure:');
if (sys < 90)&& (dia < 60)
    disp('Low b.p.');
else
    if (sys < 120)&& (dia < 80)
        disp('Ideal b.p.');
    else
        if (sys < 140)&& (dia < 90)
            disp('Pre-high b.p.');
        else
            disp('High b.p.');
        end
    end
end

```

11. Write a `switch` statement that tests the value of a character variable called `grade`. Depending on the value of `grade` a message should be displayed as shown in the table below. If none of the listed values are matched the text “Unknown grade” should be displayed.

| <u>Grade</u> | <u>Message</u>  |
|--------------|-----------------|
| ‘A’ or ‘a’   | “Excellent”     |
| ‘B’ or ‘b’   | “Good”          |
| ‘C’ or ‘c’   | “OK”            |
| ‘D’ or ‘d’   | “Below average” |
| ‘F’ or ‘f’   | “Fail”          |

```

switch grade
case {'a', 'A'}
    disp('Excellent');
case {'b', 'B'}
    disp('Good');

```

```

case {'c', 'C'}
    disp('OK');
case {'d', 'D'}
    disp('Below average');
case {'f', 'F'}
    disp('Fail');
otherwise
    disp('Unknown grade');
end

```

12. Write MATLAB code to solve the following problems:

- (a) Write a script containing a `for` loop that displays the square roots of the numbers from 1 to 20.
- (b) Extend the script to read a number from the user and use this as the maximum number.
- (c) Further extend the script so that it displays the square root values only if they are less than 5.

```

(a) for x = 1:20
    disp(sqrt(x));
end

(b) nmax = input('Enter max number: ');
for x = 1:nmax
    disp(sqrt(x));
end

(c) nmax = input('Enter max number: ');
for x = 1:nmax
    if (sqrt(x)< 5)
        disp(sqrt(x));
    end
end

```

13. Write a MATLAB *m*-file that uses an appropriate iterative programming construct to compute the value of the expression:

$$\sum_{a=1}^5 a^3 + a^2 + a$$

```
asum = 0;
for a=1:5
    asum = asum + a^3 + a^2 + a;
end
asum
```

14. Write a MATLAB *m*-file to read in an integer  $N$  from the keyboard and then use an appropriate iterative programming construct to compute the value of the expression:

$$\sum_{n=1}^N \frac{n+1}{\sqrt{n}} + n^2$$

```
N = input('Enter N: ');
val = 0;
for n=1:N
    val = val + (n+1)/sqrt(n)+ n^2;
end
val
```

15. Write a MATLAB *m*-file that continually displays random numbers between 0-1, each time asking the user if they want to continue. If a ‘y’ is entered the program should display another random number, if not it should terminate.

```
c = 'y';
while (c == 'y')
    num = rand(1,1)
    c = input('Continue? ', 's');
end
```

16. Modify your solution to Exercise 15 so that rather than displaying a single random number, it asks the user how many random numbers they would like. If they enter a number greater than zero the program displays that many random numbers. Otherwise it should repeat the request for the number of random numbers. The program should still terminate when any character other than a ‘y’ is entered when the user

is asked if they want to continue.

```
c = 'y';
while (c == 'y')
    n = input('How many numbers? ');
    if (n < 1)
        continue;
    end
    num = rand(1,n)
    c = input('Continue? ', 's');
end
```

17. The Taylor series for the exponential function  $e^x$  is:

$$1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} \dots$$

Write a MATLAB *m*-file to read a number  $x$  from the keyboard and compute its exponential using the Taylor series. Also compute it using the built-in MATLAB function `exp`. Run your program several times with different numbers of terms in the Taylor series to see the effect this has on the closeness of the approximation.

```
x = input('Enter x: ');
val = 1;
for n=1:10
    val = val + (x^n) / factorial(n);
end
val
exp(x)
```

18. Now modify your solution to Exercise 17 so that, as well as computing and displaying the result of the Taylor series expansion, it also computes and displays the number of terms required for the approximation to be accurate to within 0.01.

```
% read x and compute correct answer e^x
x = input('Enter x: ');
correct = exp(x);
```

```

% loop over number of terms in expansion
nTerms = 1;
close_enough = false;
while ~close_enough

    % compute Taylor series
    estimated = 1;
    for n=1:(nTerms-1)
        estimated = estimated + (x^n) / factorial(n);
    end

    % close enough?
    if (abs(estimated - correct) < 0.01)
        close_enough = true;
    else
        % increment number of terms
        nTerms = nTerms + 1;
    end
end

disp(['e^' num2str(x) ' = ' num2str(correct)]);
disp(['Within 0.01 for ' num2str(nTerms) ' terms']);

```

19. Write a MATLAB *m*-file to compute the value of the expression

$$\sin(x) + \frac{1}{2}\cos(x)$$

for values of  $x$  between 0 and  $2\pi$  in steps of 0.0001, and produce a plot of the results.

Write two separate MATLAB *m*-files to implement this task: one that produces the result using MATLAB array processing operations as we learnt about in Section 1; and one that builds up the array result element-by-element using an iterative programming construct. Add code to time your two implementations. Which is quicker? Why?

Implementation 1 (using array processing):

```
tic
x = 0:0.00001:(2*pi);
y = sin(x)+ 0.5 * cos(x);
plot(x,y);
toc
```

Implementation 2 (using iterative construct):

```
tic
y = [];
for x = 0:0.0001:(2*pi)
    y = [y; sin(x)+ 0.5 * cos(x)];
end
plot(0:0.0001:(2*pi),y);
toc
```

The first implementation should run faster because MATLAB will allocate memory for the entire array once. In the second implementation MATLAB will not know how big the array should be and may have to copy it to a different part of the computer's memory to make extra space.

20. Modify your solution to Exercise 5 so that, rather than reading a single character/number, the program continually reads character/number pairs and displays the trigonometry result. The program should terminate when the character 'x' is entered. In this case, no number should be read and the program should exit immediately.

```
op = '';%initialise so loop is entered
while (op ~= 'x')
    op = input('(s)in, (c)os, (t)an, e(x)it:', 's');
    if (op ~= 'x')%only if 'x' not entered
        n = input('Enter number:');
        switch op
            case 's'
                disp(['Sin(' num2str(n) ')= '...
                    num2str(sin(n))]);
            case 'c'
```

```

        disp(['Cos('num2str(n)')= '...
            num2str(cos(n))]);
    case 't'
        disp(['Tan('num2str(n)')= '...
            num2str(tan(n))]);
    otherwise
        disp('Invalid character');
    end
end
end

```

21. In Section 1 Exercise 15 we introduced the concept of the injury severity score (ISS). The ISS is a medical score to assess trauma severity and is calculated as follows. Each injury is assigned an abbreviated injury scale (AIS) score between 0-5 which is allocated to one of six body regions (head, face, chest, abdomen, extremities and external). Only the highest AIS score in each body region is used. The three most severely injured body regions (i.e. with the highest AIS values) have their AIS scores squared and added together to produce the ISS score. Therefore, the value of the ISS score is always between 0-75.
- Write a MATLAB *m*-file to compute the ISS given an array of 6 AIS values, which represent the most severe injuries to each of the six body regions. Test your code using the array [3 0 4 5 3 0], for which the ISS should be 50.

```

ais = [3 0 4 5 3 0];
iss = 0;
for x=1:3
    % find value/index of maximum ais score
    [c,i] = max(ais);
    % add on square of highest ais to iss
    iss = iss + c*c;
    % remove highest ais from list
    ais = [ais(1:i-1)ais(i+1:end)];
end
iss

```

22. In Section 1 Exercise 18 you wrote MATLAB code to check if an integer was divisible by 9 by summing its digits: if they add up to 9 it is divisible by 9. For integers up to 90 only a single application of the rule is required. For larger integers, multiple applications may be required. For example, for 99 we add its digits getting the result 18. Then, because 18 still has multiple digits we add them again to get 9, confirming that 99 is divisible by 9. You should continue adding the digits until you are left with a single number.
- Write a MATLAB *m*-file to read in a single number from the keyboard, and use the rule to determine it is divisible by 9. An appropriate message should be displayed depending on the result. You can assume that the number will always be less than 1000.

```
% read number and convert to integer
num = input('Enter number (<1000): ');
num = int16(num);

% one iteration for each application of rule
apply_rule = true;
while apply_rule
    digit1 = idivide(num, 100); %100's
    num = num - digit1*100;
    digit2 = idivide(num, 10); %10's
    digit3 = mod(num, 10); %1's

    % sum digits
    num = digit1 + digit2 + digit3;

    % more applications of rule required?
    if (num < 10)
        apply_rule = false;
    end
end

% display message
```

```
if (num == 9)
    disp('Divisible by 9');
else
    disp('Not divisible by 9');
end
```

## 12.3 Chapter 3 - Functions

Solutions are shown in framed text.

1. Write MATLAB code to solve the following problems:
  - (a) Write a function called `mycube` to compute and return the cube of a number provided as an argument.
  - (b) Call this function several times from the command window using different input argument values.
  - (c) Write a script to call the function twice with different input arguments and assign the results to two different variables.
  - (d) Extend the script to read a number from the user and supply this as an argument to the function. The script should display the result of the calculation to the command window.

- (a) Inside the `mycube.m` file:

```
function result = mycube(x)
result = x*x*x;
end
```

- (b) At the command window:

```
mycube(3)
mycube(5)
etc.
```

- (c) Inside the `mycube_script.m` file:

```
cube3 = mycube(3);
cube5 = mycube(5);
```

- (d) Inside the `mycube_script2.m` file:

```
n = input ('Enter n: ');
c = mycube(n);
disp(c)
```

2. A function is defined as

```
function c = blah(a)
```

```

b = a + 3;
c = a * 2;
d = b + c

end

```

- (a) Predict the values of `blah(10)` and of `blah(blah(3))` without using MATLAB

`blah(10)= 20 and blah(blah(3))= 12`

- (b) Re-write the function `blah` so that it does exactly the same thing but does not contain any redundant code.

```

function c = blah(a)
c = a * 2;

end

```

3. Identify problems and errors in the following functions

- (a) `func1`:

```

function f = func1(x, y)
x = 3 * y;
end

```

The output variable `f` is not calculated.

- (b) `func2`:

```

function f = func2(a, b)
f = sqrt(3 * a);
end

```

The input variable `b` is unused.

- (c) `foo`:

```

function [a, b] = foo(x, y)
b = a + x + y;
end

```

Output variable `a` is not defined and, moreover, it is used in another expression before it is defined.

(d) `findSquareRoot`:

```
function findSquareRoot(x)
% Usage : findSquareRoot(x)
%   Provide the square root of the input
returnValue = sqrt(x)
end
```

The square root is calculated but not returned. The definition of the function should be changed to

```
function returnValue = findSquareRoot(x)
```

4. A function takes three numeric arguments representing the coefficients of a quadratic equation and returns two values which are the two roots of the equation. The name of the function is `quadRoots`
  - (a) Write a suitable prototype or definition for the function. Add an appropriate help and usage section at the top of the function and save it in a suitably named file.

For example

```
function [r1, r2] = quadRoots(a, b, c)
% Usage
% [r1,r2] = quadRoots(a,b,c)
% Input: a,b,c, coefficients of quadratic
% Output: r1, r2, roots (if real).
```

- (b) Add a test to check if the quadratic roots are complex and to return with an error if this is true (see Section 3.3, `doc error` or `help error`).

An example could be

:

```
d = b*b - 4*a*c;
```

```

if d < 0
    error('quadRoots: %s', 'Coefficients lead ...
        to complex root(s)')
end

:

```

- (c) Complete the function `quadRoots` and test that it gives the correct results.

```

function [r1, r2] = quadRoots(a, b, c)
% Usage
% [r1,r2] = quadRoots(a,b,c)
% Input: a,b,c, coefficients of quadratic
% Output: r1, r2, roots (if real).

% Discriminant:
d = b*b - 4*a*c;

if d < 0
    error('quadRoots: %s', 'Coefficients lead ...
        to complex root(s)')
end

r1 = (- b + sqrt(d)) / (2 * a);
r2 = (- b - sqrt(d)) / (2 * a);
end

```

- (d) Modify the `quadRoots` function so that it accepts a single array containing all three coefficients and returns a single array containing the roots.

The main parts that change are the definition and a few lines below. The whole function is *not* shown here, dots indicate omitted code (see the previous section).

```

function roots = quadRoots(coeffs)
...
a = coeffs(1);
b = coeffs(2);

```

```

c = coeffs(3);
...
roots(1) = (- b + sqrt(d)) / (2 * a);
roots(2) = (- b - sqrt(d)) / (2 * a);
end

```

5. Write a function that accepts an array of numbers and returns two arrays, the first containing all input numbers that are less than 10 and a second containing all those greater than or equal to 10. Call the function `split_at_ten`. Ensure your code has usage and help comments after the definition. Test it on different arrays to make sure it works.

*(Hint: you can generate random arrays of integers with the built-in `randi` function. Your function will need to return one or more empty arrays if necessary.)*

This example solution uses single line statements to produce each array:

```

function [below, theRest] = split_at_ten(arrayIn)

below = arrayIn(arrayIn < 10);
theRest = arrayIn(arrayIn >= 10);

end

```

6. Write a function to find the number of digits for a given positive integer. The prototype of your function should be

```
function noOfDigits = digit_count(n)
```

*(Hint: a `while` loop might be useful.)*

One possible example:

```

function noOfDigits = digit_count(n)
% Usage digit_count(n)
% Return the number of digits of a given positive ...
% integer.
%
```

```

% Input : n, postive integer given
% Output: noOfDigits, required number of digits

noOfDigits=0;

while n >= 1;
    n = n / 10;
    noOfDigits = noOfDigits + 1;
end;

end

```

7. A triangle is defined by the lengths of its sides. For a triangle to be valid, the sum of the two shorter sides must be greater than the longest side. The code below is a function that accepts a single 3-element array (e.g. [3 4 5]). It returns a value of 0 or 1 to indicate if the three elements of the array can represent the sides of a valid triangle.

