

# AP Induction Week Course

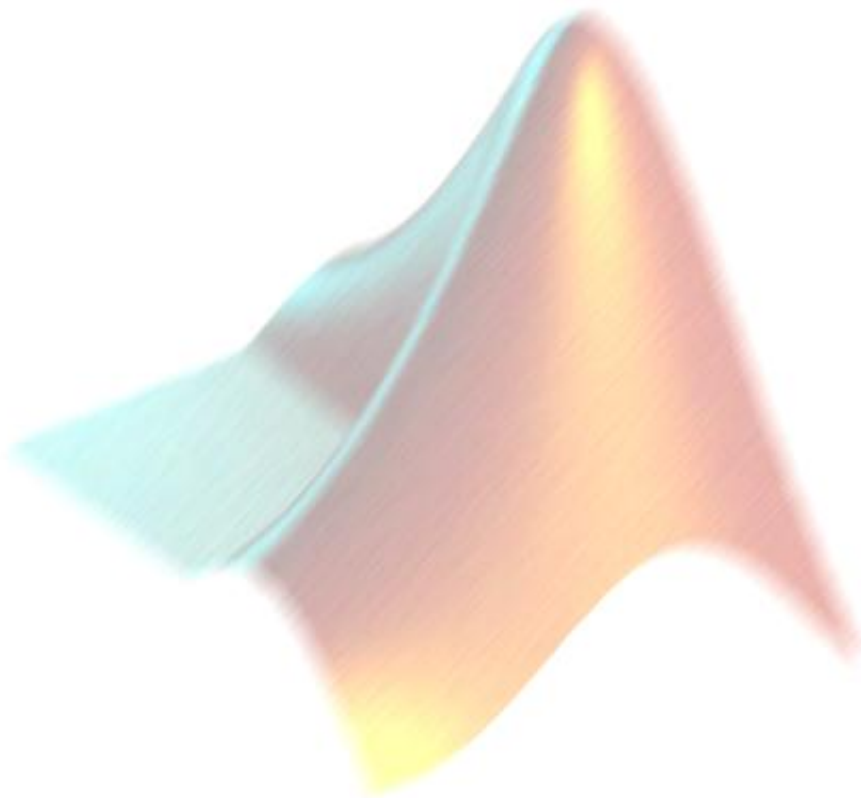
---

## Introduction to Engineering Computation

### Lecture Manual

### MATLAB programming

### Chapters 1 to 5



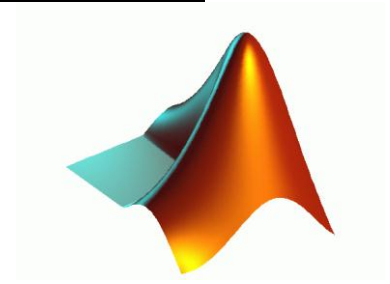
*Department of Engineering Science*

# **MATLAB Lecture Manual Contents**

<b>Chapter 1: An Introduction to MATLAB .....</b>	<b>2</b>
<b>Chapter 2: 1D Arrays, Problem Solving.....</b>	<b>17</b>
<b>Chapter 3: Functions, Problem Solving and Debugging .....</b>	<b>30</b>
<b>Chapter 4: Logical operators and conditional statements .....</b>	<b>43</b>
<b>Chapter 5: Loops.....</b>	<b>63</b>
<b>Appendix: MATLAB Command Reference .....</b>	<b>75</b>

# Chapter 1: An Introduction to MATLAB

MATLAB is an extremely useful piece of mathematical software that is used by engineers to help them solve problems. MATLAB is short for MATrix LABoratory. MATLAB can be used as an advanced calculator and graphing tool for **engineering computation**. It can also be used as a programming language to **develop software** and it has great support for manipulation of matrices.



## Learning outcomes

After working through this chapter, you should be able to:

- Understand why you are taking this course
- Understand why we are teaching you MATLAB
- Use MATLAB as a calculator
- Create and use variables
- Write a script file
- Get input from the user and display output
- Understand the importance of commenting
- Write simple comments

## Why learn to program?

Computers are important tools for modern-day engineering. Computers allow engineers to perform time-consuming tasks quickly and hence solve a wide range of interesting problems. Computers also make it possible for us to visualise our models, allowing us to interpret our results. If no existing software is able to help you solve your problem, you may need to develop your own software by writing computer programs. Writing computer programs is an important skill that is relevant to all branches of engineering.

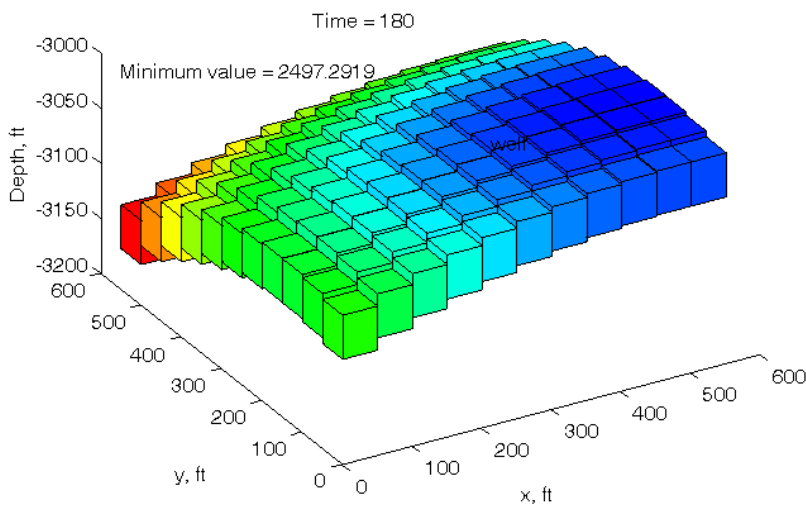


Image: Pressure in an oil reservoir

Computer programs can be used to:

- **Solve engineering problems**

**Visualise the solutions**

## An illustration: solving equations

---

To illustrate how important computers can be to solving engineering problems, consider the relatively simple task of solving a system of simultaneous equations. You are likely familiar with several methods for solving simultaneous equations by hand. If you have taken MM1 you will have encountered solving by both Gaussian elimination and Matrix methods.

### Two equations

Consider the task of solving the following equations:

$$\begin{aligned}2x + y &= 4 \\ x - y &= -1\end{aligned}$$

You should easily be able to solve these by hand to get  $x = 1, y = 2$

### Three equations

Consider the task of solving the following equations:

$$\begin{aligned}2x + y + 2z &= 4 \\ x - y - z &= -1 \\ y - 2z &= 4\end{aligned}$$

With a little more effort we can solve these by hand to get  $x = 1.2, y = 2.8, z = -0.6$

### Ten equations

Now consider the task of solving the following equations:

$$\begin{aligned}2x_1 - x_2 + 3x_4 - x_6 + 2x_7 + 3x_9 + x_{10} &= 1 \\ x_1 + x_3 + 3x_4 + 2x_5 + x_6 + 3x_9 - x_{10} &= 2 \\ 3x_1 + 3x_2 - x_3 - x_4 + 2x_5 + 3x_6 - x_7 + 2x_8 + 3x_9 + x_{10} &= 1 \\ 2x_1 + 3x_2 + 3x_3 + 2x_4 + x_5 + 2x_6 + x_7 + x_{10} &= 3 \\ 3x_1 - x_2 - x_3 + 2x_5 - x_6 + x_7 + 3x_8 + x_9 + 2x_{10} &= 2 \\ x_1 - x_3 + x_4 + 2x_5 - x_7 + 3x_8 - x_9 + 2x_{10} &= 3 \\ x_1 + x_2 + x_4 - x_5 + x_6 + x_7 + 2x_8 + x_9 + 2x_{10} &= 1 \\ 3x_1 + x_2 - x_3 + 3x_4 - x_5 + 3x_6 - x_{10} &= 0 \\ -x_1 + 2x_2 + x_3 + x_4 + 3x_5 - x_6 + x_8 - x_9 - x_{10} &= -1 \\ -x_1 + 2x_2 + 3x_4 - x_5 + 3x_6 + x_7 - x_8 - x_9 &= 2\end{aligned}$$

Could you solve these by hand? In theory it is possible but it would take a long time and you would need to be extremely careful as the chances of making a simple arithmetic error in the resulting pages of algebra is quite high.

## Solving equations with a computer

This system can be solved with relatively little effort in MATLAB.

```
A = [  
 2  -1  0  3  0  -1  2  0  3  1  
 1  0  1  3  2  1  0  0  3  -1  
 3  3  -1  -1  2  3  -1  2  3  1  
 2  3  3  2  1  2  1  0  0  1  
 3  -1  -1  0  2  -1  1  3  1  2  
 1  0  -1  1  2  0  -1  3  -1  2  
 1  1  0  1  -1  1  1  2  1  2  
 3  1  -1  3  -1  3  0  0  0  -1  
 -1  2  1  1  3  -1  0  1  -1  -1  
 -1  2  0  3  -1  3  1  -1  -1  0  
];  
  
b = [  
 1  
 2  
 1  
 3  
 2  
 3  
 1  
 0  
 -1  
 2  
];  
  
x = A \ b
```

← 2D array

← 1D array

← used to solve matrix-vector equation

```
x =  
  
-0.1607  
-0.9621  
 0.4346  
 0.2301  
 0.8881  
 1.1170  
 0.0475  
-0.3688  
-0.1944  
 1.2742
```

## Even more equations

Sometimes engineers need to solve not 10 or even 100 equations, but systems with 10,000 or 100,000 equations. This can be done very quickly using a computer.

Solving systems of linear equations is a common problem in many branches of engineering, for example you will need to solve systems of linear equations for problems related to:

- Operations research
- Mechanics and dynamics
- Electrical Circuits
- Chemical reactions

Quickly solving large systems of equations is just one of the many tasks that MATLAB can accomplish with very little effort from the programmer.

# **Why use MATLAB?**

---

There are a large range of different programming languages available to choose from. We have chosen to introduce you to MATLAB first for a number of reasons:

- MATLAB is an easy introduction language for learning how to program.
- MATLAB provides a “quick-and-easy” development environment.
- MATLAB is very useful in many engineering contexts.
- MATLAB is used in industry.

## **Programming with MATLAB**

Programming is a **transferable skill**. Basic programming concepts are common to almost all programming languages. The syntax may change but is usually similar. Learning to program in MATLAB will make it much easier to pick up just about any other language.

MATLAB is platform independent. You can write the software once for several different operating systems including both Windows and Mac platforms. If you rely only on the core functionality of MATLAB you will also be able to run your programs under Linux, using Octave.

MATLAB can be linked to other software. For example it can be used with programs written in other popular languages such as C/C++, Java or Fortran.

## **MATLAB in your degree**

In Mathematical Modelling 2 and 3 you will need to use MATLAB to solve applied mathematical models. Other courses in structural analysis, electrical circuits and systems and control also use MATLAB.

In MM2, MM3 and many other courses you may like to use MATLAB to check your calculations and plot results.

Many students use MATLAB to write programs for their 4<sup>th</sup> year project.

## **MATLAB is a marketable skill**

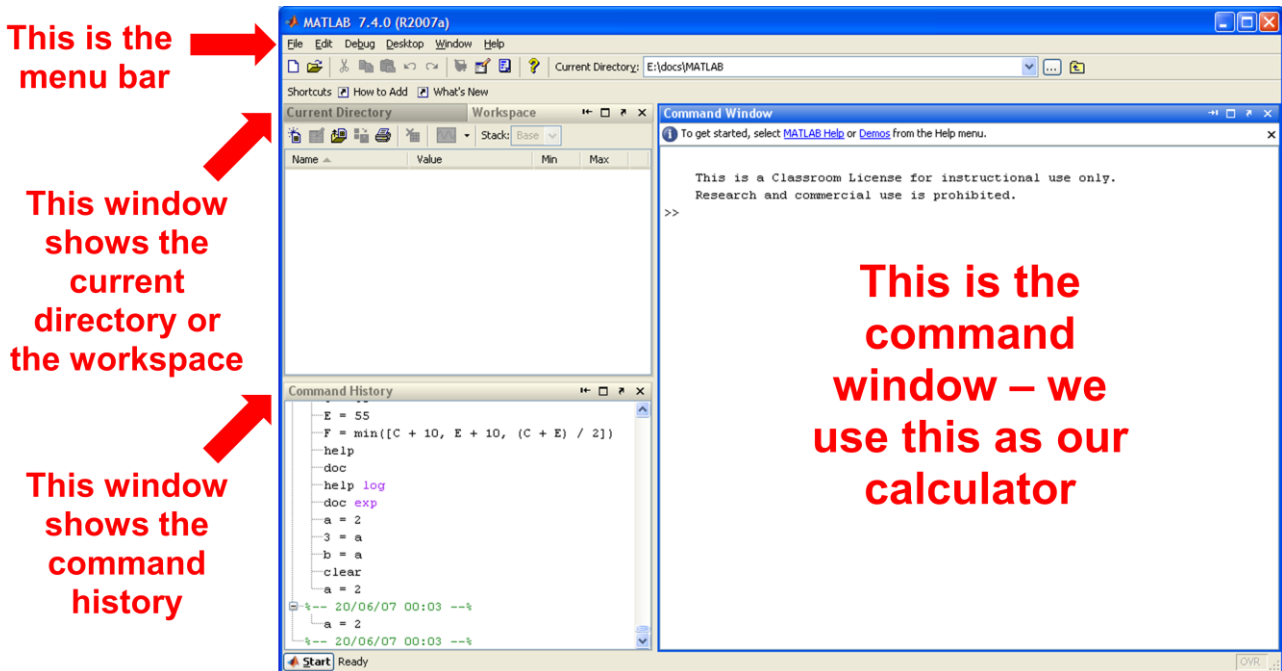
Engineering job advertisements can and do mention MATLAB as a required skill. Computer programming experience is also often mentioned as a required or desirable skill.

# MATLAB as a calculator

MATLAB can be used in a wide range of ways to help you solve engineering problems. We will begin by using MATLAB as an advanced calculator. In particular we will learn:

- To express mathematics in a form suitable for MATLAB.
- To use built-in mathematical functions in calculations.
- To use variables in calculations

This is the window that appears when you start MATLAB.



## Entering commands

You can enter expressions at the command line in the command window and evaluate them right away.

Previous command

Next command

```
>> 3 + 5 * 8
ans =
    43
>> 3*(4+2)
ans =
    18
```

**The >> symbols indicate the command prompt, where commands are typed.**

## Mathematical Operators

Operator	MATLAB	Algebra
+	+	$5 + 4 = 9$
-	-	$5 - 4 = 1$
×	*	$5 * 4 = 20$
÷	/	$5 / 4 = 1.25$
$a^b$	$a^b$	$5^4 = 625$

## Order of operations

MATLAB follows the usual BEDMAS order of operations when performing calculations.