```

function result = is_triangle(sides)
% is_triangle(sideLengths):
%     Check if array of 'sideLengths'
%     represents a valid triangle
%
% Input:
%     sideLengths : array of 3 numbers
% Output:
%     result : 1 if sides are valid
%             0 if not valid

longestSide = max(sides);

result = 0;
if longestSide < sum(sides) - longestSide;
    result = 1;
end

```

- (a) Write a line by line explanation of how the function carries out its task.

The call to the built-in `max` function finds the value of the largest number in the array.

The test part begins by assuming that the array does not represent a valid triangle `result = 0`.

The `if` clause compares the longest side with the sum of all the sides excluding the longest side (i.e. the sum of the remaining sides) and, if it is strictly less than the remaining sum, it changes the result to 1.

- (b) Type the code into an *m*-file and save it with the correct name. Call the function from the command window to check it gives the correct result for triples of numbers where you know what the output should be.

Example tests:

```
>> r = is_triangle([3 4 7])
r =
    0

>> r = is_triangle([3 4 10])
r =
    0

>> r = is_triangle([3 4 5])
r =
    1
```

- (c) Modify the function so that it does not use the `max` function but uses the `sort` function instead. Type `help sort` or `doc sort` at the command line to see what it does.

There is more than one way to do it. One possibility is illustrated below for the relevant section of the code:

```
function result = is_triangle(sideLengths)
...
% Sort sides into ascending order
```

```

sideLengths = sort(sideLengths);

% Test
result = sideLengths(3) < sideLengths(1) + ...
    sideLengths(2);

```

- (d) Add some code to the beginning of the function to make sure that the user has provided an array with the correct number of elements. If this is not the case, the function should return with an error message. Type `help error` or `doc error` at the command line to learn about this built-in function.

One way to use the `error` function is by putting the code below at the top of the function

```

...
% Check valid array
if numel(sides) ~= 3
    error('is_triangle: %s', 'Input sides must ...'
        'be a length three array');
end
...

```

8. Write a **recursive** MATLAB function to evaluate a polynomial with a given set of coefficients for a given  $x$  value. Use the description given in Example 3.10 to help you. The definition of your function should be

```
function y = my_poly_value(coeffs, x)
```

Make sure the code is well commented and that there is a help section at the top. Note that this function needs to carry the  $x$  value as an argument which makes it different from the more mathematical treatment in the notes.

One possible solution:

```
function y = my_poly_value(coeffs, x)
% my_poly_value(coeffs, x)
% Evaluate the polynomial defined by the coefficients
```

```

% at a particular x-value
% Inputs:
% coeffs Numeric array of coefficients
% x Required x value
% Output:
% y Result of evaluation.

if length(coeffs) > 1
    y = x * my_poly_value(coeffs(1:end-1), x) + ...
        coeffs(end);
else
    y = coeffs(1);
end

```

Use your function to find the value of  $y$  when  $x = -2$  for the polynomial  $x^4 - 2x^3 - 2x^2 - x + 3$ . Check that your function agrees with the built-in function `polyval`. Type `doc polyval` for details of how to use this function.

$$y = 29$$

9. The Fibonacci sequence is 1, 1, 2, 3, 5, 8, .... It is defined by the starting values  $f_1 = 1$ ,  $f_2 = 1$  and after that by the relation  $f_n = f_{n-1} + f_{n-2}$ .

Write a recursive function to find the  $n^{\text{th}}$  term of the Fibonacci sequence. Use the following definition and ensure the code has a help section and good comments.

```
function f = fibonacci(n)
```

One possible solution

```

function f = fibonacci(n)
% Usage fibonacci(n)
% Return the nth fibonacci term
%
% Input: n the position of the term in the sequence
% Output: f the value of the term

```

```
if n > 2
    f = fibonacci(n-1) + fibonacci(n-2);
else
    f = 1;
end
```

## 12.4 Chapter 4 - Program Development and Testing

Solutions are shown in framed text.

1. The following code is intended to loop over an array and display each number it contains one at a time. It is quite badly written and needs to be corrected.

```
1 % Display elements of an array one by one.  
2  
3 % Array of values  
4 myArray = [12, -1, 17, 0.5]  
5  
6 for myArray  
7     disp(myArray)  
8 end
```

- (a) Type the code into the MATLAB editor, use the hints from the editor to locate any syntax errors. While there are syntax errors in the code, the MATLAB editor will not allow you to use breakpoints for debugging. Once you have corrected any syntax errors, insert a breakpoint at the start and run the script. Step through the code to identify any further errors and fix them as you go along.

The main thing missing here is any reference to individual elements in the array. We should use a subscript to access individual values if we want to process them one at a time. Here we use a variable *k* to do this

```
for k =1:numel(myArray)  
    disp(myArray(k))  
end
```

The subscript variable *k* loops over the numbers in a list that starts at 1 and goes on until the number of elements in the array, as given by the call to `numel(myArray)`.

The `disp` statement now displays an individual element, `myArray(k)`, whereas the original code displayed the whole

array.

- (b) Identify and fix any style violations in the code, i.e. things that do not prevent the code from running but can be changed to improve the appearance or behaviour of the code.

The first working line in the original code (line 4) assigned an array of numbers to the `myArray` variable without putting a semi-colon (`;`) at the end of the line. This means that when the code is run, the value of the assigned variable is displayed automatically in the command window. The use of a semi-colon suppresses this output, i.e.:

```
% Array of values
myArray = [12, -1, 17, 0.5];
```

2. Type the following code into the MATLAB editor, note that it contains errors.

```
clear, close all

% Get 30 random integers between -10 and 10
vals = randi(21, 1, 30) - 11;

for k = 1:1:numel(vals)
    if (vals(n) > 0)
        sumOfPositives = sumOfPositives + vals(n)
    end
end

disp('The sum of the positive values is:')
disp(sumOfPositives)
```

- (a) Identify and explain what the different parts of this piece of code are trying to do. What seems to be the main aim of the code?

The code begins by generating a set of random integers between 1 and 21 inclusive. Subtracting 11 from each element in the array is carried out by *singleton expansion*, i.e. the 11

at the end of the line is automatically converted into an array containing a repetition of the number 11 that is the same size as the array returned by the call to `randi`.

After this subtraction, the resulting array will have random integers between -10 and 10 inclusive. This is stored in the array variable `vals`.

The loop after this appears to have the aim of iterating over each of the elements stored in the array `vals` and to inspect them one at a time. If the element is positive, the value is accumulated and added to a sum that is stored in a variable called `sumOfPositives`. I.e. the aim of the code appears to be to find the sum of the positive elements in the array.

- (b) Find the errors and style violations in the code and fix them using the debugger.

In this code the errors were:

- The variable `sumOfPositives` needed to be set to an initial value of zero before the loop. If this was not done, then it would be used on the right hand side of an assignment in the loop before it was defined.
- The subscript index inside the loop was `n` but the variable actually used in the loop is `k` and these need to match. Changing the `n` to a `k` fixes this.
- A style violation was the missing semi-colon after the central line in the loop. Without this, the output would be printed to the command window for every iteration of the loop.

The corrected code should look like this:

```
clear, close all

% Get 30 random integers between -10 and 10
vals = randi(21, 1, 30) - 11;

sumOfPositives = 0;
```

```

for k = 1:1:numel(vals)
    if (vals(k) > 0)
        sumOfPositives = sumOfPositives + vals(k);
    end
end

disp('The sum of the positive values is:')
disp(sumOfPositives)

```

3. A function with errors in it is given below. It aims to evaluate a quadratic polynomial for a given  $x$  value or for an array of given  $x$  values. Type this function into a file and save it with the correct name.

```

function [ ys ] = evaluate_quadratic(coeffs, xs)
% Usage: evaluate_quadratic(coeffs, xs)
%
% Find the result of calculating a quadratic
% polynomial with coefficients given at the x
% value(s) given. Return the result in the
% variable ys which must contain the same
% number of elements as xs.

clear
close all

ys = zeros(1, xs);
ys = a * xs^2 + b * xs + c;

end

```

Now type the following into a script, and call it *testScript.m*. This will be used to test the function above.

```

% Test script for evaluate_quadratic.m
clear, close all

x = 3;
coeffs = [1 2 1];
y = evaluate_quadratic(coeffs, x);
fprintf('f(%0.1f) = %0.1f\n', x, y)

xs = -1:0.5:3;

```

```
ys = evaluate_quadratic(coeffs, xs);  
disp(xs), disp(ys)
```

- (a) Try to run the script with the debugger, placing breakpoints in the main function. List the errors in the function and suggest how they should be fixed.

There are a number of errors in this code.

- There are calls to clear variables and close windows at the start of the function. These might be suitable for script files but when used in a function have a damaging effect. In particular, the call to `clear` will clear all variables from the workspace and therefore the values passed in as arguments (`coeffs` and `x`) will no longer be available to the function. All references to them will fail. The calls to `clear` and `close all` should not be used in functions and should be removed.
- There is a line where the output variable `ys` appears to be initialised: `ys = zeros(1, xs);` and this is immediately followed by an assignment to the same variable `ys`. This means that the initialisation statement above has no effect and should be removed.
- The same line above containing the call to the built-in function `zeros` can also cause a run-time error if the variable `xs` is an array. The `zeros` function called in the style of

```
arr = zeros(m, n)
```

returns an array that is of size  $m \times n$ . This means that both `m` and `n` in the call need to be integers and passing an array for `xs` above will lead to an error at run time.

- The coefficients are passed to the function as a single array variable `coeffs`. The line that actually does the evaluation assumes that they are in three separate numeric variables `a`, `b` and `c`. This means that they would be undefined here unless they were first extracted from the input argument `coeffs`, for example by

```

a = coeffs(1);
b = coeffs(2);
c = coeffs(3);

```

- Element-wise versus matrix multiplication: The line that carries out the actual evaluation contains a term with the expression `xs^2`. This will be carried out as a matrix squaring which, if the variable `xs` is a 1-D array will lead to a run-time error. The correct behaviour here is to carry out the squaring of each element separately in `xs`, i.e. *element-wise* squaring, which can be written as `xs ... .^ 2.`

The corrected function could be written as:

```

function [ ys ] = ...
    evaluate_quadratic(coeffs, xs)
    % Usage and help here ...

a = coeffs(1);
b = coeffs(2);
c = coeffs(3);

ys = a * xs.^2 + b * xs + c;

end

```

- (b) Add code to the corrected function to validate the input arguments `coeffs` and `xs` to make sure that they can be correctly processed by the function.

- The variable `coeffs` should be an array variable and it should have exactly three elements, corresponding to a quadratic polynomial. We can check for this by using the following code

```

if (numel(coeffs) ~= 3)
    error('evaluate_quadratic: Not 3 ...
        coefficients')
end

```

- The variable `xs` should be a number or a 1-D vector. A single number can be viewed as a length one vector so it can be tested in the same way. In any event, the input must be numeric. We can check this as follows:

```

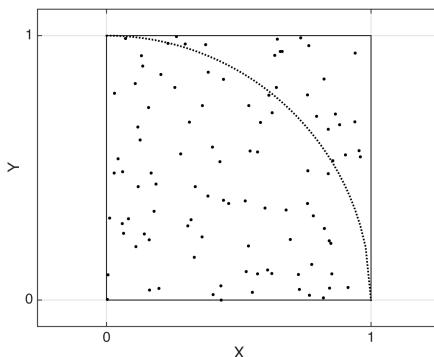
if (~isvector(xs))
    error('evaluate_quadratic: xs not a ...
        vector')
end

if (~isnumeric(xs))
    error('evaluate_quadratic: xs not numeric')
end

```

Make sure you understand how these tests work. They use the built-in functions `isvector` and `isnumeric` which will be discussed in more detail in Section 5.

- This question is about using a method called Monte Carlo simulation to estimate the value of  $\pi$ . The method uses random points chosen inside the unit square in the 2D Cartesian axes, i.e  $0 \leq x \leq 1$  and  $0 \leq y \leq 1$ . This is illustrated in the figure below.



The method picks random points inside the unit square, these are shown by dots in the diagram. Some of these points will lie inside the unit *circle* (which is also plotted) and some of the points will lie outside the unit circle.

We can estimate the area of the quarter circle by finding the fraction

of points that are inside it. From this, we can estimate the area of the whole circle and therefore the value of  $\pi$ .

Write MATLAB code to do this. You should break the problem down into a number of sub-tasks and use different functions to carry each sub-task, using an incremental development approach. Use a main function or main script to control the flow of the program. The user should be able to choose how many points are taken and the program should print a suitable output message.

Run your program and experiment with different numbers of points to see the effect on the estimate of  $\pi$ .

### Solution

In the main script file *monte\_carlo.m*

```
% The main script for using Monte Carlo ...
% simulation to estimate the area of
% a unit circle and therefore for the value of Pi.

N = getNumberOfPoints();

[xPts, yPts] = getRandomUnitSquarePoints(N);

% The count inside will be inside the quarter ...
% circle in the positive
% quadrant.
M = countPointsInsideUnitCircle(xPts, yPts);

% Estimate area of unit circle
A = 4 * M / N;

fprintf('The estimated area for the unit circle ...
is %0.3f\n', A);

% This shows the structure of function calls for ...
% this example
% monte_carlo
% |
% |_ getNumberOfPoints
% |
% |_ getRandomUnitSquarePoints
```

```
% |
% |_ countPointsInsideUnitCircle
```

In the file *getNumberOfPoints.m*

```
function N = getNumberOfPoints()
% Usage:
% Ask the user how many points they want for the ...
% Monte Carlo estimation
% Output: N The number given

disp('How many points do you want to use for the ...
Monte Carlo estimation?')
N = input('Type your number: ');

if N < 1
    error('Need at least 1 point')
end

if N ~= round(N)
    error('Need an integer number of points')
end

end
```

In the function m-file *getRandomUnitSquarePoints.m*

```
function [xs, ys] = getRandomUnitSquarePoints(N)
%
% Provide a number of points randomly chosen ...
% inside the unit square, i.e.
% for 0 <= x <= 1 and 0 <= y <= 1.
% Usage:
% Input: N, the number of points required.
% Outputs: xs, ys, two arrays of size Nx1 ...
% containing the coordinates of the
% points.

xs = rand(N,1);
ys = rand(N,1);

end
```

In the function m-file *countPointsInsideUnitCircle.m*

```

function countInside = ...
    countPointsInsideUnitCircle(xs, ys)
%
% Find the number of points inside the unit circle
% Usage:
% Inputs: xs, ys, the x and y coordinates of the ...
%         points.
% Output: countInside, the number of points ...
%         inside the circle whose centre is the
%         origin and whose radius is 1.

% Initialise the count
countInside = 0;

% How many points altogether.
countAll = numel(xs);

for i = 1:countAll
    x = xs(i);
    y = ys(i);
    r = x^2 + y^2;

    if r <= 1
        countInside = countInside + 1;
    end
end

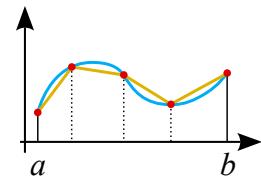
end

```

5. The trapezium rule can be used to estimate the integral of a function between a pair of limits. The formula for the trapezium rule is

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} (f(x_1) + 2f(x_2) + 2f(x_3) + \dots + 2f(x_{n-1}) + f(x_n))$$

where  $f$  is the function to integrate,  $a$  and  $b$  are the limits of the integration and  $\{x_1, x_2, \dots, x_{n-1}, x_n\}$  are a sequence of points on the  $x$ -axis with  $x_1 = a$  and  $x_n = b$ .  $\Delta x$  is the width of the interval between successive points.



Wikimedia

Write MATLAB code that uses the trapezium rule to estimate the integral of  $f(x) = \sin x$  between a pair of limits and for a given number of points. The user should be able to decide the limits and the number of points, for example they might choose ten points between  $x = 0$  and  $x = \pi/4$ . Use an incremental development approach when writing your solution.

### Solution

In the main script file *trapezium.m*

```
% This is the main script that calls other ...
% functions to carry out trapezium
% rule integration estimation.

N = getNumberOfPoints();

[a, b] = getLimits();

trapEst = estimate_integral(a, b, N);

displayResults(a, b, N, trapEst);

% The diagram below shows how the functions are ...
% called

% trapezium
% |
% |__ getNumberOfPoints
% |
% |__ getLimits
% |
% |__ estimate_integral
% |   |
% |   |__ function_to_integrate
% |
% |__ displayResults
```

In the file *getNumberOfPoints.m*

```
function N = getNumberOfPoints()
% Usage:
% Ask the user how many points they want for the ...
% trapezium rule
% Output: N The number given
```

```

disp('How many points do you want to use for the ...')
    trapezium rule?')
N = input('Type your number: ');

if N < 2
    error('Need at least 2 points')
end

if N ~= round(N)
    error('Need an integer number of points')
end

end

```

In the function file *getLimits.m*

```

function [a,b] = getLimits()
% Usage:
% Input: No input arguments
% Output: Two values for the limits of the ...
    trapezium integral estimate.

fprintf('Please give the lower and upper limits ...
        for the integral\n');

a = input('Lower limit: ');
b = input('Upper limit: ');

if a >= b
    error('Lower limit should be less than upper ...
        limit')
end

end

```

In the function m-file *estimate\_integral.m*

```

function I = estimate_integral(a, b, N)
% Usage:
% Input arguments:
% a,b : Limits for the integral

```

```

% N    : Number of points to use in the trapezium ...
        estimate
% Output arguments:
% I    : the value of the trapezium integral estimate.

% Initialise I

I = 0;

% The points that will be used.
xValues = linspace(a, b, N);

% The step size between successive points
deltaX = xValues(2) - xValues(1);

% First value
I = function_to_integrate(xValues(1));

% Intermediate values
for m = 2:N-1
    I = I + 2 * function_to_integrate(xValues(m));
end

% Final value
I = I + function_to_integrate(xValues(N));

% Multiply to get final result
I = I * deltaX / 2;

end

```

In function m-file *function\_to\_integrate.m*

```

function val = function_to_integrate(x)
% Usage:
% Input: An x value
% Output: The value of the function that is being ...
        integrated by the
% trapezium rule.
%
% This is 'hard coded' to the sin function. If a ...
        different function is
% required, the script should be edited.

```

```

val = sin(x);

end

```

In the function m-file *displayResults.m*

```

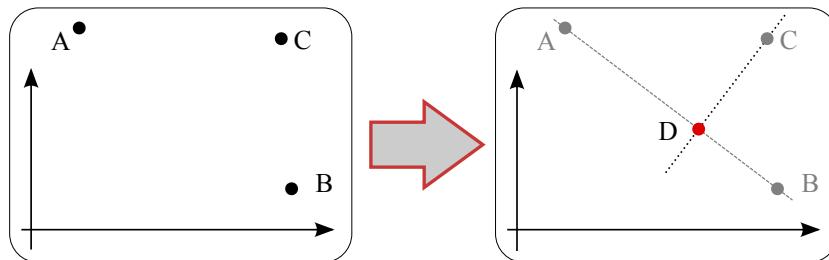
function displayResults(a, b, N, estVal)
% Usage: Show results for estimated trapezium ...
    integration
% Input:
% a, b : Integral limits
% N     : Number of points
% estVal : Estimated value.
% No output arguments.

fprintf('Results of trapezium integration ...
    estimate\n')
fprintf('Limits          : %0.2f %0.2f \n', a, b)
fprintf('Number of points : %u \n', N)
fprintf('Integral estimate : %0.3f\n', estVal)

end

```

6. A high school geometry problem gives the coordinates of three points, A, B, C in the Cartesian plane. The task is to find a point D on the line AB such that AD is perpendicular to BC. This is illustrated in the figure below.



Write MATLAB code that asks the user for the coordinates of the points A, B, C, and calculates the coordinates of point D.

Break the problem down into sub-tasks and write separate functions for each, using an incremental development approach. Write a single main function or script that controls the flow of the code.

## Possible solution

In the main script *geometry.m*

```
% Main script for running the geometry exercise ...
% given in the manual.
% Given three points, A,B,C, find a fourth point ...
% D on the line AB such that
% AB and CD are perpendicular.

% Get points
[xA, yA] = getPoint('Give coordinates of point A');
[xB, yB] = getPoint('Give coordinates of point B');
[xC, yC] = getPoint('Give coordinates of point C');

% Get the gradient and intercept for line AB
[mAB, cAB] = lineFromTwoPoints(xA, yA, xB, yB);

% Get the gradient for line CD which is ...
% perpendicular to the line AB.
mCD = -1 / mAB;

% Get the gradient and intercept of line CD
[mCD, cCD] = lineFromGradAndPoint(mCD, xC, yC);

% Get the intersection of lines AB and CD
[xD, yD] = intersectionOfTwoLines(mAB, cAB, mCD, ...
cCD);

fprintf('The coordinates of point D are :(%0.1f, ...
%0.1f)\n', xD, yD)

% This diagram shows the structure of function ...
% calls in this example
%
% geometry
%
%   |
%   |__ getPoint
%   |
%   |__ lineFromTwoPoints
%       |
```

```
% |     |_ gradFromTwoPoints
%
% |_ lineFromGradAndPoint
%
% |_ intersectionOfTwoLines
```

In the function m-file *lineFromTwoPoints.m*

```
function [m,c] = lineFromTwoPoints(x1, y1, x2, y2)
% Provide a representation for a line that goes ...
% through two points. The
% line is represented by its gradient (m) and ...
% intercept (c), i.e. the
% equation of the line is 'y = mx + c'.
%
% Usage:
% Input:
% x1,y1 : coordinates of first point
% x2,y2 : coordinates of second point
%
% Output:
% m      : gradient of line between points
% c      : intercept of line on the y axis

% Get the gradient
m = gradFromTwoPoints(x1, y1, x2, y2);

% Use one of the points to find the constant.
c = y1 - m * x1;

end
```

In the function m-file *lineFromGradAndPoint.m*

```
function [m, c] = lineFromGradAndPoint(m, x, y)
% Give a representation of the equation of a line ...
% that goes through a point
% with a given gradient. The line is represented ...
% by its gradient (m) and
% intercept (c), i.e. the equation of the line is ...
% 'y = mx + c'.
%
% Usage:
% Input :
% m :gradient of line
```

```

% x,y : coordinates of point
% Output:
% m : gradient of line (same as input!)
% c : Intercept of line on y-axis.

c = y - m * x;

end

```

In the function m-file *intersectionOfTwoLines.m*

```

function [x,y] = intersectionOfTwoLines(m1, c1, ...
m2, c2)
%
% Find the intersection of two straight lines in ...
% the 2-D plane. The lines
% are represented by 'y = m1 x + c1' and 'y = m2 ...
% x + c2'
%
% Usage:
% Input:
% m1, c1 : Gradient and intercept of one line.
% m2, c2 : Gradient and intercept of second line.
% Output:
% x,y : Intersection of the lines.

```

```

% Check if lines are parallel.
if m1 == m2
    error('intersectionOfTwoLines: Lines are ...
parallel!');
end

% Solve for x in the equation 'm1 x + c1 = m2 x ...
+ c2'
x = (c2 - c1) / (m1 - m2);

% Find y, base on one of the lines.
y = m1 * x + c1;

end

```

In the function m-file *gradFromTwoPoints.m*

```

function m = gradFromTwoPoints(x1, y1, x2, y2)
    % Provide the gradient of the line joining two ...
    % points in 2-D.
    % Raises an error if the gradient is infinite
    %
    % Usage:
    %   x1,y1 : Coordinates of one point
    %   x2,y2 : Coordinates of second point

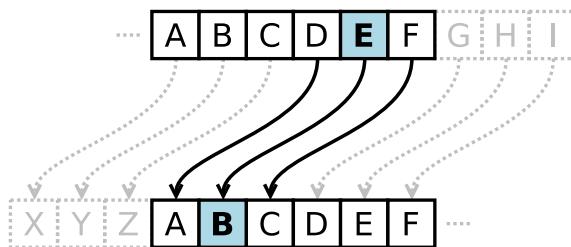
    if x2 == x1
        error('gradFromTwoPoints: Gradient is infinite!');
    end

    m = (y2 - y1) / (x2 - x1);

end

```

7. A shift cipher is a simple way to make encrypted messages. It works by replacing each letter in a message with another letter that is shifted by a fixed amount in the alphabet (forwards or backwards). This is illustrated below for a backward shift of three.



*Wikimedia*

With this shift, for example, the word EAT is encrypted to become BXQ. If we want to reverse the encryption (decrypt), we can just apply the same encryption process but use a shift in the opposite direction, i.e. a shift of forward three for the BXQ example.

Write MATLAB code that takes a message and encrypts it with a given shift. The user can choose the message and the required shift. You can use the convention that a forward shift is a positive number and a backward one is a negative number. As an example that you can use

to check, the message HELLO WORLD encrypted with a shift of +7 is encrypted to become OLSSV DVYSK. Include code to check that your encrypted message can be decrypted correctly afterwards. You should use an incremental development approach when writing your solution.

### Possible solution

In main script m-file *cipher.m*

```
% Main script to run the functions for encrypting ...
% and decrypting a message.

% The message
mesg = input('Please type your message ', 's');

% How far to shift when encrypting.
shift = input('What shift is needed? ');

encryptedMesg = encryptString(mesg, shift);

fprintf('Original message: %s \n', mesg)
fprintf('Encrypted message: %s \n', encryptedMesg)

% Decrypting can be done with the same shift in ...
% the opposite direction.
decrypted = encryptString(encryptedMesg, -1 * shift);
fprintf('Decrypted again : %s \n', decrypted)

% This diagram shows the structure of function ...
% calls in this example:
% cipher
%   |
%   |_ encryptString
%       |
%       |_ isLetter
%           |
%           |_ encryptChar
%               |
%               |_ charToNumber
%                   |
%                   |_ numberToChar
```

In function m-file *encryptString.m*

```
function outputString = ...
    encryptString(inputString, shift)
% Take a string and s shift to apply and encrypt ...
    the string with it
%
% Usage:
% Inputs:
% inputString: the string to be encrypted
% shift      : The shift to use for the code
% Output
% outputString: The encrypted version of the input.

% How many letters?
nLetters = numel(inputString);

% Initialise the output by copying the input.
outputString = inputString;

% Loop over the string
for n = 1:nLetters

    % Take a character
    inputChar = inputString(n);

    % Only encrypt it if it is a letter, otherwise ...
    % copy it.
    if isLetter(inputChar)
        codeChar = encryptChar(inputChar, shift);
    else
        codeChar = inputChar;
    end

    % Assign to output.
    outputString(n) = codeChar;

end

end

%%

function result = isLetter(c)
```

```

% A helper function also inside the file ...
% 'encryptString.m' that tests
% if a given character is a letter. Returns true ...
% or false.

result = false;

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

for i = 1:26
    if upper(c) == alphabet(i);
        result = true;
        break;
    end
end

end

```

In function m-file *encryptChar.m*

```

function charOut = encryptChar(charIn, shift)
% Encrypt a single character by shifting along
% the alphabet, e.g. to % encrypt 'C' with a
% shift of +3, the return value will be 'F'.
%
% Further helper functions are included below
% within this file.
%
% Usage:
% Inputs:
% charIn : character to encrypt
% shift : shift to apply
% Output:
% charOut : the encrypted character.

% Convert to upper case
charIn = upper(charIn);

% Get position in alphabet
n = charToNumber(charIn);

% Apply shift.
n = n + shift;

```

```

% If the shift is positive and makes the result ...
% higher than 26, subtract
% until we are in the range 1-26
while n > 26
    n = n - 26;
end

% If the shift is negative and makes the result ...
% less than 1, then add until
% we are in the range 1-26
while n < 1
    n = n + 26;
end

% Get letter at resulting position.
charOut = numberToChar(n);

end

%%

function n = charToNumber(c)
% Helper function, returns the position of a ...
% letter in the alphabet.

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

n = -1;

for i = 1:26
    if upper(c) == alphabet(i);
        n = i;
        break;
    end
end

if n == -1
    error('encryptChar: charToNumber: Not a ...
        character in the alphabet');
end

```

```
end

%%

function c = numberToChar(n)
% Helper function, returns the letter in a given ...
    position in the alphabet.

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

if n < 1 || n > 26
    error('encryptChar: numberToChar: Number out of ...
        range');
end

c = alphabet(n);

end
```

## 12.5 Chapter 5 - Data Types

Solutions are shown in framed text.

1. (a) Initialise your own  $1 \times 3$  array of numbers. Try and choose them so there is at least one integer that matches the ASCII code for an alphanumeric character.

E.g.

```
>> a = [100 12 70]  
  
a =  
  
100      12      70
```

- (b) Make an explicit conversion of the array to a character array. If some of the characters appear to be missing, why should this occur?

In the above case.

```
>> char(a)  
  
ans =  
  
d F
```

In this example, the number 12 does not correspond to a recognisable character.

- (c) Make an explicit conversion of the array to the logical type.

```
>> logical(a)  
  
ans =  
  
1      1      1
```

- (d) Make an *implicit* conversion of the array's contents to a character.

(Hint: we can use concatenation).

```
>> b = 'xyz';
>> [a b]
ans =
d Fxyz
```

2. (a) Make up a one row character array containing 'some string' using square brackets and individual characters

E.g.

```
>> s = ['s' 'o' 'm' 'e' ' ' 's' 't' 'r' 'i' ...
'n' 'g']

s =
some string
```

or, if we want to use commas:

```
>> s = ['s', 'o', 'm', 'e', ' ', 's', 't', ...
'r', 'i', 'n', 'g']

s =
some string
```

- (b) Make the same character array using single quotes:

```
>> s = 'some string';
```

- (c) Make a two row character array containing 'some' and 'string' on separate rows. What do we need to do to make sure these two words can be fitted into a two row array?

We need to add extra spaces to the shorter word so that it matches the length of the longer one.

```
>> s = ['some' ; 'string']

s =

some
string
```

- (d) Make an explicit conversion of the character array of the last part to a numeric type.

```
>> double(s)

ans =

115    111    109    101     32     32
115    116    114    105    110    103
```

- (e) Make an implicit conversion of the character array to a numeric type.

We can do this, for example, by simply adding 10 to the array.

```
>> s + 10

ans =

125    121    119    111     42     42
125    126    124    115    120    113
```

- (f) For each of the following character arrays, try and generate a corresponding numeric value using the `str2double` function. Give the reason why some of these conversions can fail and how we can test for it.

```
'2.3'
'e'
'0.7'
'X.3'
'1.3e+02'
```

```

>> str2double('2.3')
ans =
    2.3000

>> str2double('e')
ans =
    NaN

>> str2double('0.7')
ans =
    0.7000

>> str2double('X.3')
ans =
    NaN

>> str2double('1.3e+02')
ans =
    130

```

The second and third fail because the character arrays cannot be interpreted as a number by MATLAB (even though one of them should probably be!).

We can check if the return value of a `str2double` call is a `NaN` value, e.g. by using the `isnan` function. E.g.

```

if ( isnan(someConvertedString) )
    % do something here.
end

```

- (g) Make an array of string types containing the strings '`'Baa'`', '`'Black'`' and '`'Sheep'`' on separate rows. Write MATLAB code using a `for` loop to create a new string array which is a copy of the first array but with all characters converted to lower case.

Now we use double quotes and we do not need to pad:

```

s=[ "Baa"; "Baa"; "Black"; "Sheep"]
for r=1:length(s)
    s_lower(r) = lower(s(r));

```

```

end
s_lower

```

3. Write a MATLAB *m*-file to define the matrices  $A$ ,  $B$  and  $C$  as follows:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 3 & 2 & 1 \\ 3 & 2 & 4 \end{pmatrix}, B = \begin{pmatrix} 4 & 4 & 4 \\ 2 & 2 & 2 \\ 1 & 1 & 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 0 & 1 \\ 2 & 1 & 3 \\ 1 & 0 & 1 \end{pmatrix}.$$

Now modify the script to compute the following:

- (a)  $D = (AB)C$
- (b)  $E = A(BC)$
- (c)  $F = (A + B)C$
- (d)  $G = A + BC$

Save all variables to a MATLAB *MAT* file.