B = Brackets	>> 3*4 + 2
E = Exponentials	ans =
D = Division	14
M = Multiplication	>> 3*(4+2)
A = Addition	ans =
S = Subtraction	18

Be careful using brackets - check that opening and closing brackets are matched up correctly.

## Built-in functions

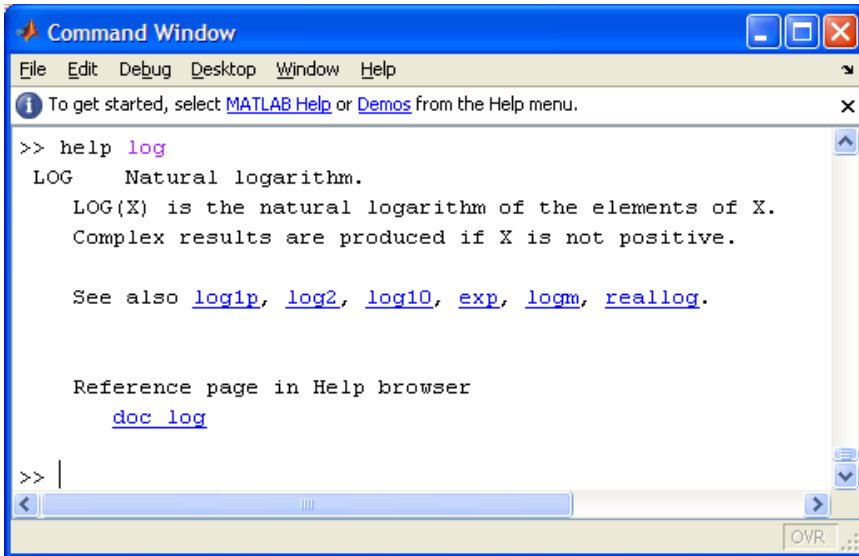
Like a calculator, MATLAB has many built-in mathematical functions. The MATLAB command reference at the rear of this manual lists some of the functions available. To call a function is simple:

```
>> sqrt(4)
ans =
    2
>> abs(-3)
ans =
    3
```

Note the use of brackets around the function's input argument. There should be no spaces between the function name and the opening bracket. Function names are case sensitive and **all built-in functions should be called using lower case letters**, even though MATLAB help files show the function name in capitals. You can search for and find out more about the available functions by using MATLAB's help.

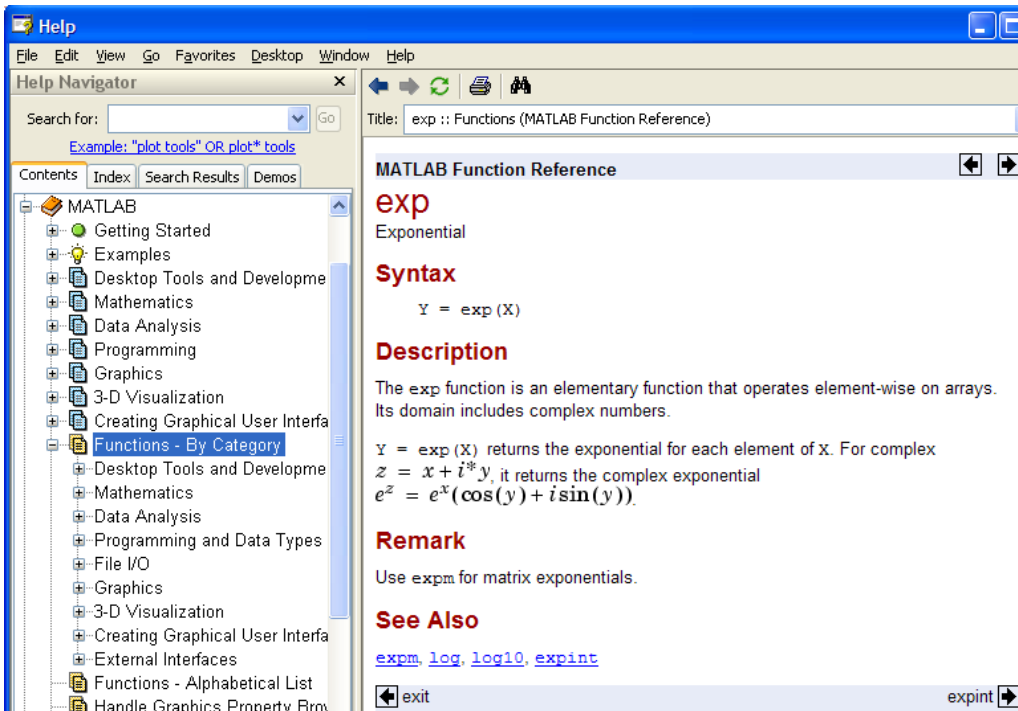


>> help                    gives command line help



>> doc                    gives GUI help

>> doc exp



# Variables

---

In algebra we use variables so calculations are easily represented. We can also use variables with MATLAB.

## Algebra

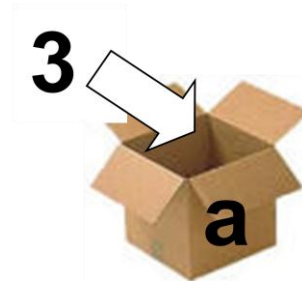
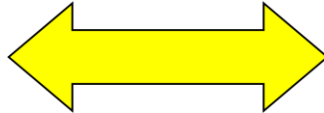
$$c = (f - 32) \times \frac{5}{9}$$
$$f = 100 \Rightarrow c = 37.8$$
$$f = 32 \Rightarrow c = 0$$

## MATLAB

```
>> f = 100
f =
    100
>> c = (f-32)*5/9
c =
    37.7778
>> f = 32
f = 32
>> c = (f-32)*5/9
c =
    0
```

You can think of variables as *named locations in the computer's memory in which a number can be stored*. It may help to think of your computer memory as a large set of “boxes” in which numbers can be stored. Boxes can be labeled with a variable name and the values placed in the boxes can be inspected and changed.

```
>> a = 3
a =
    3
```



## Assigning variables a value

The equals sign is used to assign values to variables. Variable assignment either creates the variable OR, if it already exists, changes the variable value.

```
>> a = 2
a =
    2
>> 3 = a
??? 3 = a
    |
Error: ...
```

Assignment is always left to right:

```
>> a = some expression
```

## Naming variables

Variable names may only use alphanumeric characters and the underscore. They should not begin with a number and cannot include spaces.

If using a single letter to identify a variable then use a lower case name for scalar values. We will reserve upper case letters for matrices, to be consistent with standard mathematical notation.

It is a good programming practice to give variables descriptive names, so that it is possible to tell at a glance what a variable contains. If using word(s), the first letter should be lower case, and the first letter of subsequent words should be upper case:

```
numberOfStudents = 570;  
taxRate = 0.33;
```

This makes variable names easier to read.

## Special variables

MATLAB has some special variables:

- **ans** is the result of the last calculation
- **pi** represents  $\pi$
- **Inf** represents infinity
- **NaN** stands for not-a-number and occurs when an expression is undefined e.g. division by zero
- **i, j** represent the square root of -1 (necessary for complex numbers)

## Calculations with variables

Suppose we want to calculate the volume of a cylinder.

Its radius and height are stored as variables in memory with sensible names. We can then calculate the volume in MATLAB as follows:

```
>> volume = pi * radius^2 * height
```

Visually:



# Script Files

---

You can save a sequence of commands so that you may reuse them. To save commands we place them in a script file. When a script file is run each line of the file is executed in turn. This has the same effect as typing each line in the command window.

Script files need to be saved with the file extension “.m”. Script file names can only use letters, numbers and the underscore character. Eg *vol\_surf.m* is a valid name.

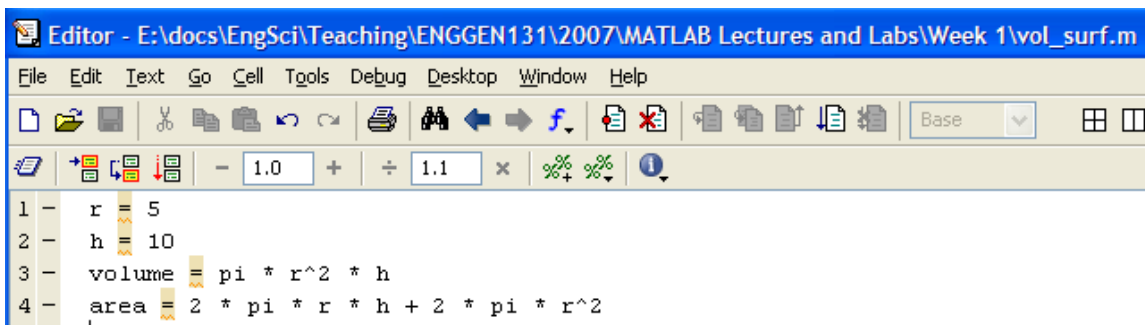
There are some important limitations to be aware of when naming your files.

## File names should NOT:

- include spaces
- start with a number
- use the same name as an existing command

If you do any of the above you will get unusual errors when you try and run your script.

MATLAB includes an editor for working on script files:



```
Editor - E:\docs\EngSci\Teaching\ENGEN131\2007\MATLAB Lectures and Labs\Week 1\vol_surf.m
File Edit Text Go Cell Tools Debug Desktop Window Help
[Icons] Base [Grid]
[Icons] - 1.0 + ÷ 1.1 × %>% %>% ⓘ
1 - r = 5
2 - h = 10
3 - volume = pi * r^2 * h
4 - area = 2 * pi * r * h + 2 * pi * r^2
```

Once you have saved a script file you can run the sequence of commands from the file by typing the filename, without the “.m” extension.

Eg

```
>> vol_surf
r =
    5
h =
   10
volume =
   785.3982
area =
   471.2389
>>
```

# Commenting

---

Lines that start with the % character are not executed by MATLAB. This means you can add comment lines to your file which explain the purpose of various commands. Comments are VERY IMPORTANT. They help people understand what the code is doing and why.

Your code may make sense to you now but it is important to realise that you may not be the only person who needs to understand your code. Most code written in the real world will need to be modified by either yourself or someone else at some point in the future. These modifications are usually done to improve performance, add extra functionality or fix errors. A program written months or years ago can be very hard to understand without good commenting, even when you were the person who originally wrote it!

All script files MUST be commented. Lab tasks will NOT be checked off if you have not commented your files.

## Header comments

Every script file should have a header comment at the top of the file which indicates the purpose of the file and who wrote it.

```
% ConvertTemp.m converts the freezing and boiling points for  
% water from degrees Celsius (c) to Fahrenheit (f)  
% Author: Peter Bier
```

Including the author's name is important as if someone else is trying to use your code they then know who to ask for help. MATLAB uses this header comment to generate help. Typing help and your filename will display your header comments. Eg

```
>> help ConvertTemp  
  
ConvertTemp.m converts the freezing and boiling points for  
water from degrees Celsius (c) to Fahrenheit (f)  
Author: Peter Bier
```

## Other comments

You should add comments to any sections of code which are not straight forward to understand. This may include comments that explain what is being done or relate to how to use MATLAB functions you are not very familiar with. Aim to write useful comments.

The following is an example of useless commenting and poor choice of variable names:

```
% set x to zero  
x = 0  
% calculate y  
y = x * 9/5 + 32
```

The above comments do not really help you understand anything about the code. We can already see that x is being set to zero and that y is being calculated. We are also given no clues what x and y represent.

The following is far more useful and actually helps a reader understand what the code is doing:

```
% Convert freezing point of water from celsius to Fahrenheit
c = 0
f = c * 9/5 + 32

% Convert boiling point of water from celsius to Fahrenheit
c = 100
f = c * 9/5 + 32
```

From these comments it is obvious what the variables c and f are meant to represent and it is clear what the calculation of f is achieving.

The complete script file is as follows:

```
% ConvertTemp.m converts the freezing and boiling points for
% water from degrees Celsius (c) to Fahrenheit (f)
% Author: Peter Bier

% Convert freezing point of water from celsius to Fahrenheit
c = 0
f = c * 9/5 + 32

% Convert boiling point of water from celsius to Fahrenheit
c = 100
f = c * 9/5 + 32
```

## Basic user interaction: Input/Output

---

Script files are more useful if they can interact with users, prompting them to enter values and then displaying relevant results. We want to be able to get *input* from the user and then display the appropriate *output*.

This is easy to do using the built-in `input` and `disp` commands.

```
% CalculateRectArea.m calculates the area for
% a rectangle with dimensions provided by the user
% Author: Peter Bier

% prompt user to enter height and width
height = input('Enter the height of the rectangle: ')
width = input('Enter the width of the rectangle: ')

% note the semi-colon on the end of the line
% this supresses the output of the command, the command is
% executed but the result of the calculation is not displayed
area = height * width;

disp('The area of the rectangle is')
disp(area)
```

Note that the input function takes as an argument a **string of characters** enclosed in single quotes. This string of characters could include words, numbers and punctuation characters. When MATLAB executes an input command the entire contents of this string are printed to the screen and then the computer will wait for the user to type something and press enter.

The disp function can either take a string of characters enclosed in single quotes OR a variable. If it is called with a string of characters the characters enclosed in quotes are printed to the screen. If it is called with a variable then the value of the variable is displayed.

When this program is run it produces the following output:

```
>> CalculateRectArea
Enter the height of the rectangle: 3
height =
3
Enter the width of the rectangle: 4
width =
4
The area of the rectangle is
12
```

## Chapter 1 Summary Program

---

We can now write commented simple script files that get input from a user, use MATLAB as a calculator and then display results back to the user.

For example, the following program calculates the final result for a 131 student based on their course work and exam marks.

```
% Calculate131Result.m calculates the final result for a
% student with a course work and final exam mark
% entered by the user
% Author: Peter Bier

% get course and exam mark from user
c = input('Please enter the course work mark: ')
e = input('Please enter the exam mark: ')

courseAverage = 1 / 2 * (c + e)

% Neither coursework nor exam mark may raise the average by
% more than 10 percent, so the course work and exam marks
% set an upper limit on the possible mark that can be achieved

% calculate upper limits
cmax = c + 10
emax = e + 10

% the final mark cannot be above either of the upper limits,
% so it will be the minimum of the course average and the
% two upper limits
% We use the min function to calculate the minimum of these
% three values. The input argument for the min function is
% a list of values enclosed in square brackets and separated
% by commas.
% note we have suppressed the output of the min function
% calculation with a semi-colon on the end
final_mark = min([courseAverage, cmax, emax]);

disp('Final mark is ')
disp(final_mark)
```



An example of running this program is shown below (user input is in bold):

```
>> Calculate131Result
Please enter the course work mark: 38
c =
38
Please enter the exam mark: 62
e =
62
courseAverage =
50
cmax =
48
emax =
72
Final mark is
48
```

# Chapter 2: 1D Arrays, Problem Solving

In this chapter we introduce the basics of working with 1D arrays. An array is a variable that can hold multiple values.