```

A=[1 2 1; 3 2 1; 3 2 4];
B=[4 4 4; 2 2 2; 1 1 1];
C=[1 0 1; 2 1 3; 1 0 1];
D=(A*B)*C
E=A*(B*C)
F=(A+B)*C
G=A+B*C
save ('data.mat', 'A', 'B', 'C', 'D', 'E', 'F', 'G');

```

4. Systems of linear equations such as

$$\begin{aligned} 5x_1 + x_2 &= 5, \\ 6x_1 + 3x_2 &= 9, \end{aligned}$$

can be solved by expressing the equations in matrix form, i.e.

$$\begin{pmatrix} 5 & 1 \\ 6 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \end{pmatrix}$$

and then rearranging to solve for  $x_1$  and  $x_2$ :

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 & 1 \\ 6 & 3 \end{pmatrix}^{-1} \begin{pmatrix} 5 \\ 9 \end{pmatrix}$$

Write a MATLAB *m*-file that uses matrices to solve the system of equations given above.

```
a = [5 1; 6 3];
y = [5; 9];
inv(a)*y
```

5. The `intmax` function can be used to show that the largest unsigned sixteen bit integer is 65535. Show the numeric calculations that prove this is the largest possible unsigned sixteen bit integer.

The long way: There are 16 bytes which correspond to the powers of two

$$2^{15}, 2^{14}, \dots, 2^1, 2^0$$

The binary pattern consisting of sixteen 1's :

$$1111111111111111_{bin}$$

corresponds in decimal to the sum of these powers of two

$$2^{15} + 2^{14} + \dots + 2^1 + 2^0 = 65535$$

The short way: The sum  $2^{15} + 2^{14} + \dots + 2^1 + 2^0$  equals

$$2^{16} - 1 = 65535$$

6. A set of numeric variables are defined as follows:

```
a = pi
b = uint8(250)
c = int32(100000)
d = single(17.32)
```

(a) Give the full name of the type for each of the variables above.

- a: Double precision float
- b: Unsigned eight bit integer
- c: Signed 32 bit integer
- d: Single precision float

(b) An array is created by concatenating the variables above:

```
X = [ a b c d ]
```

Identify the type of the resulting array

```
class(X) gives uint8 as the type.
```

(c) Create the array by concatenating different subsets of the elements in different orders:

```
X = [a d]
X = [d a]
X = [a d b]
X = [a d c]
X = [a d c b]
X = [a d b c]
```

Check the type of each result each time.

Identify the rule that MATLAB uses for deciding on the type of the output:

- when the array contains one or more integer types.
- when the array contains a mixture of floating point types and no integer types.

If there is an integer type present, the type of the combined array will match the type of the first integer that appears (reading from the left).

If the array only contains a mixture of floating point types,

i.e. double and single, the result will have the single type.

- (d) A fifth variable is assigned a character: `e = 'M'`. What is the data type of the resulting array if variable `e` is included in any of the concatenations above?

The result is always a character array. Some of the numeric variables may not correspond to an ASCII or Unicode character so will appear as blanks or boxes.

7. (a) Make a cell array called `c` that contains the following eleven character arrays

```
'the'  
' '  
'cow'  
' '  
'jumped'  
' '  
'over'  
' '  
'the'  
' '  
'moon'
```

```
c = {'the', ' ', 'cow', ' ', ...  
      'jumped', ' ', 'over', ' ', ...  
      'the', ' ', 'moon'};
```

- (b) Write code at the command line that gives the cell at index 7 in the array `c`. Make sure you use the correct type of brackets. Use the expression `class(ans)` to confirm that a cell is returned.

For this, we need to use round brackets.

```
>> c(7)
```

```
ans =
```

```
    'over'
```

```
>> class(ans)  
ans =  
cell
```

- (c) Write code that gives the cells with odd indices in the array *c*, i.e. the cells at index 1, 3, 5, etc. How many cells are returned?

We can still use round brackets, just need to give a range of indices. We can do this with the expression `1:2:end` which will start at index 1 and go up in steps of 2 until the end of the array.

```
>> c(1:2:end)  
ans =  
'the'  'cow'  'jumped'  'over'  'the'  'moon'
```

Six cells are returned.

- (d) Write code that gives the cells with even indices in the array *c*.

```
>> c(2:2:end)  
ans =  
' '  ' '  ' '  ' '
```

I.e. we get the cells containing the five spaces!

- (e) Write down code to return the character array *contained* in the cell at index 5 of the array *c*

For this we need to use the *curly* brackets.

```
>> c{5}
```

```
ans =  
jumped
```

- (f) Write code that returns the last character of the character array contained in the cell at index 5 in array c

```
>> c{5}(end)  
  
ans =  
  
d
```

Or, if we know that the character array 'jumped' contains 6 characters, then we could use

```
>> c{5}(6)  
  
ans =  
  
d
```

- (g) Write code that creates a new cell array c2 that is a copy of c but with all instances of the letter 'e' removed from words with length greater than 3.

```
for x = 1:length(c)
    if (length(c{x})>3)
        c2{x} = erase(c{x}, 'e');
    else
        c2{x} = c{x};
    end
end
```

- (h) Make a new cell array d that contains at least three different data types.

Many possible answers, e.g.

```
>> x = 3.2; % double  
>> y = true; % logical  
>> z = 'hi!'; % char array  
>> d = {x, y, z} % cell array initialisation  
  
d =  
  
[3.2000] [1] 'hi!'
```

8. Here are the names of five animals: Aardvark, Lion, Lemur, Camel, Elephant.

- (a) By padding the names with spaces where necessary, show how you can construct a 2-D character array to contain all the five names in 1-D character arrays. Assign this to a variable called a.

For example

```
>> a = ['Aardvark'; 'Lion      '; 'Lemur      '; ...  
       'Camel    '; 'Elephant']  
%  
Lion needs four spaces, Lemur and Camel need ...  
three, the remaining two both have eight ...  
characters and don't need padding.
```

- (b) Now show how we can put the five names into a cell array of character arrays. Assign this to a variable called c.

```
>> c = {'Aardvark', 'Lion', 'Lemur', ...  
       'Camel', 'Elephant'}
```

- (c) Write the code that is needed to replace the Lemur with a Dog in the 2-D char array a.

```
% Need to replace the entire third row of the ...
array:
>> a(3,:) = 'dog'      ; % padded with five ...
spaces
```

- (d) Write the code that is needed to replace the Lemur with a Dog in the cell array c.

```
% Note curly bracket to replace an element.
>> c{3} = 'Dog'; % No padding needed.
```

The questions beyond this point are on struct and map data types.

9. A list of animals (vertebrates) and their classes are given below:

| animal | class   |
|--------|---------|
| zebra  | mammal  |
| gecko  | reptile |
| mouse  | mammal  |
| robin  | bird    |
| heron  | bird    |
| cobra  | reptile |
| hyena  | mammal  |

- (a) Show how a map can be constructed so that the name of each animal is a key and its class is the corresponding value. Assign the map to a variable called `classes`.

For example

```

keys = {'zebra', 'gecko', 'mouse', 'robin', ...
        'heron', 'cobra', 'hyena'}
values = {'mammal', 'reptile', 'mammal', ...
          'bird', 'bird', 'reptile', 'mammal'}
classes = containers.Map(keys, values)

```

- (b) Write the code for adding a horse to the map `classes`.

```
classes('horse') = 'mammal'
```

- (c) Write code for iterating over the keys of the map. Hint: assign the keys to a variable and use the built-in function `numel` to find out how many elements it has.

For example:

```

ks = classes.keys;
count = numel(ks);

for i = 1:count
    disp( ks{i} );
end

```

10. Chemical elements can be described by their *name* or their *symbol*. Each element also has an *atomic number* (indicating the number of protons in its nucleus).

- (a) Show how a struct can be created to represent these three pieces of information for an element. Use the italics above for the field names. Give an example for an element of your choice.

More than one way to do this, for example

```

>> myElement = struct
>> myElement.name = 'carbon'

```

```
>> myElement.symbol = 'C'  
>> myElement.atomicNumber = 12
```

or

```
>> myElement = struct('name', 'carbon', ...  
                      'symbol', 'C', 'atomicNumber', 6)
```

- (b) Repeat the previous part for two more elements, and show how you can collect them into a single array of structs.

For example:

```
c = struct('name', 'carbon', 'symbol', 'C', ...  
          'atomicNumber', 6)  
he = struct('name', 'helium', 'symbol', 'He', ...  
           'atomicNumber', 2)  
mg = struct('name', 'magnesium', 'symbol', ...  
           'Mg', 'atomicNumber', 12)  
  
allElts = [he c mg]
```

- (c) Show how we can access

- the name of the second element in the array
- the symbols of the first and last elements in the array

```
- allElts(2).name  
- allElts([ 1 3 ]).symbol
```

11. This question is a continuation of the map one in Example 5.16. Write a function to add a new person with a given age to the map. The function should accept three arguments. The first argument is a map that is assumed to have the correct key and value data types (numeric and cell array of character arrays). The second argument represents the key, consisting of the age of the person to add. The last argument is the name of the person to add. The code will need to check whether the key is already used. A basic structure of the code is given at the

end of Example 5.16.

Test your function by adding new persons with and without ages that are already contained in the map.

For example

```
function addPerson(people, age, name)
%
% Usage : addPerson(mapName, age, name)
%
% Arguments:
% people : A map with age (numeric) as key
% age     : Age of person to include
% name    : Name of person to include
%
% Insert a person with the given name into
% the map using the age as a key. Include
% them with others of the same age if the
% key is already in use.

if ( people.isKey(age) )
    % Add the person to those who are already
    % included in the map.
    people(age) = [ people(age) name ];
else
    % New key, can enter it directly and start
    % a new cell array containing one char array.
    people(age) = { name };
end

return
```

## 12.6 Chapter 6 - File Input/Output

Solutions are shown in framed text.

1. Write a MATLAB *m*-file to create the following variables in your MATLAB workspace, and initialise them as indicated:

- a character variable called *x* with the value ‘a’;
- an array variable called *q* containing 4 floating point numbers: 1.23, 2.34, 3.45 and 4.56;
- a Boolean variable called *flag* with the value *true*.

Now save only the array variable to a *MAT* file, clear the workspace, and load the array in again from the *MAT* file.

```
x = 'a';
q = [1.23 2.34 3.45 4.56];
flag = true;
save('nums.mat', 'q');
clear;
load('nums.mat');
```

2. Write the same 1-D array of floating point numbers that you defined in Exercise 1 to a text file using the *writematrix* function. Then clear the workspace and read the array in again using *readmatrix*.

```
q = [1.23 2.34 3.45 4.56];
writematrix(q, 'nums.txt', 'Delimiter', ';');
clear;
q = readmatrix('nums.txt', 'Delimiter', ';');
```

3. Again, working with the array from Exercise 1, again write it to a text file using the *writematrix* function, but this time use the semicolon character as the delimiter. Then clear the workspace and read the array in again using *readmatrix*.

```
q = [1.23 2.34 3.45 4.56];
writematrix(q, 'nums.txt', 'Delimiter', ';');
```

```
clear;
q = readmatrix('nums.txt', 'Delimiter', ';');
```

4. The file *patient\_data.txt* (available from the KEATS system) contains personal data for a group of patients taking part in a clinical trial. There is a row in the file for each patient, and the five columns represent: patient ID, age, height, weight and heart rate. Write a MATLAB *m*-file that uses the `readmatrix` function to read in this data.

```
% read data
data = readmatrix('patient_data.txt', ...
'Delimiter', ';');
```

5. Continuing with Exercise 4, write code to input a single integer value *n* from the keyboard, and display the height and weight data for the *n* oldest patients.

```
% how many data items to display?
n = input('Enter n: ');
% loop n times and display data for oldest patient
d = data;
for x = 1:n
    % find array index of current oldest patient
    [vals,ind] = max(d(:,2));
    % height/weight of current oldest patient
    disp(['Height = ' num2str(d(ind,3)) ...
        ', weight = ' num2str(d(ind,4))]);
    % remove current oldest patient
    d = [d(1:ind-1, :); d(ind+1:end, :)];
end
```

6. In the KEATS system you will also be able to access a second patient data file called *patient\_data2.txt*. This also contains patient data (patient ID, age, height, weight, heart rate) but for some different patients in the trial. Modify the *m*-file you wrote in Exercise 4 to also read in this data and combine it with the data from the first file. There were

five pieces of data for each of nine patients in the first file and the same data for an additional five patients in this second file, so your final data should be a  $14 \times 5$  array. Notice that this second file has no line break characters so you will not be able to use `readmatrix`, because this uses line breaks to separate the rows of data.

Modified program:

```
% read first data file
data = readmatrix('patient_data.txt', ...
'Delimiter', ';');
% read second data file
fid = fopen('patient_data2.txt', 'r');
data2 = fscanf(fid, '%d;%d;%d;%d;%d;', [5 inf]);
data2 = data2';
fclose(fid);
% how many data items to display?
n = input('Enter n: ');
% combine data into one 2-D array
d = [data; data2];
% loop n times and display data for oldest patient
for x = 1:n
    % find array index of current oldest patient
    [vals,ind] = max(d(:,2));
    % height/weight of current oldest patient
    disp(['Height = ' num2str(d(ind,3)) ...
        ', weight = ' num2str(d(ind,4))]);
    % remove current oldest patient
    d = [d(1:ind-1, :); d(ind+1:end, :)];
end
```

7. The equation for computing a person's body mass index (BMI) is:

$$\text{BMI} = \frac{\text{mass}}{\text{height}^2} \quad (12.1)$$

where the mass is in kilograms and the height is in metres. A BMI of less than 18.5 is classified as underweight, between 18.5 and 25 is normal, more than 25 and less than 30 is overweight, and 30 or over is obese. In the KEATS system you will be provided with two text

files: *bmi\_data1.txt* and *bmi\_data2.txt*. These both contain data about patients' height and weight (preceded by an integer patient ID), but in different formats. Write a MATLAB *m*-file to read in both sets of data, combine them, and then report the patient ID and BMI classification of all patients who do not have a normal BMI.

```
% read first data file
fid = fopen('bmi_data1.txt', 'r');
data1 = fscanf(fid, '%d/%f/%d/', [3 inf]);
data1 = data1';
fclose(fid);
% read second data file
data2 = readmatrix('bmi_data2.txt');
% combine data into one 2-D array
d = [data1; data2];
% loop over all patients
for x = 1:length(d)
    % compute BMI
    bmi = d(x,3) / d(x,2)^2;
    % check if BMI outside normal range
    if (bmi < 18.5)
        disp(['Patient ' num2str(d(x,1)) ...
               ': underweight']);
    elseif (bmi >= 30)
        disp(['Patient ' num2str(d(x,1)) ': obese']);
    elseif (bmi > 25)
        disp(['Patient ' num2str(d(x,1)) ...
               ': overweight']);
    end
end
```

8. When MR scans are performed a log file is typically produced containing supplementary information about the scan's operation. An example of such a file (*logcurrent.log*) is available to you through the KEATS system. One piece of information that this file contains is the date and time at which the scan started. This information is contained in the 2<sup>nd</sup> and 3<sup>rd</sup> words of the line that contains the text "Scan starts". Write a MATLAB *m*-file to display the start date and time for the scan which

produced the *logcurrent.log* file.

(Hint: look at the MATLAB documentation for the *strfind* and *strsplit* functions.)

```
% open file
fid = fopen('logcurrent.log','r');
% get text
str = 'Scan starts';
% loop until end of file
while ~feof(fid)
    % read line
    tline = fgetl(fid);
    % is the text in this line?
    if strfind(tline,str)
        % print 2nd and 3rd words of line
        s = strsplit(tline);
        disp([s{2} ' ' s{3}]);
    end
end
% close file
fclose(fid);
```

9. The file *names.txt* (available through KEATS) contains the first and last names of a group of patients. Write a MATLAB *m*-file to read in this data using the *textscan* function and display the full names of all patients with the last name “Bloggs”.

(Hint: Look at the MATLAB documentation for the *strcmp* command.

```
% read data
fid = fopen('names.txt','r');
names = textscan(fid, '%s %s');

% find Bloggs
for user=1:length(names{1})
    if strcmp(names{2}{user}, 'Bloggs')
        disp([names{1}{user} ' ' names{2}{user}]);
    end
end
```

```
% close file  
fclose(fid);
```

10. The file *bp\_data.txt* (available through KEATS) contains data about patients' systolic blood pressure. There are four fields for each patient: patient ID, first name, last name and systolic blood pressure. Write a MATLAB *m*-file to read in this data, identify any 'at risk' patients (those who have a blood pressure greater than 140) and write only the data for the 'at risk' patients to a new file, called *bp\_data\_at\_risk.txt*.

```
% open input/output files  
fin = fopen('bp_data.txt','r');  
fout = fopen('bp_data_at_risk.txt','w');  
% read data  
data = textscan(fin, '%d %s %s %d', ...  
    'delimiter', '/ ' );  
% find high b.p.  
for user=1:length(data{1})  
    if (data{4}(user)> 140)  
        fprintf(fout, '%d: %s %s %d\n', ...  
            data{1}(user), data{2}{user}, ...  
            data{3}{user}, data{4}(user));  
    end  
end  
% close files  
fclose(fin);  
fclose(fout);
```

11. Whenever a patient is scanned in an MR scanner, as well as the log file mentioned in Exercise 8, the image data is saved in files that can then be exported for subsequent processing. One format for this image data, which is used on Philips MR scanners, is the PAR/REC format. With PAR/REC, the image data itself is stored in the REC file whilst the PAR file is a text file that contains scan sequence details which indicate how the REC file data should be interpreted. An example of a real PAR file (*patient\_scan.par*) is available to you through the

KEATS system. Write a MATLAB *m*-file to extract from the PAR file and display the following information: the version of the scanner software (which is included at the end of the 8<sup>th</sup> line), the patient name and the scan resolution.

```
% open input file
fid = fopen('patient_scan.par', 'r');
% software version
data = textscan (fid, '%*s %*s %*s %*s %*s ... ...
%*s %s', 1, 'headerlines', 7);
version = data{1}{1}
% patient name
data = textscan (fid, '%*s %*s %*s %*s %s %s', ...
1, 'headerlines', 4);
name = [data{1}{1} data{2}{1}]
% scan resolution
data = textscan (fid, '%*s %*s %*s %*s %*s %*s ... ...
%d %d', 1, 'headerlines', 16);
resolution = [data{1} data{2}]
% close file
fclose(fid);
```

12. The file *cholesterol.bin* (available through KEATS) is a binary file. The first data element in the file is an 8-bit integer specifying how many records the following data contains. Each subsequent record consists of an 8-bit integer representing a patient's age followed by a 16-bit integer representing the same patient's cholesterol level. Write a MATLAB *m*-file to read the age/cholesterol data into two array variables.

```
% open file
fid = fopen('cholesterol.bin', 'r');
% read number of records
n = fread(fid, 1, 'int8');
% preallocate arrays for efficiency reasons
ages = zeros(n, 1);
chol = zeros(n, 1);
% loop over number of records
```

```

for x=1:n
    ages(x)= fread(fid, 1, 'int8');
    chol(x)= fread(fid, 1, 'int16');
end
% close file
fclose(fid);

```

13. Measuring blood sugar levels is an important part of diabetes diagnosis and management. Ten patients have had their post prandial blood sugar level measured and the results were 5.86, 8.71, 4.83, 7.05, 8.25, 7.87, 7.14, 6.83, 6.38 and 5.77 mmol/L. Write a MATLAB *m*-file to write this data to a binary file using an appropriate data type and precision. Your program should then clear the MATLAB workspace and read in the data again.

```

% write data
fid = fopen('sugar.bin', 'w');
sugar = [5.86, 8.71, 4.83, 7.05, 8.25, ...
          7.87, 7.14, 6.83, 6.38, 5.77];
fwrite(fid, sugar, 'float32');
fclose(fid);
% clear workspace and read in again
clear
fid = fopen('sugar.bin', 'r');
sugar = fread(fid, inf, 'float32');
disp(sugar);
fclose(fid);

```

14. Write a MATLAB *m*-file to read in a text file and write out a new text file that is identical to the input file except that all upper case letters have been converted to lower case.

```

% open files
fid_in = fopen('text.txt', 'r');
fid_out = fopen('text_lc.txt', 'w');
% loop until end of input file

```

```

while ~feof(fid_in)
    % read line & convert to lower case
    tline = fgetl(fid_in);
    tline = lower(tline);
    % write to output file
    fprintf(fid_out, '%s\n', tline);
end
% close files
fclose(fid_in);
fclose(fid_out);

```

15. Write a MATLAB *m*-file to read in another *m*-file, strip away any blank lines or comments, and write the remaining statements to a new *m*-file. Note that a comment is any line in which the first non-white space character is a %.

*(Hint: the MATLAB command `strtrim` can be used to remove leading and trailing white space from a character array.)*

```

% open files
fid_in = fopen('ex6.m', 'r');
fid_out = fopen('ex6_nocomments.m', 'w');
% loop until end of input file
while ~feof(fid_in)
    % read line & remove leading white space
    tline = fgetl(fid_in);
    tline = strtrim(tline);
    % is it blank or a comment?
    if isempty(tline) || strncmp(tline, '%', 1)
        continue;
    end
    % write non-comment line to output file
    fprintf(fid_out, '%s\n', tline);
end
% close files
fclose(fid_in);
fclose(fid_out);

```

## 12.7 Chapter 7 - Program Design

Solution files for exercises 5-9 will be provided through the KEATS system.

Solutions are shown in framed text.

1. Write a MATLAB implementation of the perfect number top-down design given in Example 7.1. You will need to consider whether to implement each module using a sequence of statements in the main script *m*-file or as a separate MATLAB function. Use an incremental development approach and test stubs when developing your solution.

```
ex1_perfect.m:  
% script to read integer and check if it is perfect  
% (i.e. equal to sum of its divisors)  
% I. get number  
n = input('Enter positive integer: ');  
if n < 0  
    error('Must enter positive integer');  
end  
% II. is number perfect?  
yes_no = is_perfect(n);  
% III. display answer  
if (yes_no)  
    fprintf('%d is perfect\n', n);  
else  
    fprintf('%d isn''t perfect\n', n);  
end  
  
is_perfect.m:  
function answer = is_perfect(num)  
% Usage:  
% answer = is_perfect(num)  
% num : Number to be tested  
% answer : Output, Boolean value – is num perfect?  
% sum divisors  
    divisor_sum = 1;
```

```

d = 2;
while d <= (num/2)
    if (mod(num, d)== 0)
        divisor_sum = divisor_sum + d;
    end
    d = d + 1;
end
% is sum = n?
if (divisor_sum == num)
    answer = true;
else
    answer = false;
end
end %function

```

2. Modify your solution to Exercise 1 so that your program reads multiple numbers from the user, stopping only when they enter a negative number. (This corresponds to the structure chart shown in Figure 7.2.) When a negative number is entered the program should exit with no error message.

```

ex2_perfect.m:
% read integers and check if they are perfect
% (i.e. equal to sum of their divisors)
% loop multiple times
n = 0; %so that code enters loop first time
while n >= 0
    % I. get number
    n = input('Enter positive integer: ');
    if n < 0
        break;
    end
    % II. is number perfect?
    yes_no = is_perfect(n);
    % III. display answer
    if (yes_no)

```

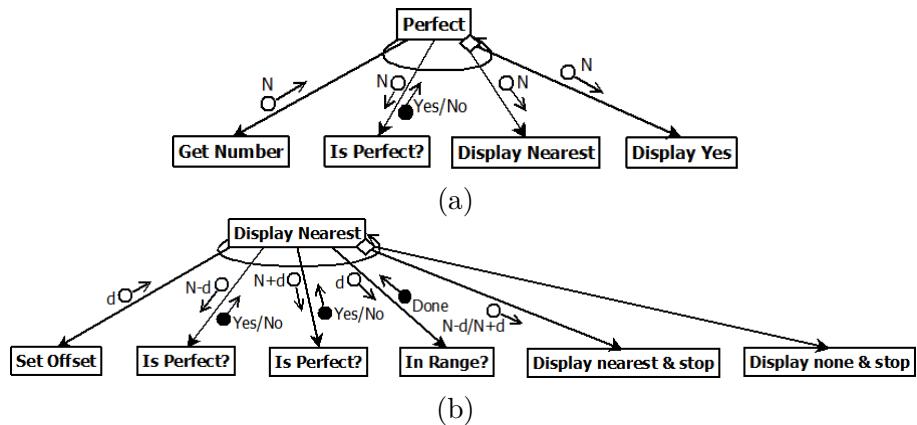
```

        fprintf('%d is perfectn', n);
else
    fprintf('%d isn't perfectn', n);
end
end

```

3. Modify your solution so that, if the number entered is not perfect, the program will report the nearest perfect number to the number entered. If no perfect number is found within a predefined range of the number (say, 100 either side) then a suitable message should be displayed. You may need to make changes to the structure charts, pseudocode and implementation.

The figure below shows the changes to the original structure charts (see Figures 7.1-7.2): (a) first level factoring; (b) further factoring of *Display Nearest*. The structure charts for *Is Perfect?* and *Get Number* from Figures 7.1b-c are unchanged.



The *Display Nearest* module could alternatively be represented by pseudocode similar to the following:

*PSEUDOCODE - Display Nearest:*

```

For d from 1 to 100
If N-d is perfect

```

```

    Display message
    Return
Else If N+d is perfect
    Display message
    Return
Display message that no perfect number within 100

```

The modified implementation is shown below.

*ex3\_perfect.m:*

```

% read integers and check if they are perfect
% (i.e. equal to sum of their divisors)
% loop multiple times
n = 0; %so that code enters loop first time
while n >= 0
    % I. get number
    n = input('Enter positive integer: ');
    if n < 0
        break;
    end
    % II. is number perfect?
    yes_no = is_perfect(n);
    % III. display answer
    if (yes_no)
        fprintf('%d is perfect\n', n);
    else
        DisplayNearest(n);
    end
end

```

*DisplayNearest.m:*

```

function DisplayNearest (num)
% Usage:
% DisplayNearest (num)
% num : Number to be tested
% look for perfect number within 100 of num
for d=1:100

```

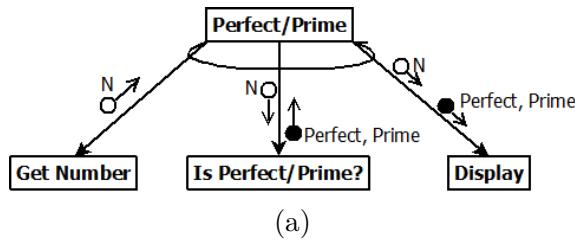
```

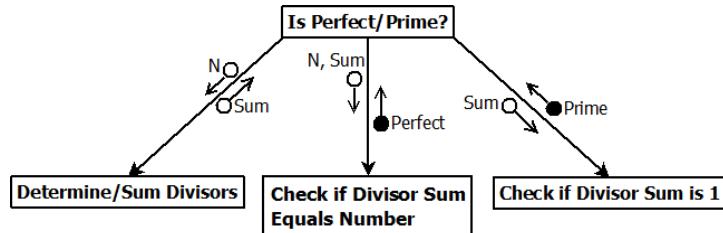
if (is_perfect(num-d))
    fprintf('%d not perfect - nearest = %d\n', ...
            num, num-d);
    return;
elseif (is_perfect(num+d))
    fprintf('%d not perfect - nearest = %d\n', ...
            num, num+d);
    return;
end
%
% not found
fprintf('%d not perfect - none within 100\n', num);
end %function

```

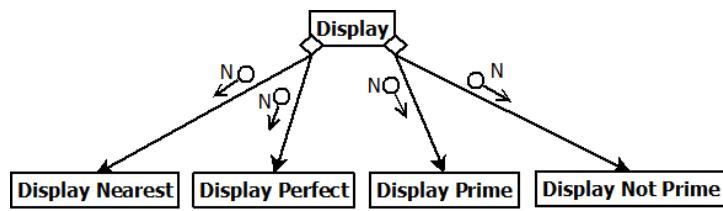
4. A prime number is one which has no divisors apart from 1 and itself. Therefore, much of the processing required to determine if a number is perfect or prime is the same. Modify your solution so that, for the number being tested, it also reports whether or not it is prime. Again, you should make any necessary changes to the structure charts, pseudocode and implementation. (For the purpose of this exercise, try not to use the built-in MATLAB function `isprime`.)

The figure below shows the changes to the original structure charts (see Figures 7.1-7.2 and the solution to Exercise 3): (a) first level factoring; (b) further factoring of *Is Perfect/Prime?*; (c) further factoring of *Display*. Other structure charts are unchanged.





(b)



(c)

The modified implementation is shown below.

*ex4\_perfect.m:*

```

% read integers and check if they are perfect
% (i.e. equal to sum of their divisors)or prime
% loop multiple times
n = 0; %so that code enters loop first time
while n >= 0
    % I. get number
    n = input('Enter positive integer: ');
    if n < 0
        break;
    end
    % II. is number perfect or prime?
    [perfect, prime] = is_perfect_prime(n);
    % III. display answer
    if (perfect)
        fprintf('%d is perfect\n', n);
    else
        DisplayNearest(n);
    end
end

```

```

    end
    if (prime)
        fprintf('%d is prime\n', n);
    else
        fprintf('%d is not prime\n', n);
    end
end

is_perfect_prime.m:
function [perfect prime] = is_perfect_prime(num)
% Usage:
% [perfect prime] = is_perfect_prime(num)
% num : Number to be tested
% perfect : Output, Boolean value – is num perfect?
% prime : Output, Boolean value – is num prime?
% sum divisors
divisor_sum = 1;
d = 2;
while d <= (num/2)
    if (mod(num, d) == 0)
        divisor_sum = divisor_sum + d;
    end
    d = d + 1;
end
% is sum = n?
if (divisor_sum == num)
    perfect = true;
else
    perfect = false;
end
% prime?
if (divisor_sum == 1)
    prime = true;
else
    prime = false;
end
end %function

```

## 12.8 Chapter 8 - Visualisation and User Interfaces

Examples of possible solutions are shown in framed text.

1. The file *patient\_data.txt* (available through the KEATS system) contains four pieces of data for multiple patients: whether they are a smoker or not ('Y'/'N'), their age, their resting heart rate and their systolic blood pressure. Write a program to read in these data, and produce separate arrays of age and heart rate data for smokers and non-smokers.