Recall that the reason we are learning how to develop software is so that we may write computer programs to solve problems. When confronted with a new problem it can be tricky to know how to approach it so we will outline a simple five step method that can be used to help develop a computer program to solve a problem.

## Learning outcomes

After working through this chapter, you should be able to:

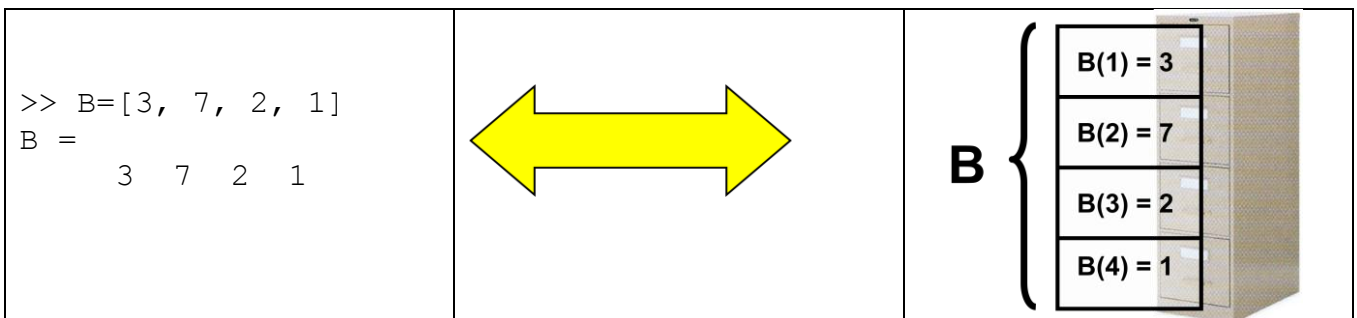
- Explain the concept of a 1D array
- Create and manipulate 1D arrays
- Draw plots of 1D arrays
- Use 1D arrays in programs
- Outline the five steps for problem solving
- Use the five steps to solve a problem

## 1D arrays

So far we have dealt with MATLAB variables that hold a single value. We can also create MATLAB arrays that hold multiple values.

Arrays are useful for storing lists of values (1D arrays) or tables of values (2D arrays). They are also ideal for representing vectors (1D arrays) and matrices (2D arrays)

If a scalar variable (for a single value) is like a *cardboard box*, then a 1D array variable is like a *filing cabinet*. Each drawer of the filing cabinet can store a value.



## Creating 1D arrays

To create an array we can assign a list of values to a variable. The values need to be enclosed by square brackets and separated by a comma or a space.

```
>> dailyHighs = [10, 11, 13, 12, 19, 18, 17]
dailyHighs =
10  11  13  12  19  18  17

>> dailyLows = [3 2 4 1 5 6 4]
dailyLows =
3  2  4  1  5  6  4
```

## Accessing array elements

You can access or change a particular array element using round brackets.

```
>> dailyHighs
dailyHighs =
10  11  13  12  19  18  17
>> dailyHighs(2)
ans =
11
>> dailyHighs(2) = 14
dailyHighs =
10  14  13  12  19  18  17
```

## Extending arrays

You can add extra elements by creating them directly using round brackets or by concatenating them (adding them onto the end).

```
>> dailyHighs
dailyHighs =
10  14  13  12  19  18  17
>> dailyHighs(8) = 12
dailyHighs =
10  14  13  12  19  18  17  12
>> dailyHighs = [dailyHighs, 14]
dailyHighs =
10  14  13  12  19  18  17  12  14
```

## Default array elements

If you don't assign array elements, MATLAB gives them a default value of 0

```
>> dailyHighs
dailyHighs =
10 14 13 12 19 18 17 12 14
>> dailyHighs(12) = 10
dailyHighs =
10 14 13 12 19 18 17 12 14 0 0 10
```

## Using arrays in programming

The main use for arrays in programming is data storage. Rather than creating a large number of variables to store related values, we can use a single array to store the values.

Some examples of where you would use an array for data storage are as follows:

- keeping track of the trajectory of a basketball
- storing the stress along a beam
- storing pressures inside the heart

## Using Arrays in MATLAB

MATLAB was originally written for use with arrays and subsequently it is very good at dealing with them. MATLAB provides lots of special array functionality to make it easier to create and manipulate arrays. Using arrays and MATLAB functions allows repetitive calculations to be done quickly. It also allows us to write compact programs.

## Automatic 1D Arrays

There are a number of ways to create 1D arrays automatically. The colon operator and the **linspace** function are particularly useful

Using the colon operator, you can specify an array that contains a sequence of increasing or decreasing values. A start and stop value are required.

```
>> x = 0:10
x =
    0    1    2    3    4    5    6    7    8    9   10
```

The colon operator creates an array of elements beginning with the start value and ending with the stop value. By default the values will increase by a step size of 1. You can specify a different step size if you prefer as follows:

```
>> x = 0:2:10
x =
    0    2    4    6    8   10
```

The `linspace` function is a convenient function for creating an array that contains a sequence with a specific number of elements. The input arguments for the `linspace` function are the start value, stop value and number of elements to create. When passing inputs to a function the order is important, so you must pass in first the start value then the stop value and finally the number of elements:

```
>> t = linspace(0,10,7)
t =
    0    1.6667    3.3333    5.0000    6.6667    8.3333   10.0000
>>
```

The above command has created a list of 7 points spaced evenly between 0 and 10.

### **Array Slicing**

It is possible to access several elements of an array at once using array slicing. Instead of using a single value to index the array we can use another array. For example to pull out the 2<sup>nd</sup>, 4<sup>th</sup> and 6<sup>th</sup> elements of the `dailyHighs` array we can do the following:

```
>> dailyHighs
dailyHighs =
10 14 13 12 19 18 17 12 14 0 0 10
>> dailyHighs([2,4,6])
dailyHighs =
14 12 18
```

The colon operator can be particularly handy when you wish to pull out a sequential slice of an array:

```
>> dailyHighs
dailyHighs =
10 14 13 12 19 18 17 12 14 0 0 10
>> dailyHighs(3:5)
dailyHighs =
13 12 19
```

In the above example we are displaying the 3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> elements, as the colon operator has been used to create the array `[3, 4, 5]`.

### **Array Arithmetic**

Arrays of the same length can be added or subtracted to each other. Arrays can also be multiplied by scalar constants.

```
>> dailyHighs = [10, 11, 13, 12, 19, 18, 17];
>> dailyLows = [ 3,  2,  4,  1,  5,  6,  4];
>> dailyRange = dailyHighs - dailyLows
dailyRange =
    7    9    9   11   14   12   13
>> dailyAverage = 0.5 * (dailyHighs + dailyLows)
dailyAverage =
    6.5    6.5    8.5    6.5   12   12   10.5
```

It is possible to multiply the elements in one array by the corresponding elements in another array. Here is an example of element by element multiplication:

```
>> heights = [9, 8, 4, 6];
>> widths = [3, 2, 1, 5];
>> areas = heights .* widths
areas =
    27    16     4    30
```

Note the use of the dot before the multiplication symbol. The dot operator means MATLAB will perform element by element multiplication rather than attempting matrix multiplication.

Element by element division is also available:

```
>> heights = [9, 8, 4, 6];
>> widths = [3, 2, 1, 5];
>> ratios = heights ./ widths
ratios =
     3     4     4     1.2
```

It is also possible to raise every number in an array by a particular power, using element by element exponentiation.

```
>> heights = [9, 8, 4, 6];
>> square = heights.^2
square=
    81    64    16    36
```

Again notice the use of the dot operator.

A common error is to forget to use a dot when manipulating arrays.

If you leave the dot off then MATLAB will attempt to do Matrix multiplication or exponentiation and will generate the error:

The error is because MATLAB is attempting a matrix multiplication of the two vectors, which is not valid.

```
>> heights = [9, 8, 4, 6];
>> widths = [3, 2, 1, 5];
>> areas = heights * widths
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

## Array Functions

Standard mathematical functions (sin, cos, exp, log, etc) can apply to arrays as well as scalars. This has the same effect as applying the function to each of the elements in turn.

```
>> x = [1, 2, 3];  
>> y = sin(x);
```

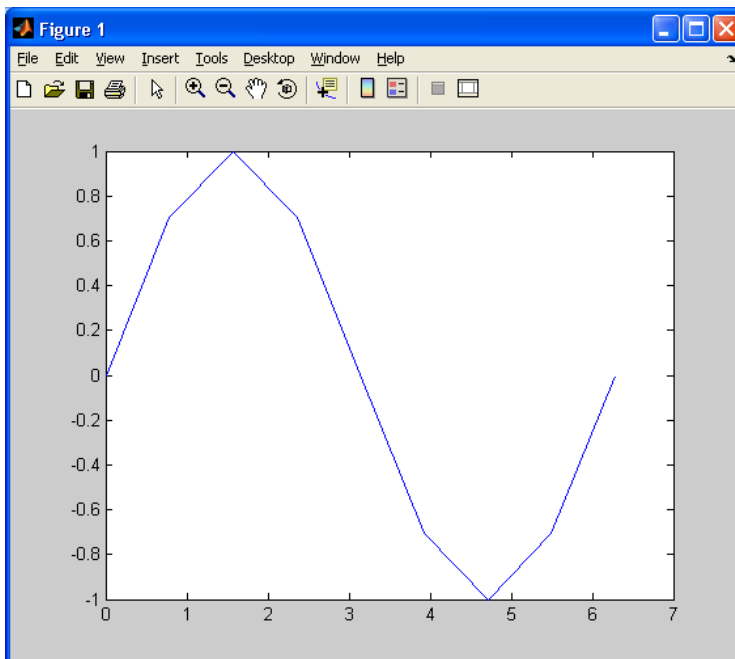
y is now [sin(1), sin(2), sin(3)]

When writing functions later in the course remember that your input variables might be arrays

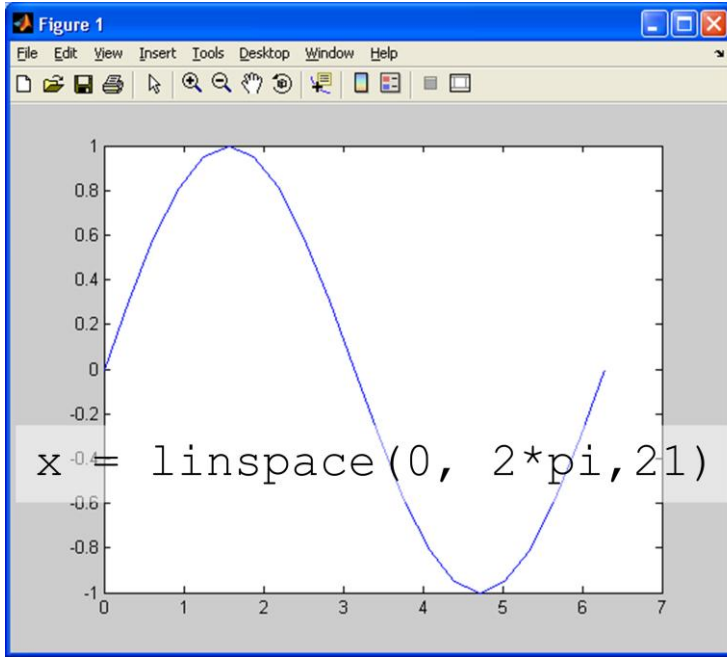
The following example shows how we can use the sin function on an array to produce a plot of this function.

```
>> x = linspace(0, 2*pi, 9)  
x =  
0 0.7854 1.5708 2.3562 3.1416 3.9270 4.7124 5.4978  
6.2832  
>> y = sin(x)  
y =  
0 0.7071 1.0000 0.7071 0.0000 -0.7071 -1.0000 -0.7071  
0.0000  
>> plot(x, y)
```

The result of running this code is the following plot:

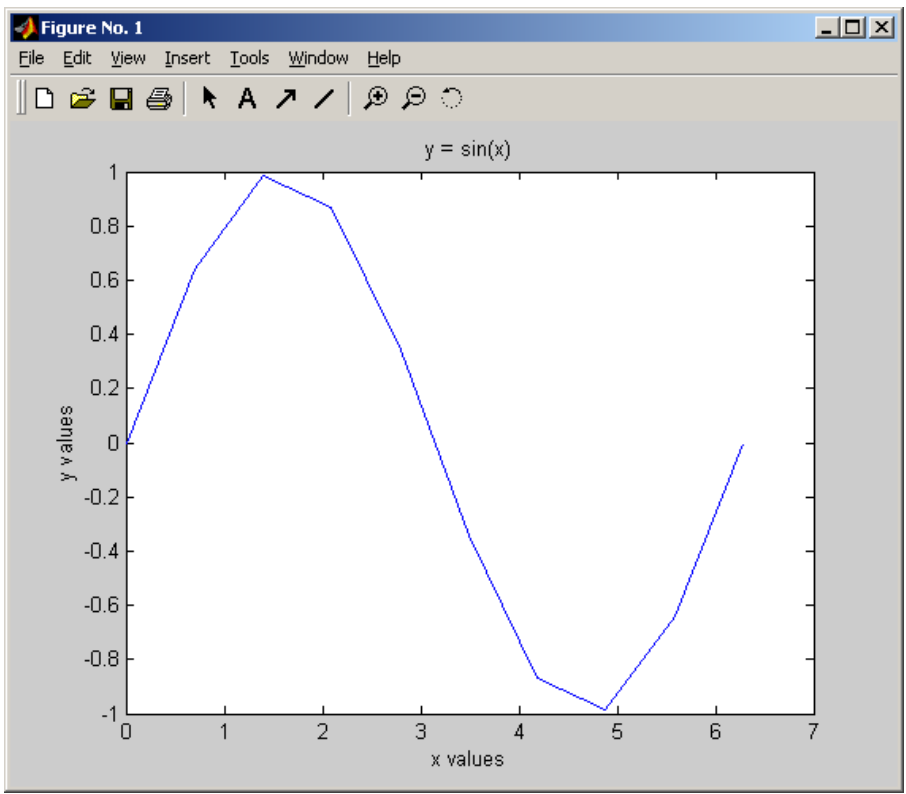


Notice that the curve is not very smooth as we have only used 9 points. If we increase our number of points we get a smoother curve:



We have been a bit naughty leaving our graph unlabelled. You should give all graphs a title and label the axes. Fortunately this is very easy in MATLAB. We simply use the title, xlabel and ylabel functions, passing in the labels between single quotes:

```
>> title('y = sin(x)');
>> xlabel('x values');
>> ylabel('y values');
```





## **Special Array Functions**

Some functions are specialised for use with 1D arrays:

- `length(array)` gives the number of elements in array
- `min(array)` gives the minimum value in array
- `max(array)` gives the maximum value in array
- `sum(array)` gives the sum of values in array

## Five steps for problem solving

---

There are many different problem solving methodologies available. The following five steps provide a simple framework which can help you approach a problem.

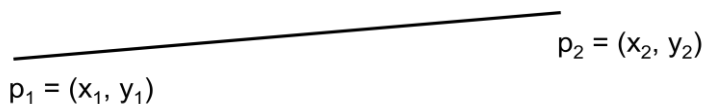
1. State the problem clearly
2. Describe the input and output information
3. Work the problem by hand (or with a calculator) for a simple set of data
4. Develop a solution and convert it to a computer program
5. Test the solution with a variety of data

We will be using these five steps through out this course.

## Problem solving worked example

---

We want to compute the distance between two points in a plane



### **Step 1: State the problem clearly**

Compute the straight-line distance between two points in a plane.

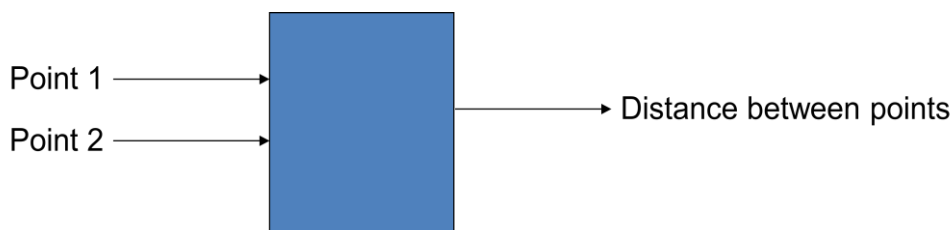
### **Step 2: Describe the input and output information**

Our inputs are the information given that we require to solve the problem. Note that sometimes we will be given irrelevant information, so not all given information may be required.

Our outputs are the values we need to compute.

It is often helpful to draw an I/O diagram.

I/O = Input/Output



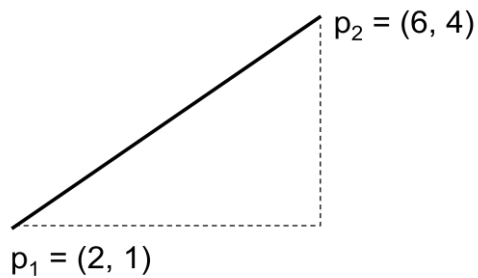
### Step 3: Work the problem by hand

Working the problem by hand is a very important step. Use a calculator if necessary.

Don't skip this step, even for a simple problem. If you cannot do this step:

- Read the problem again
- Consult reference material
- Diagrams can be useful

Working the problem by hand will help you understand what steps need to be taken to solve the problem. It will also give you a known solution value for a simple data set, which you can use later to test your program.



$$\begin{aligned}\text{distance} &= \sqrt{(\text{side}_1)^2 + (\text{side}_2)^2} \\ &= \sqrt{(6-2)^2 + (4-1)^2} \\ &= \sqrt{4^2 + 3^2} = \sqrt{16+9} = \sqrt{25} \\ &= 5\end{aligned}$$

#### **Step 4: Develop a solution and convert it to a computer program**

Decompose the problem into a set of steps and write pseudocode or a flowchart for code. Then write the code.

Simple problems give simple steps. Complex problems give complex steps.

If we are dealing with a complex problem we still decompose the problem into a series of steps. Each complex step may also require the problem solving process. We will discuss how to create pseudocode and flowcharts for complex problems later in the course.

#### **Pseudocode**

1. Get x- and y-values for two points
2. Compute length of two sides of right angle triangle generated by points
3. Use hypotenuse calculation to get distance
4. Display the distance

```
% CalculateDistance.m calculates the distance between two
% points p1 and p2 on a plane
% Author: Peter Bier

% get x and y values for two points
x1 = input('Please enter the x coord of point 1: ');
y1 = input('Please enter the y coord of point 1: ');
x2 = input('Please enter the x coord of point 2: ');
y2 = input('Please enter the y coord of point 2: ');

% compute length of two sides of right angle triangle
% generated by points
side1 = x2 - x1;
side2 = y2 - y1;

% Use hypotenuse calculation to get distance
d = sqrt(side1^2 + side2^2);

% Display the distance
disp('The distance between the two points is');
disp(d);
```

#### **Step 5: Test the solution with a variety of data**

Test using hand worked example. Also test with other data. We need to verify that our code works correctly by testing it with a range of data.

```
>> CalculateDistance
Please enter the x coord of point 1: 2
Please enter the y coord of point 1: 1
Please enter the x coord of point 2: 6
Please enter the y coord of point 2: 4
The distance between the two points is
    5
```

```
>> CalculateDistance
Please enter the x coord of point 1: 0
Please enter the y coord of point 1: 0
Please enter the x coord of point 2: 1
Please enter the y coord of point 2: 1
The distance between the two points is
    1.4142
```

## Chapter 2 Summary Program

---

We can now write programs that use 1D arrays and apply the five steps of problem solving.

For example, the following program calculates an approximate value for the integral:

$$\int_0^{\pi/4} \sin^2(x) - \cos^2(x) dx$$

by using the rectangle method.

```
% IntegrateWithRectangleMethod.m calculates an approximate
% value for the integral of sin(x)^2 - cos(x)^2,
% between the limits 0 and pi/4.
% The rectangle method is used,
% ie A = h(y_1 + y_2 + ... + y_(n-1) )
% Author: Peter Bier

% get number of points to use to represent the function over
% the specified range
n = input('Please enter the number of points to use: ');

% set up a range of x values from 0 through to pi/4
x = linspace(0,pi/4,n);

% calculate the value of the function for each x value
y = sin(x).^2 - cos(x).^2;

% determine the width of the rectangles we will use to
% approximate the area
h = x(2) - x(1);

% calculate area of all the little rectangles
% notice we drop the last y value by using array slicing
rectangles = h * y(1:n-1);

% sum all the little rectangles to get our area approximation
area = sum(rectangles);

% display the area value
disp('The approximate value of the definite integral is');
disp(area);
```

Some examples of this program running are given below (user input is in bold):

```
>> IntegrateWithRectangleMethod  
Please enter the number of points to use: 10  
The approximate value of the definite integral is  
-0.54236
```

```
>> IntegrateWithRectangleMethod  
Please enter the number of points to use: 1000  
The approximate value of the definite integral is  
-0.50039
```

It can be shown analytically that the solution is  $-0.5$ , so our approximation is pretty good.

# Chapter 3: Functions, Problem Solving and Debugging

MATLAB has many built-in functions, such as the trigonometric functions and the functions for processing arrays that we have already met. The MATLAB command reference appendix lists more of the built-in functions available.

In addition we can define our own functions in a function file and use them in just the same way as the built-in functions. Writing functions is a key part of software development and will allow you to write code that is shorter, easier to understand and easier to maintain.

## Learning outcomes

After working through this chapter, you should be able to:

- Explain the concept of a function
- Call functions from your own programs
- Define your own functions
- Examine the function and command workspaces
- Debug script files and functions

## What is a function?

A function is one of the basic building blocks of software development. Generally a function takes some input value(s), processes them and returns some output value(s).

We are already familiar with mathematical functions, which take an input value and transform it into an output value. You will have encountered the following notation for mathematical functions:

$$y = f(x)$$

The function  $f$  takes an input value  $x$  and returns an output value  $y$ . We will also refer to the input value as the **argument** of the function  $f$ .

As we have already seen, MATLAB functions use the same notation when called.

Mathematical functions:	MATLAB functions:
$y = \sin(x)$	<code>y = sin(x)</code>
$y = \ln(x)$	<code>y = log(x)</code>
$y = \sqrt{x}$	<code>y = sqrt(x)</code>

Here are a few more examples of some functions you have already encountered:

```
largestValue = max(x)
y = exp(x)
a = linspace(-pi, pi, 10)
```

Note that the **linspace** function takes not one but **three input arguments**. It is possible for a function to have many input arguments. Depending on the function the input argument(s) may be scalars or arrays. It is also possible for a function to have no input arguments, eg the following function can be called to return a random number between 0 and 1:

```
x = rand()
```

### **Why use functions?**

There are several excellent reasons for using functions.

**Functions enable us to use a “divide and conquer” strategy.** A complex programming task can be broken into smaller manageable tasks, with a function written for each task. The functions do not even need to be written by the same person, allowing several people to work on the same project at the same time.

**Functions allow us to reuse code.** The same function may be useful for many problems. Rather than repeating the same lines of code several times in order to do a common task, we can write a function to do that task and then simply call it. This not only saves us from writing lots of extra code, it enables us to write easier to understand programs.

**Functions make code easier to maintain.** A function has well defined behaviour. We know that given certain inputs it should return certain outputs. It is easy to check that the right outputs are being returned for possible inputs. If there is a problem with our code we can test each function to help us pinpoint the piece of problem code.

**Functions allow us to hide implementation.** Once a function has been written we don't need to look at its code to use it. The only interaction is via inputs and outputs. How the function is written (the implementation) is hidden inside the function.

### **Behaviour of a function**

- Functions should be well commented (users must be able to find out how a function works)
- Functions should be well defined (given inputs should give known outputs)
- Functions should be well tested (inputs should always give correct outputs)



## Calling functions

We can call functions from the command line or a script file. In either case they are called in the same way. To call a function we need to know the name of the function, what input(s) it takes and what output(s) it returns. Inputs can be either numbers or variables. Eg:

<pre>y = sin(3);  x = 3; y = sin(x);</pre>	<pre>y = min([3, 5, 1])  a = [3, 5, 1] y = min(a)</pre>
--	---

For built-in functions the information we need to call the function can be found using the MATLAB help. This is very handy when we meet a new function and need to learn how to use it for the first time.

Let's look at the meshgrid function we will be using later in the course:

```
Command Window
File Edit Debug Desktop Window Help
>> help meshgrid
MESHGRID X and Y arrays for 3-D plots.
[X,Y] = MESHGRID(x,y) transforms the domain specified by vectors
x and y into arrays X and Y that can be used for the evaluation
of functions of two variables and 3-D surface plots.
The rows of the output array X are copies of the vector x and
the columns of the output array Y are copies of the vector y.
[X,Y] = MESHGRID(x) is an abbreviation for [X,Y] = MESHGRID(x,x).
[X,Y,Z] = MESHGRID(x,y,z) produces 3-D arrays that can be used to
evaluate functions of three variables and 3-D volumetric plots.
For example, to evaluate the function x*exp(-x^2-y^2) over the
range -2 < x < 2, -2 < y < 2,
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
MESHGRID is like NDGRID except that the order of the first two input
and output arguments are switched (i.e., [X,Y,Z] = MESHGRID(x,y,z)
produces the same result as [Y,X,Z] = NDGRID(y,x,z)). Because of
this, MESHGRID is better suited to problems in cartesian space,
while NDGRID is better suited to N-D problems that aren't spatially
based. MESHGRID is also limited to 2-D or 3-D.
Class support for inputs X,Y,Z:
float: double, single
See also surf, slice, ndgrid.
Reference page in Help browser
doc meshgrid
```

function input

function output

After reading through the help we now have some idea of how the meshgrid function works. We could call the meshgrid function to set up some arrays for 3D plotting. We know it takes two inputs and returns two outputs. Here is an example of a call to meshgrid:

```
xRange = -2:.2:2
yRange = -2:.2:2
[X, Y] = meshgrid(xRange, yRange)
```

Note that the name of the function is in lower case even though the help shows it written in uppercase. Function names are case sensitive, which means that meshgrid(x,y), meshGrid(x,y) and MESHGRID(x,y) would be interpreted as three different functions.

Input argument(s) are passed to the function by placing them inside the parentheses following the function name. If there is more than one argument we separate them by commas. Note that there is no space between the function name and the opening parentheses. Also note that the order of the input arguments is very important. When calling a function with several inputs it is the order of the arguments that is used to tell which is which, rather than the name. The input arguments do NOT need to have the same name as those shown in the help file.

We usually assign the output of a function to a variable so that it can be used. Some functions return more than one output value. If more than one variable is returned we assign the output variables by using a list of variables inside square brackets. The output arguments do NOT need to have the same name as those shown in the help file.

If there is only one output we can omit the square brackets:

```
y = atan(x)
a = atan(0.5)
```

**IMPORTANT:** If a function returns several outputs but you forget the square brackets and only assign the result to one variable, it will contain only the first output returned. The other outputs will be thrown away. This is a common programming error that beginner programmers make.

Some functions print information to the screen or draw images on the screen rather than returning a value. In this case there is no need to store the output of the function as a variable:

```
plot(x, y)
```

# Writing functions

Let's start by writing a very simple function that will square a number for us.

The mathematical definition of this function looks like this:

$$y(x) = x^2$$

To define this function in Matlab you could type the following into a file:

```
function y = square(x)
    y = x^2;
return;
```

The keyword “function” lets Matlab know we are writing a function. Next is the output variable, which must be assigned a value before the function returns. Following the equals sign is the function name, which is case sensitive. Following the name is the function input, in brackets.

The function “body” contains a single line that is used to calculate the output value for our given input. Finally the keyword “return” tells Matlab the function is finished.

Once this file was saved with the name square.m we can call our function, just as we would call any other Matlab functions:

```
>> fourSquared = square(4)
```

Let's have a look at a slightly more complicated example:

The screenshot shows a Matlab editor window titled "Editor - C:\Program Files\MATLAB704\work\polar\_to\_cartesian.m". The code in the editor is as follows:

```
1 function [x, y] = polar_to_cartesian(r, theta)
2     x = r .* cos(theta);
3     y = r .* sin(theta);
4     return;
5
```

Annotations with arrows point to various parts of the code:

- "function" keyword: points to the word "function" on line 1.
- output variable/s (must be assigned a value in function body): points to "[x, y]" on line 1.
- file name: points to "polar\_to\_cartesian.m" in the window title bar.
- function name: points to "polar\_to\_cartesian" on line 1.
- input variable/s or "arguments" (these are the only variables whose values the function can access): points to "(r, theta)" on line 1.
- Return statement, signifies the end of the function.: points to "return;" on line 4.
- function body (can be 1 line or 100's of lines): points to the lines between "function" and "return;" (lines 2 and 3).

The `polar_to_cartesian` function is an example of most general case of a function, as it takes multiple input arguments and produces multiple outputs. It is possible for a function to have multiple inputs, one input or no inputs. It is also possible for a function to have multiple outputs, one output or no outputs.

The first line of your file varies in each case.

### **Multiple outputs**

- No inputs            `function [o1, o2, ...] = myfunc()`
- One input            `function [o1, o2, ...] = myfunc(i1)`
- Multiple inputs      `function [o1, o2, ...] = myfunc(i1, i2, ...)`

### **One output**

- No inputs            `function [o1] = myfunc()`
- One input            `function [o1] = myfunc(i1)`
- Multiple inputs      `function [o1] = myfunc(i1, i2, ...)`

For a function with one output value the square brackets around the output argument are optional and may be left off if you desire, as was done with our `square` function.

### **No outputs**

- No inputs            `function [] = myfunc()`
- One input            `function [] = myfunc(i1)`
- Multiple inputs      `function [] = myfunc(i1, i2, ...)`

For a function with no output value the square brackets and equals sign are optional and may be left off if you desire (you need to leave off BOTH if you wish to do this).

### **Function filenames**

Functions must be saved to a file with a `.m` extension, using exactly the same filename as the function name. As with standard script files there are some important limitations when naming functions.

#### **Function names should NOT:**

- include spaces
- start with a number
- use the same name as an existing command

You may only use alphanumeric characters and the underscore when naming functions. One convention is to use underscores to separate words in a function name, eg `polar_to_cartesian`

Another popular convention to follow when naming functions is to use an upper case letter for the first letter of the name and for the first letter of subsequent words, eg `PolarToCartesian`

These conventions helps to distinguish function names from variable names. It also helps distinguish from built-in functions which are named using lower case letters only.

You may use whichever convention you prefer. In general this course manual will follow the latter convention of using an upper case letter for the first letter of the name and for the first letter of

subsequent word. We'll use the other convention occasionally too, so that you remember it. In practice it makes for easier to read code if you stick to one of the conventions.

### **Function headers**

Every function file should have a header comment at the top of the file, just beneath the function definition. This header should describe the function's input(s) and output(s), the purpose of the function and who wrote it.

```
function [f] = ConvertToFahrenheit(c)
% ConvertToFahrenheit(c) takes a temperature value c
% measured in degrees celsius and returns the equivalent
% value in Fahrenheit
% Author: Peter Bier

f = 9/5 * c + 32;

return
```

Typing help and your function name will display your header comments. Eg

```
>> help ConvertToFahrenheit

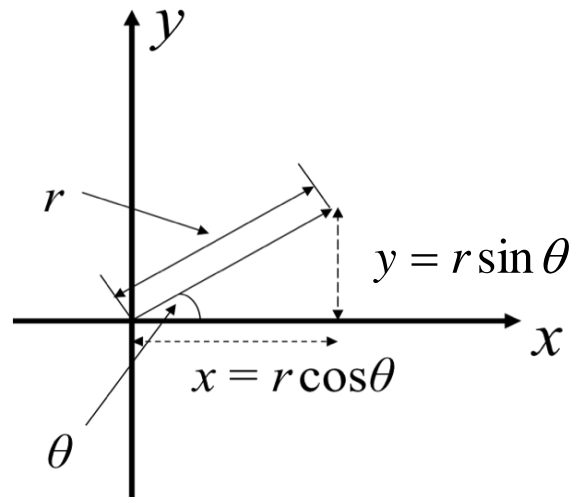
ConvertToFahrenheit(c) takes a temperature value c
measured in degrees celsius and returns the equivalent
value in Fahrenheit
Author: Peter Bier
```

## Writing functions: Polar to cartesian example

---

Let's revisit the polar\_to\_cartesian function, only this time we will develop it from scratch and include a detailed function header and comments.

Recall that polar coordinates are useful for describing circular shapes. We need to convert to Cartesian coordinates for plotting.



### Pseudocode

INPUTS:  $r$  and  $\theta$   
Calculate  $x$  value  
Calculate  $y$  value  
OUTPUTS:  $x$  and  $y$

### Code

```
function [x, y] = PolarToCartesian(r, theta)
% PolarToCartesian transforms r and theta from polar
% coordinates into (x,y) cartesian coordinates
% Inputs:  r      = radial distance
%          theta  = radial angle
% Outputs: x      = cartesian x coordinate
%          y      = cartesian y coordinate
% Author: Peter Bier

% we use the dot operator so that our code will also work
% if r and theta are arrays.
% Note the use of the semi-colon to suppress output,
% otherwise our function will print out the x and y values
% when calculating them
x = r .* cos(theta);
y = r .* sin(theta);

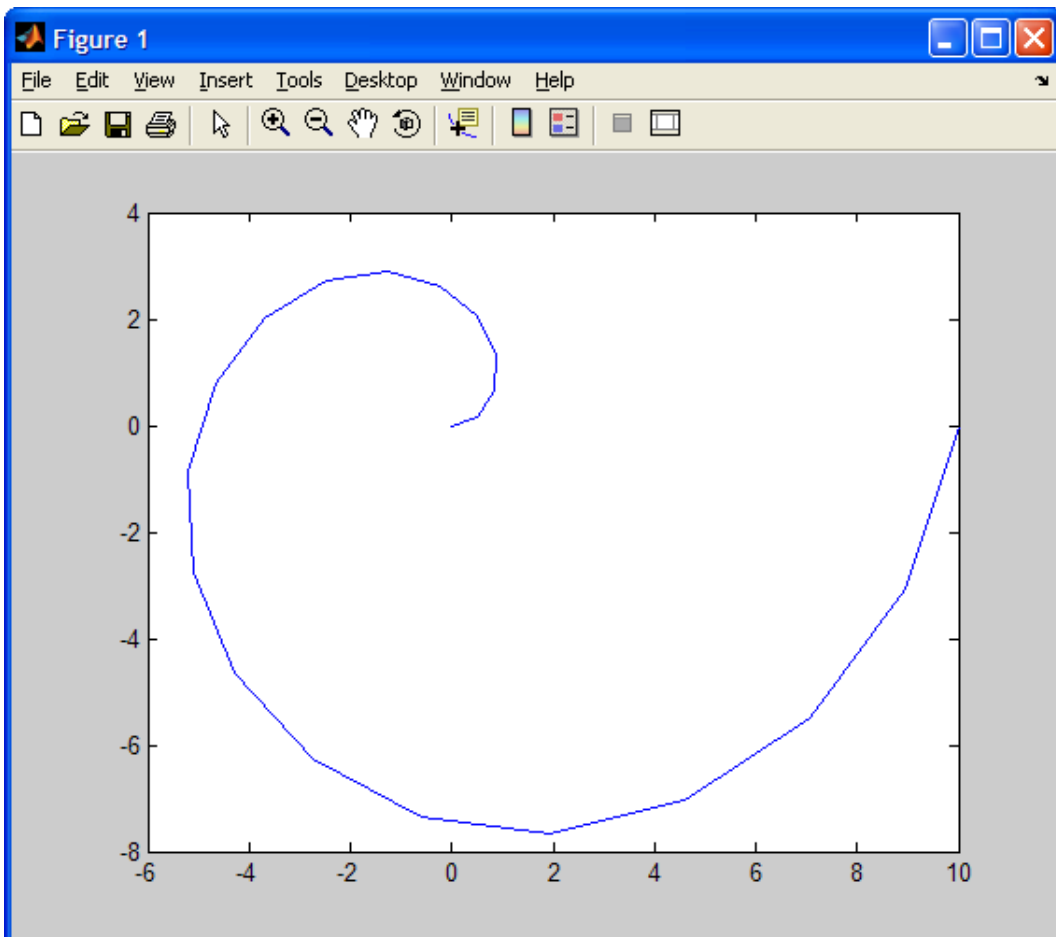
return
```

Remember that the function must be saved using the function name as the filename, eg PolarToCartesian.m

## Using our function

```
% spiral.m draws a spiral using polar coordinates.  
% Author: Peter Bier  
  
% our array of 20 radius values will range from 0 to 10  
spiralRs = linspace(0,10,20);  
% our array of 20 theta values will range from 0 to 2pi,  
% ie a full circle  
spiralThetas = linspace(0, 2*pi, 20);  
  
[x, y] = PolarToCartesian(spiralRs, spiralThetas);  
plot(x,y);
```

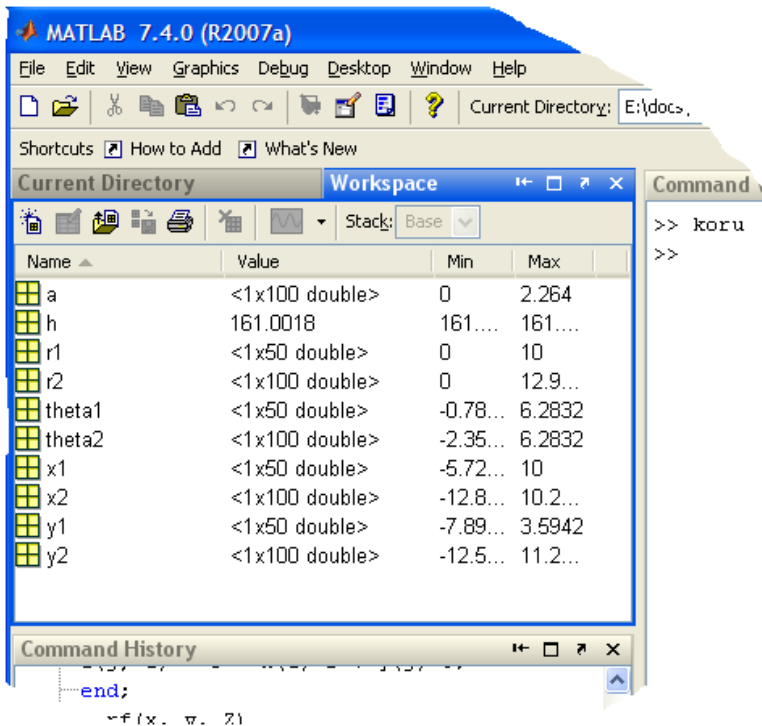
Saving this code as spiral.m and running it produces the following plot:



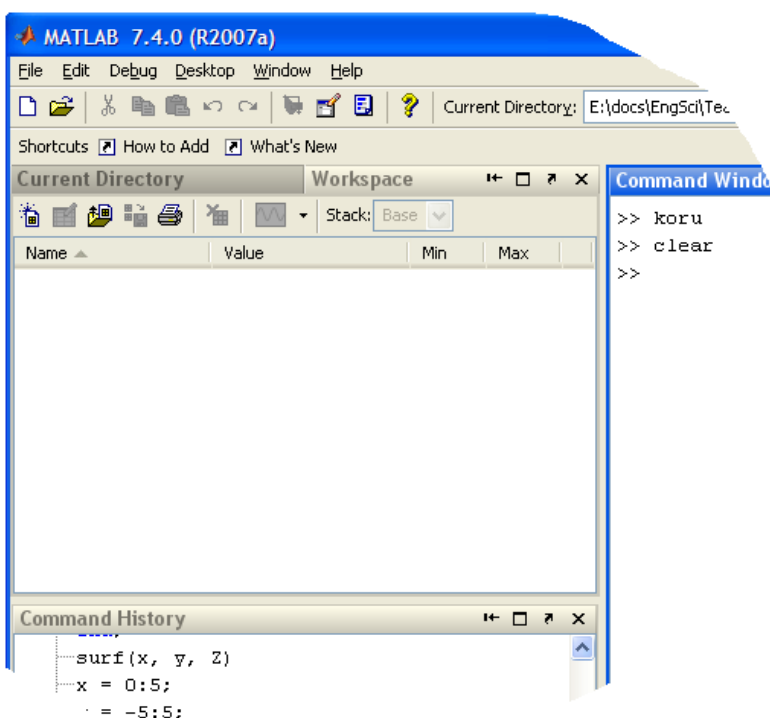
# Workspaces

## The MATLAB workspace

When you create variables in MATLAB via the command window or in script files, MATLAB stores them in the “workspace”. You can view the contents of the workspace under the Workspace tab.



The MATLAB workspace can be cleared by either restarting MATLAB or by using the `clear` command.





## **Function workspaces**

Functions create their own workspaces

Function inputs are also created in the workspace when a function starts. A function doesn't know about any variables in any other workspace, it only has access to the input values and any variables it creates.

Function outputs are copied from workspace when function ends. The MATLAB workspace knows nothing about any of the other variables a function might use in its own workspace. Function workspaces are destroyed after functions end. This means that any variables created in a function "disappear" when the function ends.

## **Debugging**

---

Often when we write a computer program it is "buggy". The program runs but it does not do what it is supposed to. The problem may be as simple as an incorrect sign in a formula or it may be quite subtle and hard to detect.

MATLAB provides a debugger to allow us to step through each line of code and examine the value of the workspace variables as we go. By checking whether each line does what we expect it is usually possible to track down the problem line (or lines).

A debugger can be a great help when working on a large file. We can set a "break-point" on a particular line which means the program will run as normal until it reaches the break point. We can then step through line by line until we have found the problem. Using a break point is a much better idea than having to step through every single line, as often we know the problem occurs after a lot of other code (which we don't want to spend time stepping through).

### **Debugging and functions**

When stepping through each line of a piece of code in a file you have a choice of what to do when you come to a line which calls a function. You can step "over" the function, in which case the debugger just goes on to the next line of code in your current file. You can also step "into" the function, in which case the debugger goes to the first line of code inside your function (which will open up a different file).

Once inside a function you can choose to step through every single line or at some point you may like to "step out", back to your original file you were debugging.

## Chapter 3 Summary Program

---

We can now define functions and write programs that use those functions

For example, the following function uses numerical differentiation to calculate the derivatives for a set of points that are passed in as inputs.