```
% read data (multiple data types)
fid = fopen('patient_data.txt');
data = textscan(fid, '%c %d %d %*d');
fclose(fid);
% separate into smokers/non-smokers
smokers_ages = data{2}(find(data{1}=='Y'));
smokers_hr = data{3}(find(data{1}=='Y'));
nonsmokers_ages = data{2}(find(data{1}=='N'));
nonsmokers_hr = data{3}(find(data{1}=='N'));
```

Using these data, produce plots of age against heart rate for smokers and for non-smokers,

- (a) on the same figure with a single use of the plot command;

```
plot(smokers_ages, smokers_hr, 'xb', ...
      nonsmokers_ages, nonsmokers_hr, 'xr');
title('Age vs. heart rate for ...
smokers/non-smokers');
xlabel('Age');
ylabel('Heart rate');
legend('Smokers', 'Non-smokers');
```

- (b) on multiple figures with the figure command;

```
figure;
plot(smokers_ages, smokers_hr, 'xb');
```

```

title('Age vs. heart rate for smokers');
xlabel('Age');
ylabel('Heart rate');
figure;
plot(nonsmokers_ages, nonsmokers_hr, 'xr');
title('Age vs. heart rate for non-smokers');
xlabel('Age');
ylabel('Heart rate');

```

(c) on the same figure with the hold command;

```

figure;
plot(smokers_ages, smokers_hr, 'xb');
hold on;
plot(nonsmokers_ages, nonsmokers_hr, 'xr');
title('Age vs. heart rate for ...
smokers/non-smokers');
xlabel('Age');
ylabel('Heart rate');
legend('Smokers', 'Non-smokers');

```

(d) on subplots with the subplot command.

```

figure;
subplot(2,1,1);
plot(smokers_ages, smokers_hr, 'xb');
title('Age vs. heart rate for smokers');
xlabel('Age');
ylabel('Heart rate');
subplot(2,1,2);
plot(nonsmokers_ages, nonsmokers_hr, 'xr');
title('Age vs. heart rate for non-smokers');
xlabel('Age');
ylabel('Heart rate');

```

2. A research team wish to evaluate a new algorithm for aligning 3-D medical images (this process of image alignment is known as *image*

*registration*). To evaluate their algorithm the team have performed multiple experiments involving registering image pairs, and computed two measures of registration success for each experiment. The first is the value of the *normalised cross correlation* (a measure of image similarity) between the image pairs. The second is a *landmark error*, which was determined by observers manually clicking on corresponding anatomical landmarks in the image pairs and computing the distance between them. The values of these measures for 10 separate experiments are provided in the files *ncc.txt* and *points.txt* respectively (available through the KEATS system). Both files contain 10 rows and two columns: the columns represent the experiment number and the respective evaluation measure.

Write a MATLAB program to read in these data and produce a single plot showing the values of both evaluation measures (on the *y*-axis) against the experiment number (on the *x*-axis). Annotate the *x* and *y* axes of your plot and add a suitable title. Your final figure should look like the one shown in Figure 8.6.

```
% read image similarity values
data1 = readmatrix('ncc.txt');
% read landmark errors
data2 = readmatrix('points.txt');
% display both datasets on same plot and annotate
yyaxis left;
plot(data1(:,1), data1(:,2))
ylabel('NCC');
yyaxis right;
plot(data2(:,1), data2(:,2));
ylabel('Landmark error (mm)');
title('Image similarity vs. landmark error');
xlabel('Experiment number');
```

3. Using the same *patient\_data.txt* data you used in Exercise 1, write a MATLAB program to produce a 3-D plot of the patient data. The three coordinates of the plot should be the age, heart rate and blood pressure of the patients, and the smokers and non-smokers should be displayed using different symbols on the same plot. Annotate your figure appropriately.

```

% read data (multiple data types)
fid = fopen('patient_data.txt');
data = textscan(fid, '%c %d %d %d');
fclose(fid);
% separate into smokers/non-smokers
smokers_ages = data{2}(find(data{1}=='Y'));
smokers_hr = data{3}(find(data{1}=='Y'));
smokers_bp = data{4}(find(data{1}=='Y'));
nonsmokers_ages = data{2}(find(data{1}=='N'));
nonsmokers_hr = data{3}(find(data{1}=='N'));
nonsmokers_bp = data{4}(find(data{1}=='N'));
% 3-D plot
plot3(smokers_ages, smokers_hr, smokers_bp, ...
    'xb', nonsmokers_ages, nonsmokers_hr, ...
    nonsmokers_bp, 'or');
title('Age vs. heart rate vs. blood pressure');
xlabel('Age');
ylabel('Heart rate');
zlabel('Systolic blood pressure');
legend('Smokers', 'Non-smokers');

```

4. A (1-D) Gaussian (or *normal*) distribution is defined by the following equation:

$$\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (12.2)$$

where  $x$  is the distance from the mean,  $\mu$ , of the distribution. The parameter  $\sigma$  is the standard deviation of the distribution, which affects how ‘spread out’ the distribution will be. An example of such a distribution is shown in Figure 8.7a.

- (a) Write a MATLAB function to display a figure of a 1-D Gaussian distribution. The function should take the following arguments: the standard deviation and mean of the Gaussian, and the minimum/maximum axis values.

```

ex4_gauss1D.m:
function ex4_gaussian_1D(sigma, mu, mn, mx)

```

```

% displays a 1-D Gaussian distribution
% Usage:
% ex4_gaussian_1D (sigma, mu, mn, mx)
% sigma: standard deviation
% mu: centre of distribution
% mn, mx: min/max axis values
% 1-D: create array representing x-axis values
X = mn:0.1:mx;
% Guassian distribution
Y = 1/(sigma*sqrt(2*pi))* ...
    exp(-(X-mu).^2 / (2 * sigma * sigma));
% plot
plot(X,Y, '-k')
title('Guassian distribution');
xlabel('X');
ylabel('Y');

```

- (b) Write another MATLAB function to display a 2-D Gaussian distribution. To make a 2-D distribution you should replace the  $(x - \mu)$  term in Equation (8.1) with a distance from a 2-D point which acts as the centre of the distribution (for example, the point  $(0, 0)$ ). See Figure 8.7b for an example. This function should take the following arguments: the standard deviation of the distribution, the  $x/y$  coordinates of the centre of the distribution, and the minimum/maximum axis values. You can experiment with different standard deviations but to begin with try a value of 3 and plot the distribution centred on  $(0, 0)$  and between the  $x/y$  coordinates of  $-10 \dots 10$ .

```

ex4_gauss2D.m:
function ex4_gaussian_2D(sigma, x0, y0, mn, mx)
% displays a 2-D Gaussian distribution
% Usage:
% ex4_gaussian_2D (sigma, x0, y0, mn, mx)
% sigma: standard deviation
% x0, y0: centre of distribution

```

```
% mn, mx: min/max axis values
% 2-D: create arrays for a grid of x/y values
[X, Y] = meshgrid(mn:0.1:mx, mn:0.1:mx);
% Guassian distribution
R = sqrt((X-x0).^2 + (Y-y0).^2);
Z = 1/(sigma*sqrt(2*pi))* ...
    exp(-R.^2 / (2 * sigma * sigma));
% mesh plot
mesh(X, Y, Z)
title('Gaussian distribution');
xlabel('X');
ylabel('Y');
zlabel('Z');
```

5. Write a MATLAB program that will read in an image from a file and display it, and then allow the user to *annotate* the image (i.e. add text to it at specified locations). To perform the annotation the user should be prompted to enter a piece of text, then click their mouse in the image to indicate where they want it to appear. The user should be allowed to add multiple annotations in this way, and the program should terminate and save the annotated image with a new file name when empty text is entered.

A sample image file, called *brain\_mr.tif*, which you can use to test your program, will be provided for you through the KEATS system.

(*Hint: look at the MATLAB documentation for the gtext command.*)

```
% read image
im = imread('brain_mr.tif');
% display image
imshow(im);
% add text annotations
str = 'x'; %anything apart from empty text
while ~isempty(str)
    str = input('Enter text: ', 's');
    fprintf('Click mouse to position text\n');
    gtext(str);
```

```

end
% save image
saveas(gcf, 'brain_mr_annotated.tif');

```

6. Write a MATLAB program to perform *image thresholding*. Image thresholding refers to creating a binary image from a greyscale image, where the binary image has a ‘high’ value (e.g. 1) where the corresponding original image intensity is greater or equal to a given threshold value, and a ‘low’ value (e.g. 0) otherwise.

Your program should first read in an image from a file. It should then display to the user the minimum and maximum intensities present in the image. The user should be prompted to enter a threshold value (which should be between the minimum and maximum values). A new thresholded image should then be created, with the value 255 for the ‘high’ pixels and 0 for the ‘low’ ones. A figure should be displayed showing both the original and thresholded images as subplots (as shown in Figure 8.8). Finally, the thresholded image should be written to a new external image file.

You can use the *brain\_mr.tif* file again to test your program.

```

% read image
im = imread('brain_mr.tif');
% get threshold
mn = min(im(:));
mx = max(im(:));
fprintf('Min/max intensities are %d/%d\n', mn, mx);
t = input('Enter threshold: ');
% check threshold
if (t < mn) || (t > mx)
    error('Error: threshold out of range');
end
% threshold image
im2 = im;
im2(find(im>=t))= 255;
im2(find(im<t))= 0;
% display original and thresholded images

```

```
subplot(2,1,1);
imshow(im);
title('Original image');
subplot(2,1,2);
imshow(im2);
str = ['Thresholded image using ' num2str(t) ];
title(str);
% save image
imwrite(im2, 'brain_mr_thresholded.tif');
```

7. Try visualising 3-D imaging data using the `imshow3D` function. The source code for `imshow3D` is available to you through KEATS, as well as a MATLAB *MAT* file containing some 3-D imaging data (called *MR\_heart.mat*).

## 12.9 Chapter 9 - Code Efficiency

Solutions are shown in framed text.

1. (a) Calculate how much memory will be required by the following assignment

```
x = ones(1000)
```

The call to `ones` with a single argument  $N$  will default to providing a square array with size  $N \times N$ . In this case, it will return a  $1000 \times 1000$  array which has 1,000,000 elements. Using the default double type, we need 8 bytes per element, so the total requested by the call is 8,000,000 bytes which is around 7.6 MB ( $8,000,000 \div 1024 \div 1024$ )

- (b) Explain why a call to `z = 3 * zeros(1000000)` is likely to cause problems when it is run.

Similar to the previous example, this call will request a  $N \times N$  array where  $N = 1,000,000 = 10^6$ . This will need  $10^{12}$  elements each with 8 bytes which is equivalent to around 7500 GB or 7.3 TB. Most computers do not have that much RAM so this call is likely to fail and cause problems for the operating system.

2. (a) Give two reasons why the following code is not efficient

```
myArr = 0;  
N = 100000;  
  
for n = 2:N  
    myArr(n) = 2 + myArr(n-1);  
end
```

The array variable `myArr` is initialised to be of size  $1 \times 1$  and it is ‘grown’ for each iteration of the loop.

The fact that a loop is needed to initialise the array is also inefficient, it can be done more efficiently using built in MATLAB initialisation functions.

- (b) Re-write the above code so that it runs more efficiently

One way is to use the colon operator which has the syntax

```
arrayVar = start:step:end
```

In this case the array can be initialised to the same values as those above using one line (after setting  $N$ ):

```
N = 100000;  
myArr2 = 0:2:(2*(N-1));
```

- (c) Use the built-in timer functions to estimate the speed-up given when the above code is re-written

Performance will vary from machine to machine but the optimised code should run about 100 times faster than the first set of code.

```
tic  
myArr = 0;  
N = 100000;  
  
for n = 2:N  
    myArr(n) = 2 + myArr(n-1);  
end  
toc  
  
tic  
myArr2 = 0:2:(2*(N-1));  
toc  
  
% Check they give the same result  
% ('all' checks if all corresponding elements
```

```
% match)
if all(myArr == myArr2)
    disp('success')
end
```

Should produce output similar to that below

```
Elapsed time is 0.031632 seconds.
Elapsed time is 0.001445 seconds.
```

3. Re-write the following code so that it is more time efficient.

```
N = 20;

a = [];
for j = 1:N
    a(j) = j * pi / N;
end

for k = 1:numel(a)
    b(k) = sin(a(k));
end

for m = 1:numel(a)
    c(m) = b(m) / a(m);
end

plot(a, b, 'r')
hold on
plot(a, c, 'k')
legend('sin', 'sinc')
```

This code can be shortened greatly using vectorised operations and avoiding loops

```
N = 20;

% Make sure you understand this line.
a = pi/N:pi/N:pi;

% Element-wise operation but no dot needed.
b = sin(a);
```

```
% Element-wise operation shown by dot
c = b ./ a;

plot(a, b, 'r')
hold on
plot(a, c, 'k')
legend('sin', 'sinc')
```

4. (a) Use the built-in random function `rand` to generate a  $5 \times 6$  array of uniform random numbers, and assign this to an array called `A`.

```
A = rand(5,6)
```

- (b) Use loops to inefficiently check the elements in `A`. If an element is less than 0.6, replace it with a value of 0.

```
A = rand(5,6)
[M,N] = size(A);
for row = 1:M
    for col = 1:N
        if A(row, col) < 0.5
            A(row, col) = 0;
        end
    end
end
```

- (c) Use logical indexing to achieve the same result as in the previous part, i.e. by avoiding loops

```
A = rand(5,6);
indices = A < 0.5;
A(indices) = 0;
```

or more compactly as

```
A = rand(5,6);
A(A < 0.5) = 0;
```

- (d) Use logical indexing to set any values in A that are greater than 0.7 to -1

```
A = rand(5,6);
indices = A > 0.7;
A(indices) = -1;
```

or

```
A = rand(5,6);
A(A > 0.7) = -1;
```

- (e) Use logical indexing to set any values in A that are greater than 0.7 *OR* less than 0.3 to -1.

*(Hint: use the bitwise logical or operator |)*

```
A = rand(5,6);
indices = A > 0.7 | A < 0.3;
A(indices) = -1;
```

or

```
A = rand(5,6);
A(A > 0.7 | A < 0.3) = -1;
```

5. (a) Write a MATLAB script to compute and display a 2-D array representing a times table for integers from 1 up to a specified maximum  $N$ . Use the `disp` statement to display the array. E.g. for  $N = 5$  the program should compute the following array:

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

For this implementation, concentrate on writing clear, understandable code that works.

E.g. this naive implementation uses a 2-D array (not pre-allocated) and nested `for` loops.

```
for a=1:n
    for b=1:n
        times_table(a,b) = a*b;
    end
end
disp(times_table);
```

- (b) Now write another implementation, this time optimised for speed. Run and time both implementations for values of  $N$  up to 200, and plot the execution times for both against the value of  $N$ . (Don't include the `disp` statement in the timed section.)

The new implementation uses MATLAB vectorised operations.

```
% Produce plots of speed vs n
clear
maxN = 200;
t1 = zeros(maxN,1);
t2 = zeros(maxN,1);
t3 = zeros(maxN,1);
for n=1:maxN

    % initial attempt at times table code
    tic
    for a=1:n
        for b=1:n
            times_table(a,b) = a*b;
        end
    end
    %disp(times_table);
    t1(n) = toc;

    % optimised for speed - using repmat
    tic
    times_table = repmat(1:n,n,1);
    times_table = times_table .* ...
```

```

    repmat((1:n)',1,n);
%disp(times_table);
t2(n) = toc;

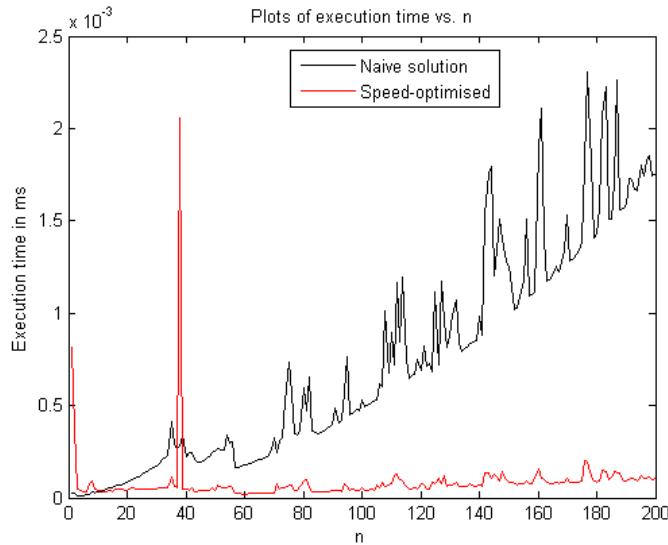
% optimised for speed - using matrix
% multiplication
tic
times_table = (1:n)' * (1:n);
%disp(times_table);
t3(n) = toc;

end

% plot times
plot(1:maxN,t1,'k', 1:maxN,t2,'r', ...
1:maxN,t3,'b');
legend('Naive solution', 'Speed-optimised ... ...
(repmat)', 'Speed optimised (mat mult)');
title('Plots of execution time vs. n');
xlabel('n');
ylabel('Execution time in ms');

```

The plots for these implementations are shown below.



As you can see, both speed-optimised implementations are sig-

nificantly faster than the naive implementation, with the difference becoming more marked as the value of  $N$  increases. The implementation based on matrix multiplication seems to be slightly faster than the one that uses `repmat`.

- (c) Finally, write a third implementation that is optimised for memory.

This time we don't use an array at all, just compute and display in nested for loops.

```
%% optimised for memory
for a=1:n
    for b=1:n
        fprintf('%4d', a*b); % NB only works ...
        if max number has < 4 digits
    end
    fprintf('\n');
end
```

6. Download from KEATS both versions of the code that counts the number of full binary trees with  $N$  leaves:

- The naive recursive version (Example 9.11)
- The version that uses memoisation (Example 9.12)

- (a) Write a MATLAB script to measure the execution times of the two implementations for a single value of  $N$ .

Be careful when choosing the value of  $N$ . When using the naive version, a value of  $N$  greater than around 14 could lead to a long wait!

```
ex_fbt.m

N=14;

% naive
tic;treecount_naive(N);toc;
```

```
% DP
tic;treecount_dp(N);toc;
```

The difference between the two versions is quite significant. The times will vary according to the machine but the following shows typical results for  $N = 14$ :

```
Elapsed time is 5.724823 seconds.
```

```
Elapsed time is 0.009562 seconds.
```

which shows a speed up factor of around 500.

- (b) Extend your script to compute the execution times for the two implementations for  $N=1 \dots 14$ , and plot both on the same graph against  $N$ .

```
ex_fbt_plots.m

% time and plot naive and DP implementations
% of full binary tree count software
maxN = 14;
times = zeros(maxN,2);
disp('Running ...');
for N=1:maxN

    disp(N);

    % naive
    tic;treecount_naive(N);times(N,1)=toc;

    % DP
    tic;treecount_dp(N);times(N,2)=toc;

end

% plots
plot(1:maxN, times(:,1),'k-', ...
      1:maxN, times(:,2),'r-');
title('Execution time vs. N for full binary ...')
```

```

    tree count problem')
legend('Naive implementation', 'Dynamic ...
    programming implementation');
xlabel('N');
ylabel('t (secs)');

```

7. The Fibonacci sequence is

$$1, 1, 2, 3, 5, 8, \dots$$

and it is defined mathematically and recursively by the formula

$$f_1 = 1, \quad f_2 = 1, \quad f_{n+1} = f_n + f_{n-1}, \text{ for } n > 2$$

(Exercise 9 from Section 3 involved writing a recursive implementation of a function to compute Fibonacci numbers.)

- (a) If you haven't already, write the recursive function that implements the recursive Fibonacci formula. Call the function *fibonacciRecursive* and save it to a suitable .m file.

This is the direct implementation of the recursive formula.

```

function f = fibonacciRecursive(n)
% Usage fibonacci(n)
% Return the nth fibonacci term
%
% Input: n, the position of a term in the ...
%         sequence
%         Assumed to be positive
% Output: f the value of the term

if n > 2
    f = fibonacciRecursive(n-1) + ...
        fibonacciRecursive(n-2);
else
    % Base case: either n==1 or n==2
    f = 1;
end

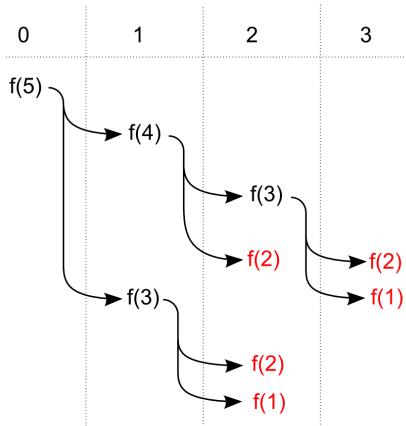
```

- (b) Each call to the function will make further calls recursively with smaller values of  $n$ . If the function is called originally (recursion depth 0) with a value of  $n = 5$ , work out how many calls there are for the function with values of  $n = 4, 3, 2$  and  $1$ , and how many calls in total.

We shorten the name of the function to  $f(n)$  for brevity. As illustrated below, counting up the calls in the illustration, starting from  $n = 5$  we have

| call  | $f(5)$ | $f(4)$ | $f(3)$ | $f(2)$ | $f(1)$ |
|-------|--------|--------|--------|--------|--------|
| count | 1      | 1      | 2      | 3      | 2      |

giving a total call count of 9. Note how some evaluations are called repeatedly which is unnecessary.



The numbers along the top indicate the depth of the recursion and the calls marked in red hit the ground case and make no further recursive calls. The same call to  $f(2)$ , for example, is made three times.

- (c) Extend this to work out the number of all such calls to the function when we start with higher values of  $n$ , i.e.  $n = 6, n = 7$ , etc.

The calls that start from  $n = 6$  will use all the calls that we would get starting from  $n = 5$  and an entire set that we would get starting from  $n = 4$ .

| call  | $f(6)$ | $f(5)$ | $f(4)$  | $f(3)$  | $f(2)$  | $f(1)$  |
|-------|--------|--------|---------|---------|---------|---------|
| count | 1      | 1      | $1 + 1$ | $2 + 1$ | $3 + 2$ | $2 + 1$ |

giving a total call count of

$$1 \text{ for } f(6) + 9 \text{ for } f(5) + 5 \text{ for } f(4) = 15$$

If we start with  $n = 7$ , the number of calls needed will be

$$1 \text{ for } f(7) + 15 \text{ for } f(6) + 9 \text{ for } f(5) = 25$$

The next few terms in the sequence will be

| Starting $n$ | Count              |
|--------------|--------------------|
| 8            | $41 = 1 + 25 + 15$ |
| 9            | $67 = 1 + 41 + 25$ |
| :            | :                  |

and so on.

- (d) One way to avoid the overhead of multiple calls which evaluate the same value of the function in Exercise 7 is to avoid recursive calls altogether. One way to do this is calculate the values in a ‘bottom up’ way, starting with the values for the lowest values of  $n$  and ‘building up’. This way, we can visit each value of  $n$  once only.

By defining values for the current and previous terms in the sequence (or otherwise) write a function that loops upwards to the required value of  $n$ . Within each iteration, the following will need to be done:

- Calculate the value of the next term in the sequence
- Increment a counter that keeps track of how far we have moved along the sequence
- Update variables holding the previous and current terms based on the incremented counter  
*(Hint: use the value calculated in the first step).*
- Check if the loop has reached the required value of  $n$ , stopping if this is the case and returning the required value.

NB such *bottom-up evaluation* is another technique used in dy-

namic programming (as well as memoisation).

One possible implementation

```
function f = fibonacciBottomUp(n)

    % Can return immediately if n < 2
    if n < 2
        f = 1;
        return
    end

    % Counter:
    k = 2;
    % current and previous values.
    fCurrent = 1;
    fPrevious = 1;

    % Main loop
    while k < n
        fNext = fCurrent + fPrevious;
        % update k
        k = k + 1;
        % update the current and previous values
        fPrevious = fCurrent;
        fCurrent = fNext;
    end

    % Assign return value
    f = fCurrent;

end
```

- (e) Estimate the speed up of the dynamic programming bottom-up evaluation of the Fibonacci code compared with the previous recursive version. How many times faster is the new code? It may be advisable not to call the previous version with a value much higher than 30 or so.

Performance will depend on the machine but the following gives an indication that, for  $n = 30$ , there is a speed-up factor

of close to 10,000.

```
>> tic; disp( fibonacciRecursive(30) ) ; toc  
832040
```

```
Elapsed time is 6.505600 seconds.
```

```
>> tic; disp( fibonacciBottomUp(30) ) ; toc  
832040
```

```
Elapsed time is 0.000670 seconds.
```

## 12.10 Chapter 10 - Images and Image Processing

Solutions are shown in framed text.

These exercises involve use of a number of sample medical images which you will need to download from the KEATS system before proceeding.

1. Using the *CT\_ScoutView.jpg* image, produce, display and save a binarised image of the head using an appropriate threshold, i.e.



```
head_ct=ct_im(25:200,370:500);
head_mask = head_ct>100;
imshow(head_mask)
imwrite(head_mask, 'ct_head-mask.png');
```

2. Write two functions called `thresholdLow` and `thresholdHigh`, which both take an image and a threshold as their two arguments, and produce a thresholded image as their result (the thresholded images should have zero intensity at any pixel that had a value below/above the threshold in the original image).

*thresholdLow.m:*

```
function thresh_im = thresholdLow (orig_image, ...
    thresh)
% thresholdLow: thresholds an image so that ...
    % intensities below
    % thresh = 0
```

```

%
% Usage:
%   thresh_im = thresholdLow (orig_image, thresh)
% orig_image : input image
% thresh     : threshold value
% thresh_im  : thresholded image

thresh_im = orig_image;
thresh_im(orig_image<thresh) = 0;

end

thresholdHigh.m:
function thresh_im = thresholdHigh (orig_image, ...
    thresh)
% thresholdHigh: thresholds an image so that ...
    intensities above
% thresh = 0
%
% Usage:
%   thresh_im = thresholdHigh (orig_image, thresh)
% orig_image : input image
% thresh     : threshold value
% thresh_im  : thresholded image

thresh_im = orig_image;
thresh_im(orig_image>thresh) = 0;

end

```

3. Write two functions called `truncateLow` and `truncateHigh`, which both take an image and a threshold as their two arguments, and produce a truncated image as their result (the truncated images should have the intensity of any pixel below/above the threshold set to the threshold).

```

truncateLow.m:
function thresh_im = truncateLow (orig_image, thresh)
% truncateLow: truncates an image so that ...
    intensities below

```

```

% thresh are set to thresh
%
% Usage:
%   thresh_im = truncateLow (orig_image, thresh)
%   orig_image : input image
%   thresh      : threshold value
%   thresh_im   : thresholded image

thresh_im = orig_image;
thresh_im(orig_image<thresh) = thresh;

end

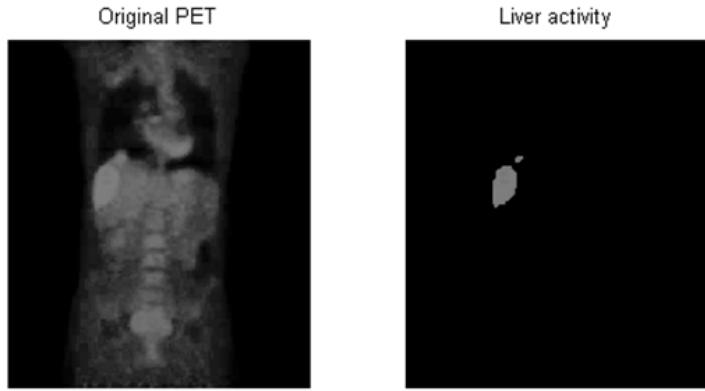
truncateHigh.m:
function thresh_im = truncateHigh (orig_image, ...
    thresh)
% truncateHigh: truncates an image so that ...
    % intensities above
% thresh are set to thresh
%
% Usage:
%   thresh_im = truncateHigh (orig_image, thresh)
%   orig_image : input image
%   thresh      : threshold value
%   thresh_im   : thresholded image

thresh_im = orig_image;
thresh_im(orig_image>thresh) = thresh;

end

```

4. Write a pipeline of operations to process *pet-image-liver.png* to highlight the area of high activity in the right liver, i.e. output of program should be:



(Also display a count of the number of pixels that are non-zero in the final image.)