```
function [dfdx] = NumericalDerivative(x,fx)
% NumericalDerivative uses numerical differentiation to find
% the derivative at each point for a set of discrete points
% passed to it that represent a mathematical function.
% Inputs: x      A 1D array of x values
%          fx    A 1D array of values corresponding to a
%               mathematical function f applied to each of
%               the x values, ie f(x)
% Outputs: dfdx  A 1D array of derivative values, 1 for each
%               of the x values
% Author: Peter Bier

% find the number of elements in the array
n = length(x);

% determine the step size;
h = x(2) - x(1);

% use a forward difference to calculate the derivative of the
% first element. We cannot use a central difference or
% backward difference since there is no point to the left of
% the first element
dfdx(1) = (fx(2) - fx(1))/h;

% use a central difference to calculate the derivatives of the
% middle elements since we have points to the left and right
% and a central difference is more accurate

% Note we could do the second element by itself as follows:
%   dfdx(2) = (fx(3) - fx(1))/(2*h)
% and the second to last element could be done as follows:
%   dfdx(n-1) = (fx(n) - fx(n-2))/(2*h)

% Using array slicing we can be a little cunning and do all the
% middle elements at once by subtracting an array of
% all bar the last two elements from an array of
% all bar the first two elements and then dividing by 2h.
dfdx(2:n-1) = (fx(3:n) - fx(1:n-2)) / (2*h);

% use a backward difference to calculate the derivative of the
% last element. We cannot use a central difference or
% forward difference since there is no point to the right of
% the last element
dfdx(n) = (fx(n) - fx(n-1))/h;
```

```
return
```

Below is a script file that uses this function to create a plot of the derivative of  $y=\sin(x)$ .

```
% plot the derivative of the sin function  
  
x = linspace(0, 2*pi, 50);  
y = sin(x);  
dydx = NumericalDerivative(x,y);  
plot(x,dydx);  
title('Plot of numerical derivative of sin function');  
xlabel('x');  
ylabel('derivative');
```

The result of running this script is a graph of the cos function.

# **Chapter 4: Logical operators and conditional statements**

Part of the power of computer programming comes from the ability to test if certain conditions are met and then perform different actions depending on the result. Relational and logical operators allow us to test relationships between variables and determine if expressions are true or false. Conditional statements then allow us to control what code gets executed, based on whether a condition is true or not.

## **Learning outcomes**

---

After this chapter, you should be able to:

- Use pseudocode and flow charts to describe programs
- Understand relational and logical operators
- Understand conditional statements
- Create and use boolean variables
- Control program flow with conditional statements and boolean variables

## **Controlling your computer**

---



A computer program is a sequence of simple steps. Each step does one thing.

### **Pseudocode and Flowcharts**

Before writing a program in any computer language it is often a very good idea to write out a plan, which will outline the steps in the program. This allows us to focus on what the program will do, without worrying about exactly how to write the code. Armed with a plan it is then much quicker to write the code. There are two common methods used for describing a program: pseudocode and flowcharts.

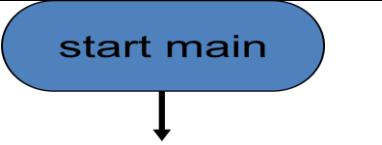
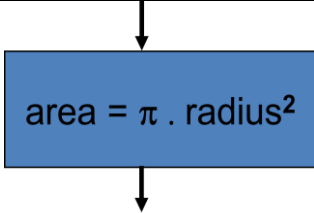
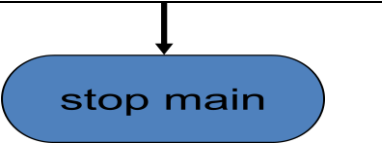
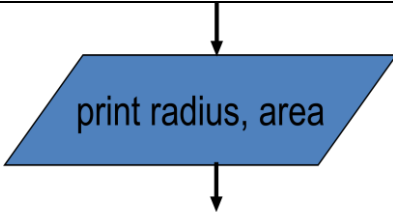
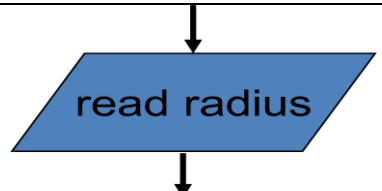
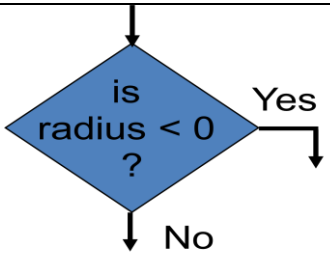
Pseudocode is a text description of program steps. It may contain fragments of code but doesn't contain the nitty-gritty details. It is similar to a recipe.

Flowcharts use geometric symbols to describe program steps. The resulting chart captures the “flow” of the program

Being able to plan programs by writing psuedocode or drawing flowcharts is a very handy skill no matter what programming language you are working with.

**Flowchart Elements**

The following table details some of the geometric elements used in flow charts. Notice how different shapes are used for different kinds of behaviour.

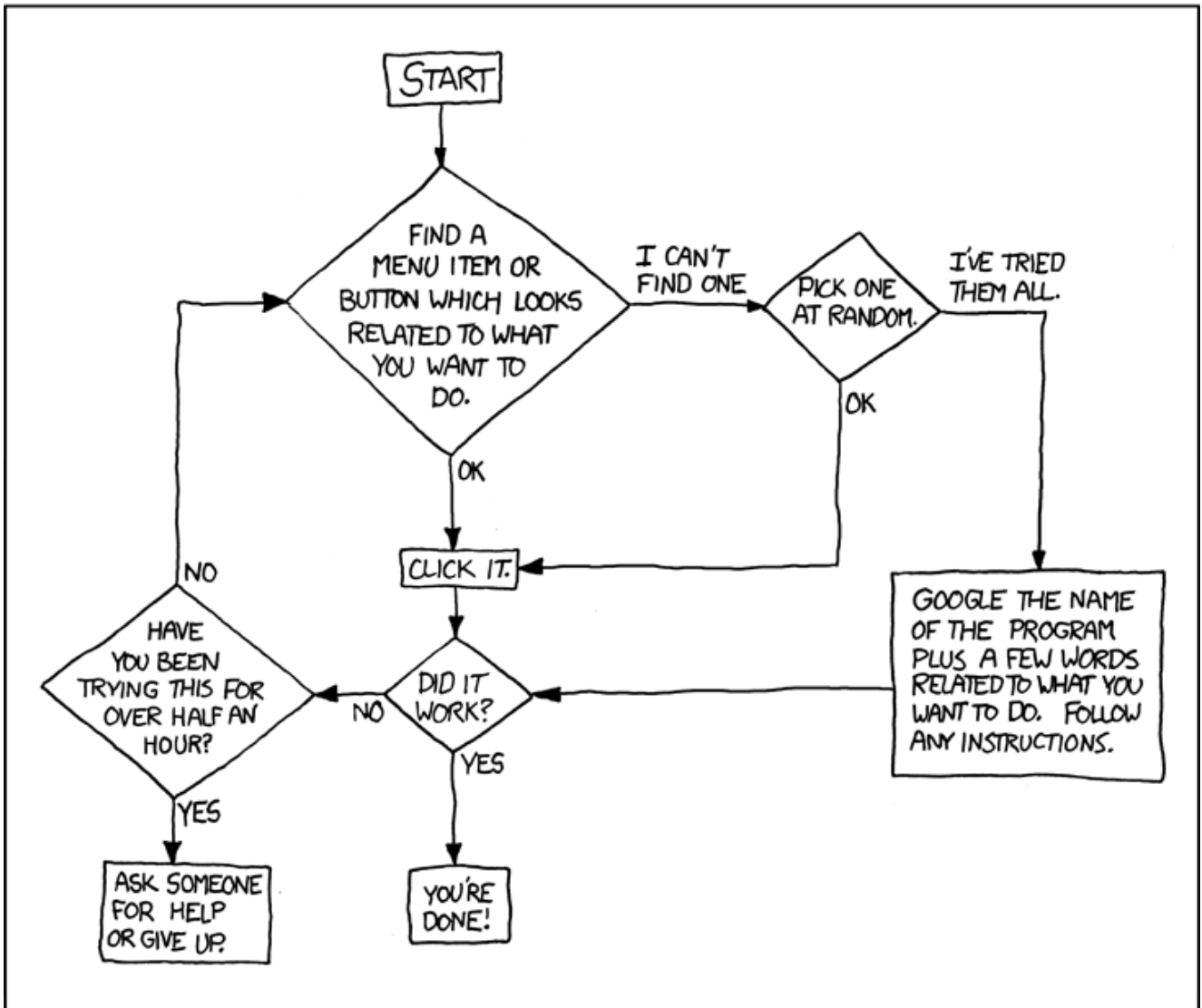
Beginning of algorithm (or function)	 <p>A blue rounded rectangle containing the text "start main". An arrow points downwards from the bottom center of the rectangle.</p>	Computation	 <p>A blue rectangle containing the text "area = π . radius<sup>2</sup>". An arrow points downwards from the top center of the rectangle.</p>
End of algorithm (or function)	 <p>A blue rounded rectangle containing the text "stop main". An arrow points downwards from the top center of the rectangle.</p>	Output	 <p>A blue parallelogram containing the text "print radius, area". An arrow points downwards from the top center of the parallelogram.</p>
Input	 <p>A blue parallelogram containing the text "read radius". An arrow points downwards from the top center of the parallelogram.</p>	Comparison	 <p>A blue diamond containing the text "is radius &lt; 0 ?". An arrow points downwards from the top center of the diamond. Two arrows exit from the right side of the diamond: one labeled "Yes" pointing to the right, and one labeled "No" pointing downwards.</p>

Here is an example of a flowchart from a popular web comic called xkcd.

Note that the comic artist wouldn't get full marks in an exam for their flowchart, as they forgot to use the correct shape for their start and stop points! For more xkcd comics see [xkcd.com](http://xkcd.com)

DEAR VARIOUS PARENTS, GRANDPARENTS, CO-WORKERS,  
AND OTHER "NOT COMPUTER PEOPLE."

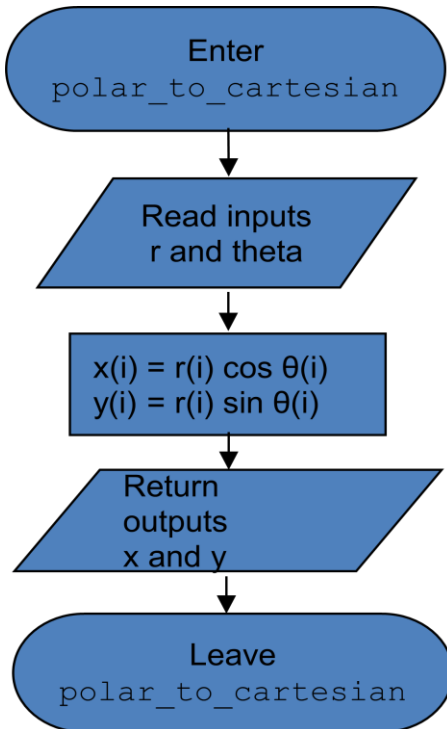
WE DON'T MAGICALLY KNOW HOW TO DO EVERYTHING IN EVERY  
PROGRAM. WHEN WE HELP YOU, WE'RE USUALLY JUST DOING THIS:



PLEASE PRINT THIS FLOWCHART OUT AND TAPE IT NEAR YOUR SCREEN.  
CONGRATULATIONS; YOU'RE NOW THE LOCAL COMPUTER EXPERT!

## Flow charting functions

Below is the flow chart for the polar\_to\_cartesian function:



Note that we use a parallelogram to show the inputs and outputs for the function.

# Programming Example

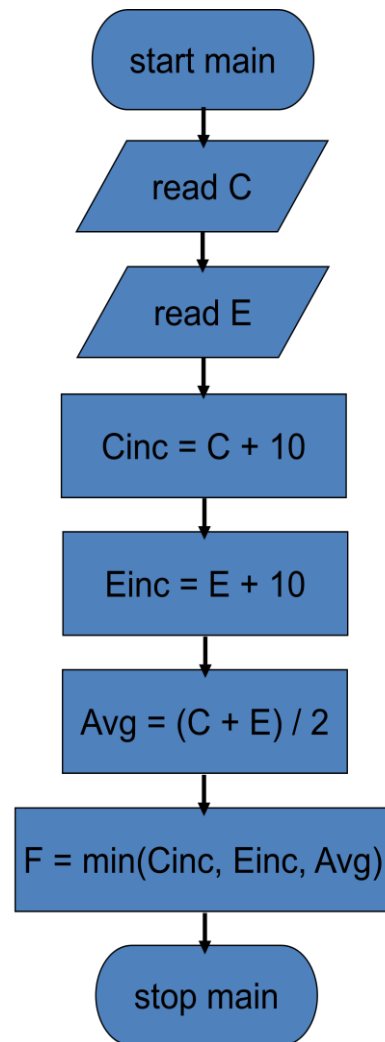
---

We can plan out a program that will calculate your final percentage given your coursework and exam percentages.

## Pseudocode

1. Get coursework percentage C
2. Get exam percentage E
3. Calculate  $C + 10$
4. Calculate  $E + 10$
5. Calculate  $(C + E) / 2$
6. Set final percentage F to be minimum of  $C + 10$ ,  $E + 10$ ,  $(C + E) / 2$

## Flowchart



Once we have our plan we can then choose a language to write our program in.



## MATLAB program

```
% This script file calculates your 131 final percentage
% from your coursework percentage and your exam percentage
% Inputs: C = coursework percentage
%          E = exam percentage
% Output: F = final percentage
clear;

% Get coursework percentage C
C = input('Enter coursework percentage > ');

% Get exam percentage E
E = input('Enter exam percentage > ');

% Calculate C + 10
Cinc = C + 10;

% Calculate E + 10
Einc = E + 10;

% Calculate (C + E) / 2
Avg = (C + E) / 2;

% Set final percentage F to be minimum of C + 10, E + 10, (C + E) / 2
F = min([Cinc, Einc, Avg])
```

## C program

```
#include <stdio.h>

#define min(X,Y) ((X) < (Y) ? (X) : (Y)) // Define the min function

int main(int argc, char* argv[]) {
/** This script file calculates your 131 final percentage
 * from your coursework percentage and your exam percentage
 */
    double C, E, Cinc, Einc, Avg, F;

    // Get coursework percentage C
    printf("Enter coursework percentage > ");
    scanf("%lf", &C);

    // Get exam percentage E
    printf("Enter exam percentage > ");
    scanf("%lf", &E);

    // Calculate C + 10
    Cinc = C + 10;

    // Calculate E + 10
    Einc = E + 10;

    // Calculate (C + E) / 2
    Avg = (C + E) / 2;

    // Set final percentage F to be minimum of C + 10, E + 10, (C + E) / 2
    F = min(Cinc, min(Einc, Avg));
    printf("Final percentage = %g\n", F);

    return 0;
}
```

# Implementing Min

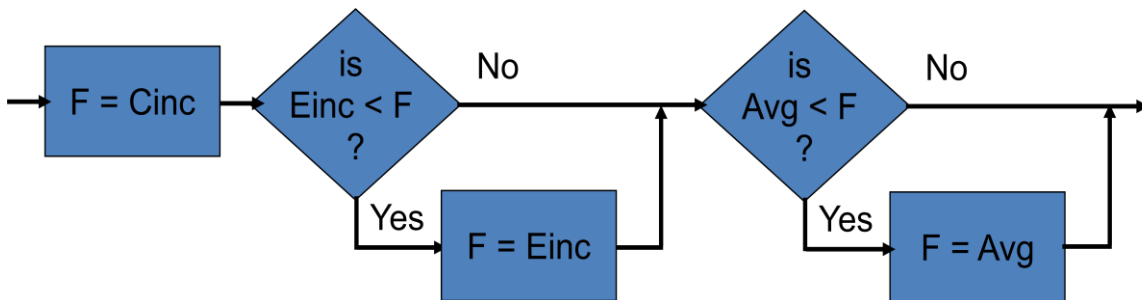
---

What if min was not an in-built function? We could write our own to perform the same task, which would need both comparisons and logic

## Pseudocode

- 1.F = Cinc
- 2.If Einc < F, set F = Einc
- 3.If Avg < F, set F = Avg

## Flowchart



Before we can write our own min function we will need a way of comparing relationships between values and a way of determining which commands to run based on that comparison.

## Relational operators

---

Relational operators test relationships between variables.

- A == B tests whether A equals B
- A ~= B tests whether A does not equal B
- A < B tests whether A is less than B
- A > B tests whether A is greater than B
- A <= B tests whether A is less than or equal to B
- A >= B tests whether A is greater than or equal to B

## Using relational operators

<pre>&gt;&gt; a=3; &gt;&gt; b=4; &gt;&gt; a==b ans =     0 &gt;&gt; a~=b ans =     1 &gt;&gt; a&lt;b ans =     1</pre>	<pre>&gt;&gt; a&gt;b ans =     0 &gt;&gt; a&lt;=b ans =     1 &gt;&gt; a&gt;=b ans =     0</pre>
--	--

# Logical Operators

---

Logical operators test conditions, usually expressed as relationships between variables or expressions. A value is false if it is 0 and true otherwise. The value of a logical operator is either true or false, represented as 1 or 0 respectively.

Common logical operators are:

~p            true if p is not true  
p & q        true if both p and q are true  
p | q        true if either p or q are true

## Using Logical Operators

<pre>&gt;&gt; a=3; &gt;&gt; b=4; &gt;&gt; ~(a==b) ans =      1 &gt;&gt; a   b ans =      1</pre>	<pre>&gt;&gt; c=5; &gt;&gt; (a&lt;b) &amp; (b&lt;c) ans =      1 &gt;&gt; (a&gt;b)   (a&gt;c) ans =      0 &gt;&gt; (a&gt;b)   (b&lt;c) ans =      1</pre>
--	--

## Conditional statements

---

A conditional statement has two parts to it, a condition and a dependent:

“If it is **sunny outside** then *I will cycle to university.*”  
          **condition**                            *dependent*

•“If I get **more than 90% in the final exam** then *I will buy myself an iPhone .*”  
          **condition**                            *dependent*

## Using MATLAB

if **examMark > 90** then *disp('Time to buy an iPhone')* end  
          **condition**                            *dependent*

# if...end

---

## Syntax

```
if condition
    some commands
end
```

*dependent*

The word end lets MATLAB know when the conditional statement is finished. It is usual to use the indentation shown above. This makes it simple to spot where the end of the if occurs.

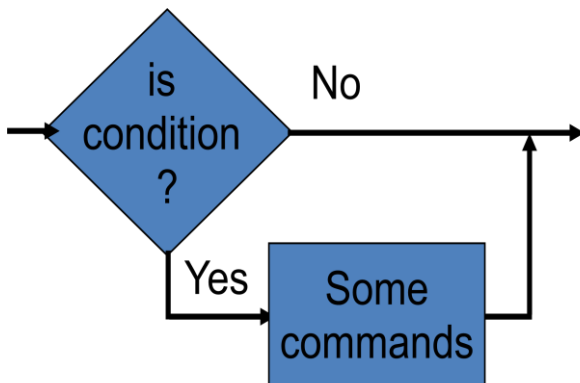
## Pseudocode

1.If *condition*, *some commands*

## Alternative Pseudocode

1. If *condition*  
a) *Some commands*

## Flowchart



## Example

```
File: myIf.m
a=2;
b=5;
if a<b
    disp(a)
end
```

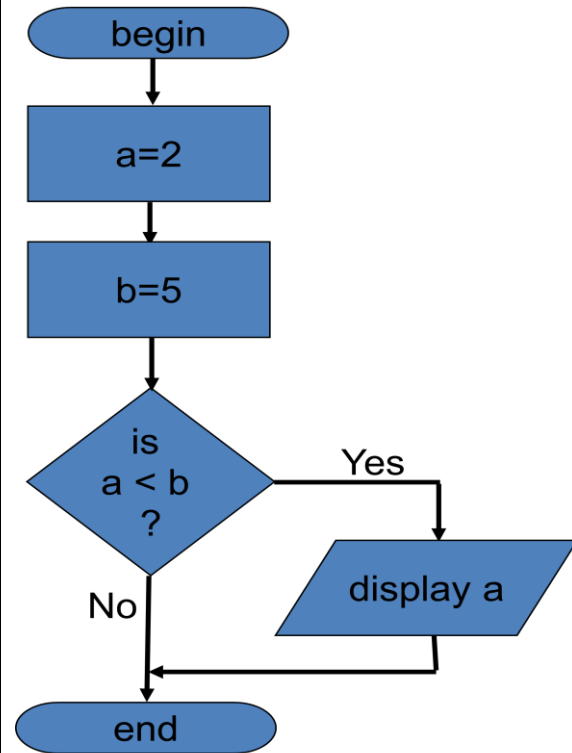
```
MATLAB command prompt
>> myIf
    2
>>
```

## Describing myIf.m

### Pseudocode

1. Set  $a = 2$
2. Set  $b = 5$
3. If  $a < b$ 
  - a) Display  $a$

### Flowchart



# if...else...end

---

## Syntax

```
if condition  
    some commands
```

```
else  
    some other commands
```

```
end
```

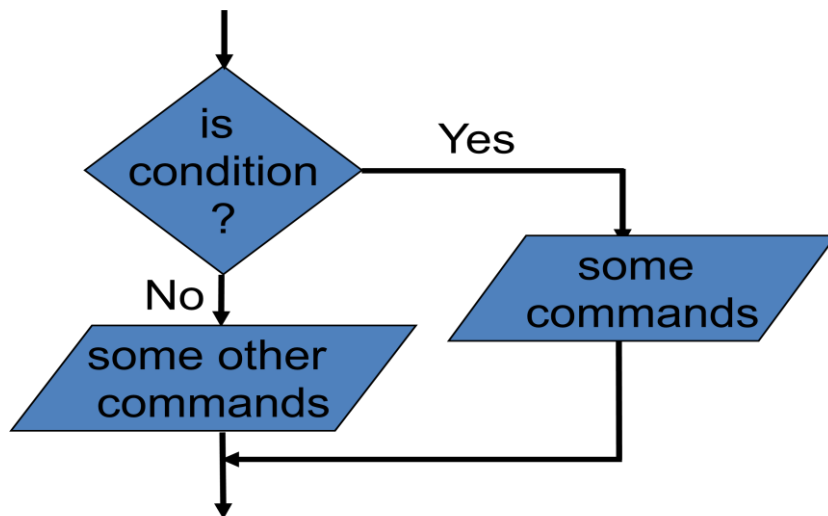
This section is  
OPTIONAL

Remember that the word end lets MATLAB know when the conditional statement is finished. It is important not to forget it!

## Pseudocode

1. If *condition*
  - a) *Some commands*
2. Else
  - a) *Some other commands*

## Flowchart



## Example

```
File: myIfElse.m  
a=5;  
b=4;  
if a<b  
    disp(a)  
else  
    disp(b)  
end
```

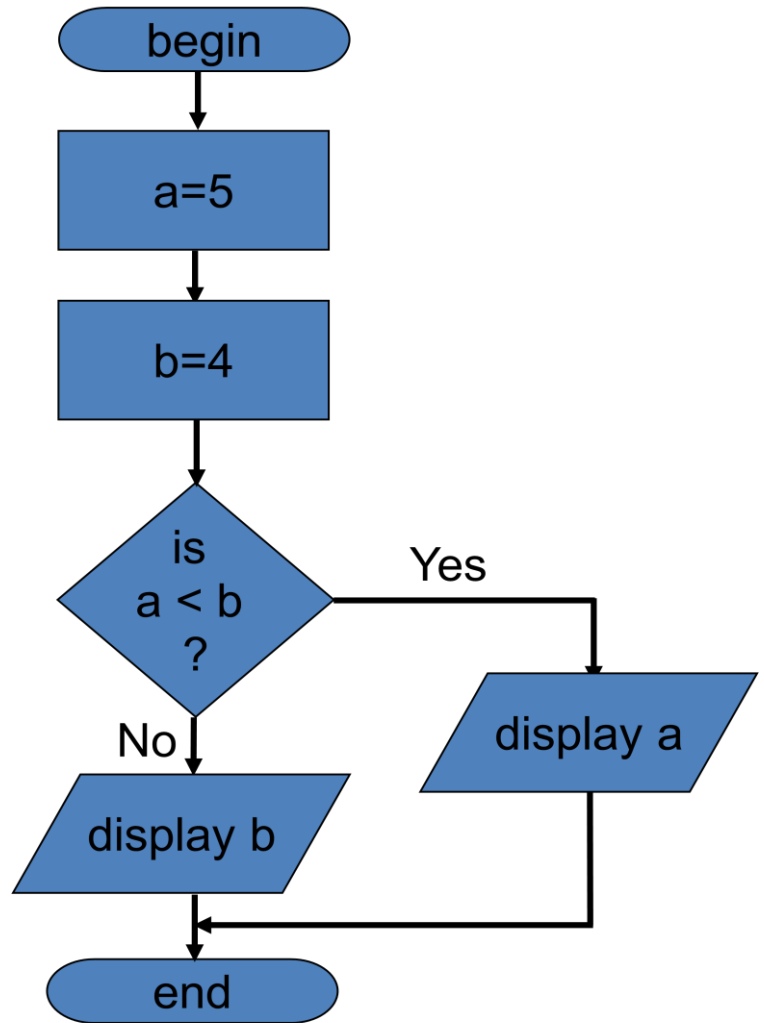
```
MATLAB command prompt  
>> myIfElse  
    4  
>>
```

**Describing myIfElse.m**

**Pseudocode**

- 1.Set a = 5
- 2.Set b = 4
- 3.If a < b
  - a)Display a
- 4.Else
  - a)Display b

**Flowchart**





# if...elseif...else...end

---

## Syntax

<pre>if <i>condition</i>     <i>some commands</i> elseif <i>another condition</i>     <i>some different commands</i> else     <i>some other commands</i> end</pre>	<p>This section is OPTIONAL</p>
--	-------------------------------------

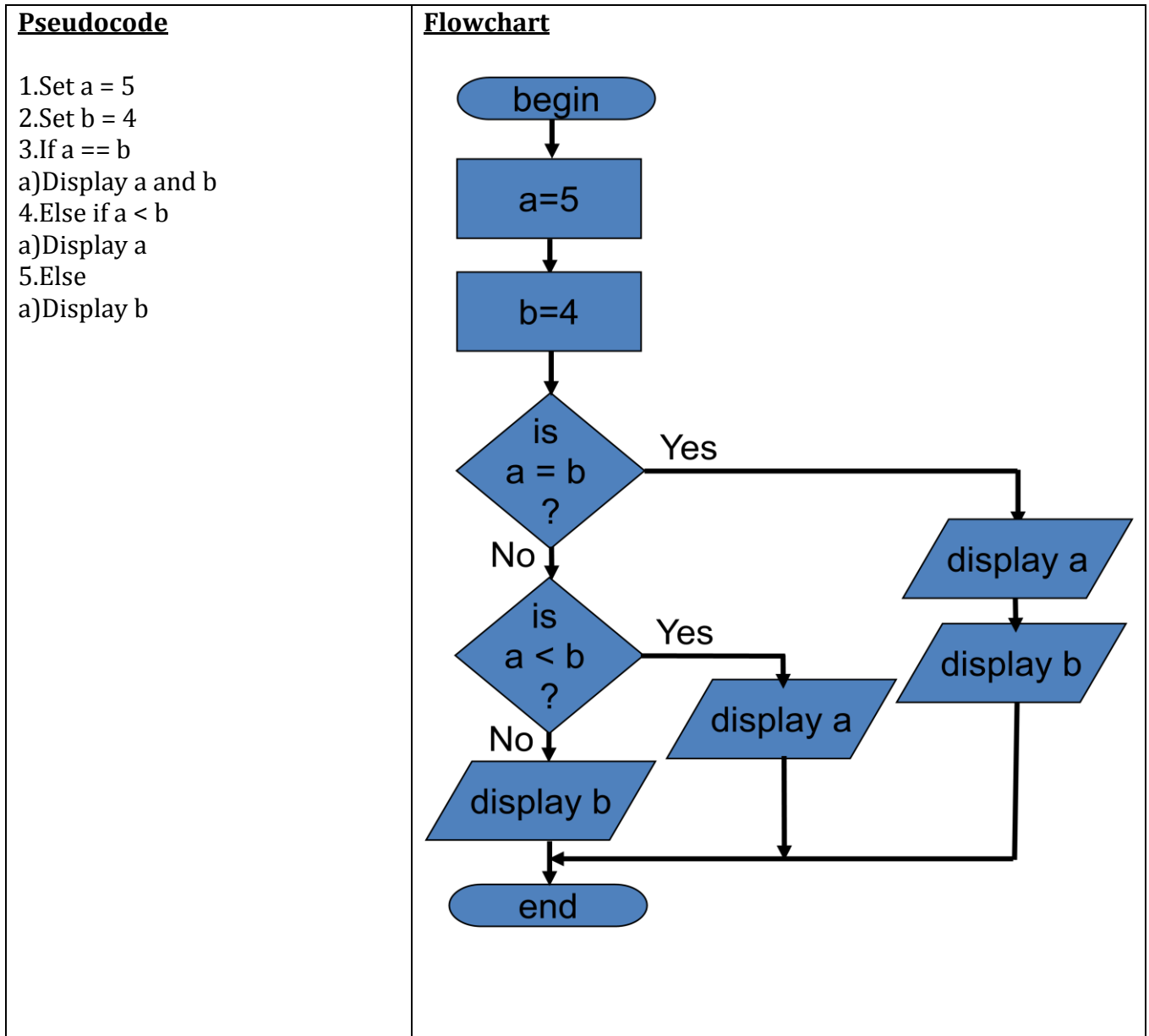
## Pseudocode

1. If *condition*
  - a) *Some commands*
2. Else If *another condition*
  - a) *Some different commands*
3. Else
  - a) *some other commands*

## Example

<pre>File: myIfElseIfElse.m a=5; b=4; if a==b,     disp(a)     disp(b) elseif a&lt;b,     disp(a) else     disp(b) end</pre>	<p>MATLAB command prompt</p> <pre>&gt;&gt; myIfElseIfElse       4 &gt;&gt;</pre>
--	--

## Describing myIfElseIfElse.m



## Testing multiple conditions

Suppose we want to check if  $a < b < c$  and then do something depending on the result.

You might be tempted to try