```

clear; close all;

% read image
pet_im=imread('pet-image-liver.png');

% pipeline - copy original image (so that it ...
    % keeps the uint8 data type
new_pet = pet_im;
% set to zero outside region of interest
new_pet(:) = 0;
new_pet(100:175,75:125) = pet_im(100:175,75:125);
% set to zero below threshold = 115
new_pet(new_pet < 115) = 0;

% display result
subplot(1,2,1); imshow(pet_im); title('Original ... PET');
subplot(1,2,2); imshow(new_pet); title('Liver ... activity');

% count number of pixels
% short version
pet_count = new_pet > 0;
count1 = sum(pet_count(:))
% long version
count2 = 0;
for x=1:size(new_pet,1)
    for y=1:size(new_pet,2)

```

```
if (new_pet(x,y) > 0)
    count2 = count2 + 1;
end
end
count2
```

5. Write a program to interactively help a user to apply some of the image processing operations we have seen in this section. It is up to you how to structure your code. The interaction with the user can be via the command window, for example through the `input` function, or via a GUI, for example by using a text box. As a minimum, your code should

- Allow the user to specify an image file to load.
- Allow the user to run a basic image processing operation
- Allow the user to visualise or save the result

Preferably, the user should be able to chain operations together, applying one operation to the image that results from a previous one. It will be important to follow the advice on program design given in Section 7.

This is an open-ended activity and there is no single solution. It is an opportunity to show what you have learned in this section and to apply the skills you have learned in the previous sections.

## 12.11 Appendix - Graphical User Interfaces

Examples of possible solutions are shown in framed text.

1. Create a new blank GUI using the guide tool and carry out the following steps:
  - (a) Add two text boxes to the GUI: one static text box and one edit text box.
  - (b) Save the GUI in a folder called guiEx1.
  - (c) Run the GUI to make sure it works.
  - (d) Close the GUI and change the code in the edit text box callback function so that the text entered by the user is displayed in the other (static) text box.

*Hints:*

- *The handles variable in the automatically generated code will have fields for each of the edit and static text boxes. These should be named edit1 and text1 (although the number might vary).*
- *Use the set and get functions to set and access the 'String' properties in the text boxes.*

Inside the function edit1\_Callback for the edit text box, code similar to the following needs to be added:

```
% Get what the user entered:  
userString = get(hObject, 'String');  
  
% Update the other (static) text box:  
set(handles.text1, 'String', userString);
```

2. This question is a continuation of the GUI examples in the main text (Examples 11.1 to 11.9).

Recall that the default amplitude of the exponential decay function is hard-coded in the script. It is set to a value of 200 in the myPlot function. This task is to allow the user to change the amplitude.

Add another pair of text boxes to the GUI using the guide tool: one static text box and one edit text box. Use the edit text box to take a number that is used for the amplitude of the function plotted. Set the string for both new text boxes in the script during the call to the opening function for the GUI.

The strings for the Text boxes can be set in the function myGui\_OpeningFcn. The same function can also have code added to set the amplitude as a field in the handles structure.

```
% ...
handles.decayFac = 0.05;
handles.amp = 200;

myPlot(handles)

set(handles.edit1, 'String', '0.05')
set(handles.edit2, 'String', '200')
set(handles.text2, 'String', 'Amplitude')
% ...
```

where we have used amp to represent the field name for the amplitude.

The callback for the new edit text box is called edit2\_Callback (your number might be different). Its code is very similar to the the code for the decay factor's text box:

```
function edit2_Callback(hObject, eventdata, handles)
% ... comments and hints here ...

amplitude = str2double(get(hObject,'String'));

if isnan(amplitude)
    return
end

handles.amp = amplitude;

myPlot(handles);

% Update handles structure
guidata(hObject, handles);
```

3. This question continues from the previous question and uses the *myGui* plotting GUI.

Modify the code so that, if a user gives a non-numeric value in an edit text box, the value displayed is re-set to the correct number. That is, the decay factor edit text box should show the current decay factor value (instead of the non-numeric value that the user entered). Similar behaviour should be shown by the amplitude edit text box.

This can be achieved by modifying the `if/then` clauses that check the user input with the `isnan` function. These are in `edit1_Callback` and `edit2_Callback`. For example in the decay factor edit text box (*edit1*) this can be done by adding a call to the `set` function:

```
% ...
if isnan(decayFactor)
    set(hObject, 'String', handles.decayFac)
    return
end
% ...
```

Similarly for the *edit2* Callback function, which is for the amplitude, we can add a `set` call:

```
% ...
if isnan(decayFactor)
    set(hObject, 'String', handles.amp)
    return
end
% ...
```

4. This question continues from the previous questions and also uses the *myGui* plotting GUI.

Insert a pop-up menu onto the GUI to control the colour of the plot. The choice should be between black, red, blue or green.

*Hints:*

- Use the `handles` object to store a field for the plot colour. This should be the single character that MATLAB uses for plot colours: '`k`', '`r`', '`b`', '`g`'.

- Set the '*String*' property for the pop-up menu in the main GUI opening function. Assign a cell array of character arrays to contain what is displayed to the user.
- Update the *myPlot* function so that it uses the plot colour field from the handles structure when it actually calls the *plot* function.

We can set the list of strings that the *popupmenu* should display in the GUI's opening function *myGui\_OpeningFcn* near where the similar properties are set for the other components:

```
% ...
set(handles.popupmenu1, 'String', {'black', ...
    'red', 'blue', 'green'})
%
```

here, the pop-up menu has a tag *popupmenu1* but your number might differ.

We can add a field to the handles structure to store the plot colour inside the same function, perhaps near where the default text for the edit text box(es) are stored.

```
% ...
handles.plotColour = 'k';
%
```

where we have chosen black as the default colour.

Inside the *myPlot* function, we need to update the call to *plot* so that it uses the colour. I.e. the line

```
plot(handles.axes1, x, y)
```

should change to

```
plot(handles.axes1, x, y, handles.plotColour)
```

Finally, the automatically generated callback function for the pop-up menu needs to have the following code added. The comments and hints have been removed to save space. The code shown below used the hints to access what the user chose from the pop-up menu.

```
function popupmenu1_Callback(hObject, eventdata, ...
    handles)
```

```

% Usage and hints go here

contents = cellstr(get(hObject,'String'));
colourString = contents{get(hObject,'Value')};

switch colourString
    case 'black'
        handles.plotColour = 'k';
    case 'blue'
        handles.plotColour = 'b';
    case 'green'
        handles.plotColour = 'g';
    case 'red'
        handles.plotColour = 'r';
    otherwise
        handles.plotColour = 'k';
end

myPlot(handles);

% Don't forget to update handles structure!
guidata(hObject, handles);

```

5. Use the guide to create a blank GUI. Save it as guiEx2.

Add two edit text boxes (their tags should be *edit1* and *edit2*) and a static text box (its tag should be *text1*).

- (a) Write code to set the edit boxes to show a (string) value of 1 at opening.

In the opening function, we can add the following:

```

set(handles.edit1, 'String', '1')
set(handles.edit2, 'String', '1')

```

- (b) Write code so that the static text box shows the sum of the two numbers shown in the edit text boxes.

*Hints:*

- Use the *handles* structure to store numeric values for the con-

tents of the edit text boxes.

- Use a separate function called `runOperation` to actually carry out the adding and the updating of the static text box.

The GUI opening function can set a pair of handles fields for the values:

```
% ...
handles.value1 = 1;
handles.value2 = 1;
%
```

There should also be a `runOperation` function to actually do the adding. It should be called inside the GUI opening function.

```
% ...
runOperation(handles)
%
```

The `runOperation` function itself can be very simple, it just adds the current values and updates the static text box:

```
function runOperation(handles)
% Apply the operation to the contents of the ...
    % edit text boxes.
% Write the result to the static textbox.

result = handles.value1 + handles.value2;
set(handles.text1, 'String', num2str(result));
```

The edit text box callback functions can update these values and make a call the the `runOperation` function. For example, for `edit1` we can have:

```
function edit1_Callback(hObject, eventdata, ...
    % handles)
% Usage and hints here ...

handles.value1 = ...
    str2double(get(hObject, 'String'));

runOperation(handles)
```

```
% Update handles structure  
guidata(hObject, handles);
```

- (c) Write code to guard against non-numeric input in the edit text boxes, i.e. ensure that all stored and displayed values are still numeric even if the user makes a mistake.

We can check for this in the edit callback functions. For example, for the *edit1* text box the above code can be changed to do this as follows:

```
function edit1_Callback(hObject, eventdata, ...  
    handles)  
% Usage and hints here ...  
  
% Get what the user entered.  
val = str2double(get(hObject, 'String'));  
  
if isnan(val)  
    set(hObject, 'String', handles.value1)  
    return  
end  
  
handles.value1 = val;  
  
runOperation(handles)  
  
% Update handles structure  
guidata(hObject, handles);
```

- (d) Use the guide to add a Button Group to the GUI. Add four Radio Buttons to it. We will use each button for one of the four basic arithmetic operations, so modify them so their strings are '*add*', '*multiply*', '*subtract*', and '*divide*'. Modify the string of the panel containing the buttons so that it says 'operation'.

Add code to the script and modify the *runOperation* function so that the correct operation is applied to the current values in the edit boxes.

*Hints:*

- Add a field to the handles to store the current operation.
- Use a `switch` statement in `runOperation` to select which operation to apply.

In the GUI opening function, we can set the operation for the first time.

```
% ...
handles.operation = 'add';
% ...
```

this can be done near the code that sets the `value1` and `value2` fields in the `handles` structure.

The code for the `runOperation` function can be changed from the above to

```
switch handles.operation
case 'add'
    result = handles.value1 + handles.value2;
case 'multiply'
    result = handles.value1 * handles.value2;
case 'subtract'
    result = handles.value1 - handles.value2;
case 'divide'
    result = handles.value1 / handles.value2;
end

set(handles.text1, 'String', num2str(result));
```

The Button Group Panel has a callback for when the selection of button changes. This can be used to update the operation stored in the `handles` structure and to update the result by making a call to the updated `runOperation` function. The name of the relevant callback is `uipanel1_SelectionChangeFcn` where `uipanel1` is the tag of the button group panel:

```
% --- Executes when selected object is ...
%       changed in uipanel2.
function uipanel1_SelectionChangeFcn(hObject, ...
    eventdata, handles)
```

```
% Hints and usage here.

handles.operation = get(hObject, 'String');

runOperation(handles)

% Update handles structure
guidata(hObject, handles);
```

The hints and usage have been removed to save space. The key hint it gives is that

```
% hObject    handle to the selected object in uipanel2
```

so we use the current object (radio button), and get its string to determine which operation has been requested.

# Index

:<sub>15</sub>, 21, 23  
<<sub>36</sub>  
≤<sub>36</sub>  
==<sub>36</sub>, 37  
><sub>36</sub>, 37  
≥<sub>36</sub>  
&<sub>233</sub>  
&&<sub>36</sub>, 37  
readmatrix, 148, 149  
writematrix, 19, 147  
~<sub>36</sub>  
~=<sub>36</sub>  
|<sub>233</sub>  
||<sub>36</sub>

abs, 14  
accessing arrays, 15  
accessing cell arrays, 123  
acos, 14  
addpath, 64  
arguments, 55  
arrays, 13  
ASCII, 104, 127, 147, 157  
asin, 14  
assignment, 40  
atan, 14  
axis, 22

binary files, 157  
binary images, 228, 233, 234  
bottom-up design, 174  
break, 44  
breakpoints, 25  
built-in functions, 13, 15, 40, 43

callback function, 241, 245, 249  
case, 38, 39  
case sensitive, 14  
casting, 128  
catch, 94, 95  
ceil, 14  
cell array, 120, 155  
chaining operations, 236  
char, 19, 112  
character types, 19, 112  
class, 115  
clear, 20, 89  
code analyser, 27, 85  
code reuse, 174  
colour images, 228  
colour maps, 229  
command history, 10  
command window, 10, 26, 55  
comments, 24, 58  
comparison operators, 36  
compilation, 9  
compiled language, 8  
conditional statement, 34, 38, 169  
continue, 44, 45  
cos, 14, 23  
cross, 16  
current folder window, 11, 21

data types, 18, 103  
debugger, 25, 91  
declarative language, 9  
delimiter, 19, 149, 156

design, 166  
disp, 16  
dot, 16  
double, 18, 19, 105, 106  
dynamic programming, 212, 217  
dynamic typing, 58  
  
editor window, 12, 24, 25  
efficiency, 43, 198  
element-wise operations, 14, 16, 91, 118, 119, 206, 210, 233  
else, 35  
end, 35, 38  
eps, 105  
erase, 115  
error, 59, 84, 93  
errors, types of, 85  
exception, 95  
exist, 88  
exp, 14  
eye, 24  
  
factorial, 40  
factoring, 168  
fclose, 150  
feof, 150, 161  
fgetl, 150, 161  
figure, 181  
file details window, 11  
find, 57, 58  
first level factoring, 168  
fix, 14  
floor, 14  
fopen, 150  
for, 39, 40  
format string, 59, 153, 160  
  
fprintf, 56, 151, 158, 161  
fread, 157, 161  
fscanf, 152, 161  
function, 54, 55  
function prototype, 57  
further factoring, 168  
fwrite, 158, 161  
  
gcf, 191  
get, 251  
ground case, 213  
guide tool, 242  
GUIs, 241  
  
handles, 182, 247  
help, 58  
hold, 181  
  
if, 34, 35  
image processing toolbox, 230  
images, 189, 226  
imfinfo, 229, 234  
imformats, 191  
import data wizard, 21  
imread, 190, 232  
imshow, 190, 230  
imshow3D, 191  
imtool, 231  
imwrite, 190, 232  
incremental development, 76, 172  
indices, 58  
Inf, 110  
input, 40  
int16, 18, 104  
int32, 18, 104  
int64, 18, 104  
int8, 18, 104

interpreted language, 8  
intmax, 109  
intmin, 109  
inv, 118  
isa, 116  
ischar, 116  
isempty, 151  
isinf, 111  
isKey, 138  
isnan, 112  
isnumeric, 116  
iteration statements, 39, 41, 169  
  
legend, 23  
length, 16, 58  
linspace, 15  
load, 20, 147, 161  
log, 14  
log10, 14  
logic errors, 85, 89  
logical, 19, 117  
logical indexing, 208  
logical operators, 36  
lower, 115  
  
m-files, 24, 59  
machine code, 9  
magic, 208  
map, 131  
MAT files, 20, 147  
MATLAB, 9  
MATLAB environment, 10  
MATLAB help, 12  
matrices, 23  
max, 16  
mean, 16  
memoisation, 217  
  
memory efficiency, 198  
menu bar, 11  
mesh, 186  
meshgrid, 187, 207  
min, 16  
mod, 14, 209  
  
NaN, 110  
nesting, 45, 205  
num2str, 127  
numeric types, 19, 104  
  
object-oriented language, 9  
ones, 24, 203  
operator precedence, 37  
otherwise, 38, 39  
  
path, 64, 92  
plot, 21, 23, 24, 181  
plot3, 185  
pre-allocating arrays, 204  
procedural language, 9  
prod, 16  
pseudocode, 78, 170  
  
rand, 24, 56  
randi, 73, 87, 98, 204, 285, 292  
readmatrix, 153, 161  
recursion, 67, 212  
return value, 56, 57  
round, 14  
run-time errors, 85, 87, 89  
  
save, 20, 147, 161  
saveas, 191  
scope, 65  
shadowing, 88  
sin, 14, 21

single, 18, 105  
size, 24  
split, 115  
sprintf, 93  
sqrt, 14, 108  
std, 16  
stepwise refinement, 167  
str2double, 111, 128, 251  
string, 114, 115  
strncmp, 151  
strongly typed, 129  
struct, 130, 229  
structure chart, 168–170  
subplot, 183  
sum, 16  
surf, 186  
switch, 38, 39  
syntax errors, 85, 86  
  
tan, 14  
test stubs, 172  
textscan, 155, 161  
thresholding, 234  
tic, 43, 200  
time efficiency, 198, 204  
title, 22  
  
toc, 43, 200  
top-down design, 167  
transpose, 118  
try, 94, 95  
  
uint16, 104  
uint32, 104  
uint64, 104  
uint8, 104, 232  
upper, 115  
  
variables, 11, 12  
  
warning, 84, 93  
weakly typed, 129  
while, 41, 42  
whos, 19, 106, 202, 232  
workspace, 11  
workspace browser, 11, 12, 26, 107  
writematrix, 161  
  
xlabel, 22  
  
ylabel, 22  
yyaxis, 183  
  
zeros, 24, 44, 203