```
a = 2
b = 1
c = 3
if (a < b < c)
    disp('b is between a and c')
end
```

This is NOT the correct way to test if b is between a and c.

MATLAB's precedence means that it will first evaluate  $a < b$ , which is false and so is given the value 0. It will then check to see if  $0 < c$ , which is true, so the entire expression evaluates to true. Hence running the above code will display the message 'b is between a and c' even though that is clearly not the case.

To test if b is between a and c we need to test two relationships:

$a < b$  **and**  $b < c$

We should test whether **both** are true:

```
if (a<b) & (b<c)
    disp('b is between a and c')
end
```

Note the use of “&” to test whether both conditions are true.

### More on precedence

Care must be taken when checking multiple conditions, as it is easy to write code which does not do what you expect. This is because certain operations take precedence over others.

Suppose we want to check if a is less than b and c. This statement is ambiguous and can be interpreted in a few different ways.

For example it could be interpreted to mean:

test that the value of a is less than b AND that the value of a is less than c	$(a < b) \& (a < c)$
find the value of b & c and test to see if a is less than this	$a < (b \& c)$
find the value of $a < b$ and see if both this value and c are true	$(a < b) \& c$

Notice that the only difference between the MATLAB code for the second and third options is the placement of the brackets.

What happens if we leave off the brackets? Which interpretation does MATLAB use?

```
a=2
b=3
c=1
```

```
if (a < b & c)
    disp('a is less than b and c')
end
```

MATLAB's precedence is that  $<$  will be evaluated before  $\&$ . First  $2 < 3$  is evaluated which is true. Then  $1 \& 1$  is evaluated, which is also true, so the message is displayed.

Omitting the brackets had the same effect as using  $(a < b) \ \& \ c$

When testing multiple conditions it is a good idea to always put in brackets, even if you don't think you need them. Brackets make your code easier to read and understand. They also help to avoid bugs which can result from using the default evaluation order of logical and relational operators, when it checks something different from what you actually wanted.

For more detail on operator precedence see the Matlab help (Search for “operator precedence”).

### **Implementing Min Again**

We now have the tools to be able to write our own min function.

```
% myMin - finds min(Cinc, Einc, Avg)

% If Cinc < Einc and Avg, set F = Cinc
if (Cinc < Einc) & (Cinc < Avg),
    F = Cinc;
% If Einc < Cinc and Avg, set F = Einc
elseif (Einc < Cinc) & (Einc < Avg),
    F = Einc;
% If Avg < Cinc and Einc, set F = Avg
else % Must be true by default
    F = Avg;
end;
```

## **Boolean variables**

---

Boolean variables are used to store “true” and “false” values. They are very useful when working with relational operators and conditional statements.

MATLAB uses:

- a value of 1 to represent true (actually any NONZERO value is treated as true)
- a value of 0 to represent false

MATLAB also has two special variables that are useful when dealing with booleans

- **true** which has the value 1
- **false** which has the value 0

You can create boolean variables just like other variables:

```
% set isSuccessfull to true and finished to false
isSuccessful = 1;
finished = 0;
```

Or equivalently:

```
isSuccessful = true;
finished = false;
```

It is common to use boolean variables to store an answer to some “question” that controls a conditional statement or *while loop*.

```
if (isSuccessful)
    disp('Time to celebrate!');
end
```

### **Naming boolean variables**

It is good programming practice to choose a variable name that indicates the type of the variable. One common naming convention for boolean variables is to start every name with the word "is". This makes it clear how to interpret a value of true or false. For example the name `isSuccessful` is much more meaningful than a variable called `status`.

Try to write statements that read well. Consider which of the following reads better:

<pre>if isSuccessful     do something end</pre>	<pre>if ~isFailure     do something end</pre>
---	---

If you do not want to use names which start with "is" then it is a good idea to use a yes/no or true/false question as the name. This helps you write readable code. eg:

```
if( atUniversity & stillAStudent )
    needMoreMoney = 1;
end
```

## Chapter 4 Summary Program

---

```
% This program requests information on a passenger's luggage
% and determines if their luggage is acceptable for air travel
% Maximum Dimensions of Carry-on Luggage: 115 linear cm
% (length + width + height)
% Maximum weight of carry-on luggage is 7 kg
% Maximum weight of checked luggage is 20kg

% get passenger luggage information
carryonLength = input('Enter carry on length:');
carryonWidth = input('Enter carry on width:');
carryonHeight = input('Enter carry on height:');
carryonWeight = input('Enter carry on weight:');
checkedWeight = input('Enter checked bag weight:');

linearDimensions = carryonLength + carryonWidth +
carryonHeight;

luggageIsAcceptable = 1;

% determine if carry on is acceptable
if( linearDimensions <= 115 & carryonWeight <= 7)
    disp('Carry on bag acceptable')
else
    % carry on not acceptable
    luggageIsAcceptable = 0;

    % display reason(s) why carry on not accepted
    if (linearDimensions > 115)
        disp('Carry on too big');
    end

    if( carryonWeight > 7)
        disp('Carry on too heavy');
    end
end

% determine if checked bag is acceptable
if (checkedWeight <= 20)
    disp('Checked bag acceptable')
else
    disp('Checked bag too heavy');
    luggageIsAcceptable = 0;
end

if( luggageIsAcceptable )
    disp('You may now board your flight');
end
```

Some examples of this program running are given below (user input is in bold):

```
Enter carry on length: 30  
Enter carry on width: 40  
Enter carry on height: 50  
Enter carry on weight: 4  
Enter checked bag weight: 21
```

```
Carry on too big  
Checked bag too heavy
```

```
Enter carry on length: 30  
Enter carry on width: 40  
Enter carry on height: 20  
Enter carry on weight: 8  
Enter checked bag weight: 19
```

```
Carry on too heavy  
Checked bag acceptable
```

```
Enter carry on length: 30  
Enter carry on width: 40  
Enter carry on height: 20  
Enter carry on weight: 5  
Enter checked bag weight: 19
```

```
Carry on acceptable  
Checked bag acceptable  
You may now board your flight
```

# Chapter 5: Loops

Often when writing computer programs we will want to run the same (or very similar) piece of code several times. MATLAB provides commands that allow us to “loop” over a piece of code and run it multiple times.

We may know how many times you want to loop over a section of code, in which case we use a for loop.

We may need to keep looping until some condition is met, in which case we use a while loop.

Remember, if you find yourself writing the same lines of code more than a couple of times in a row, chances are you should be using a loop. Using loops will save you some typing and also make your program easier to read and maintain.

## Learning outcomes

After working through this chapter, you should be able to:

- Explain the concept of a while loop
- Use while loops in a program
- Explain the concept of a for loop
- Use for loops in programs
- Manipulate 1D arrays using a for loop
- Describe loops using flowcharts and pseudocode

## While loops

While loops are used when you need to keep looping while some condition remains true. They are very similar to if statements. An if statement checks the truth of a condition and then executes a piece of code if the condition is true. A while statement checks the truth of a condition and **while** the condition remains true repeatedly loops over a piece of code, executing it again and again.

Suppose you wanted to write out the square of an integer, only if the value squared was less than 50.

It would be easy to come up with an if statement to do this:

```
i = input('Enter an integer');
if i^2 <= 50
    disp(i^2)
end
```



Suppose that we now wanted to write out the squares of all integers less than 50.

One way to do this would be as follows:

File: squares.m	MATLAB command prompt
<pre>disp(1^2); disp(2^2); disp(3^2); disp(4^2); disp(5^2); disp(6^2); disp(7^2);</pre>	<pre>&gt;&gt; squares     1     4     9    16    25    36    49  &gt;&gt;</pre>

It is pretty inefficient to type out seven nearly identical lines. The situation would be even worse if we had wanted to write out all the squares less than 1,000,000.

Notice how each line is almost identical, with only one number changing each time. In fact we can even make each display command call the same, by using a variable to hold the number we are squaring:

File: squares2.m	MATLAB command prompt
<pre>i= 1 disp(i^2) i= 2 disp(i^2) i =3 disp(i^2) i =4 disp(i^2) i = 5 disp(i^2) i = 6 disp(i^2) i = 7; disp(i^2)</pre>	<pre>&gt;&gt; squares2     1     4     9    16    25    36    49  &gt;&gt;</pre>

Unfortunately this has made our code even longer!

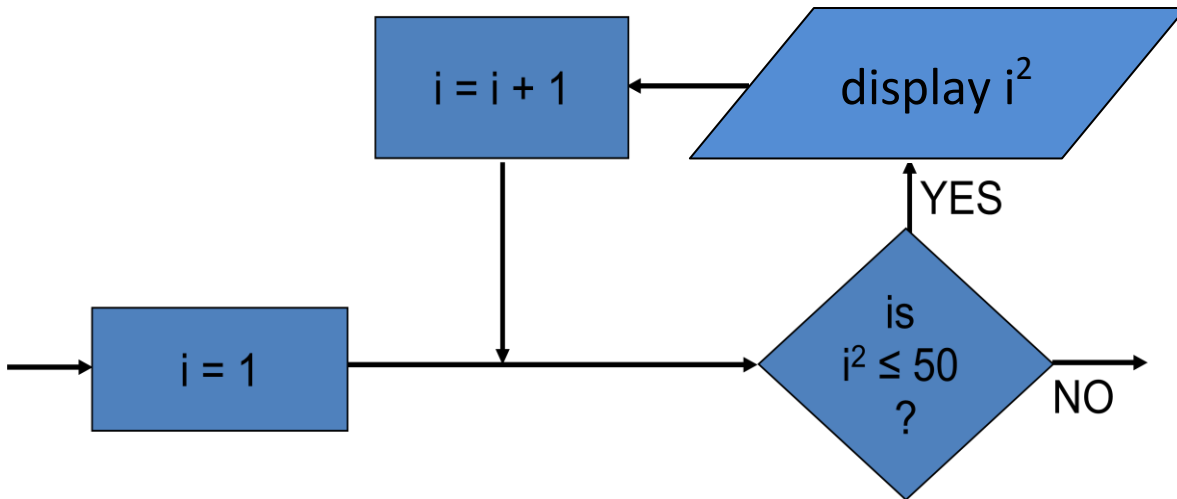
Fortunately MATLAB provides us a way of "looping" over the command `disp(i^2)` and executing it while `i^2 < 50`

We can represent what we want to do using pseudocode or a flowchart

## Pseudocode

```
i = 1
while i2 <= 50
    display i2
    i = i + 1
end
```

## Flowchart



To achieve this sort of behaviour we use a *while loop*.

## MATLAB while loop example

<pre>File: squares3.m i = 1; while i^2 &lt;= 50     disp(i^2)     i = i + 1; end</pre>	<pre>MATLAB command prompt &gt;&gt; squares3       1       4       9      16      25      36      49</pre>
--	--

**IMPORTANT:** Note that for the while loop to work, we need to make sure the variable `i` increases by one each time, as otherwise the value of `i` will stay the same and consequently the value of  $i^2$  will **never change**.

To make sure that the value of `i` changes, the following line is executed each time we go through the loop

```
i = i + 1
```

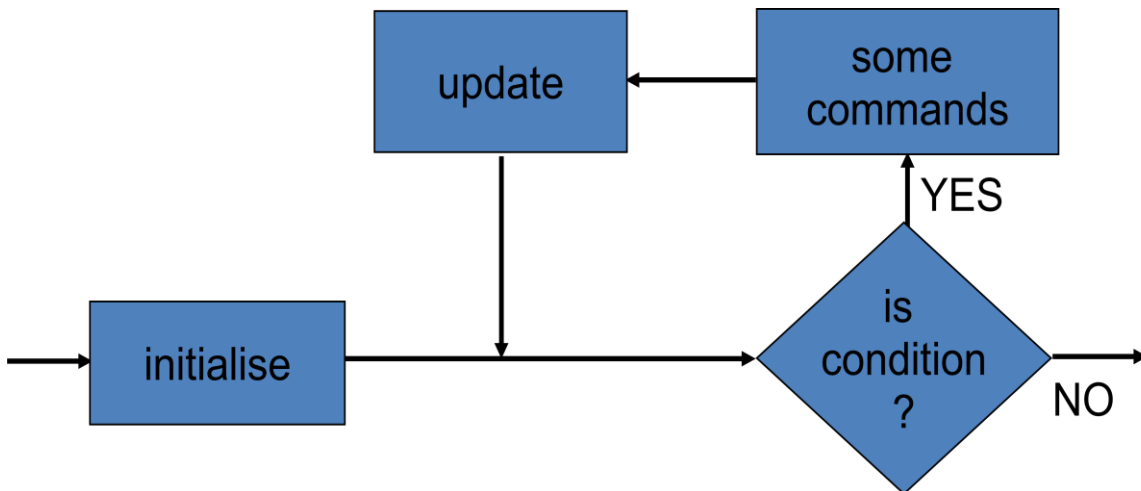
Note how this has the effect of increasing the value of `i` by one. Think about what would happen if we left out this line.

## General form of a while loop

### Pseudocode

```
initialise
while condition
    some commands
    update
end
```

### Flowchart



The update step is very important. If it is omitted or not done correctly you may get stuck in an infinite loop.

## Infinite loops

---

An “Infinite loop” is a piece of code that will execute again and again and again ... without ever ending.

Possible reasons for infinite loops:

- getting the conditional statement wrong
- forgetting the update step

If you are in an infinite loop then *ctrl-c* stops MATLAB executing your program

<pre>File: infinite_loop.m i = 1; while i &gt;= 0     disp(i)     i = i + 1; end</pre>	<pre>MATLAB command prompt &gt;&gt; infinite_loop      1      2      3      4 LOTS MORE NUMBERS     6824     6825 AND SO ON, until ctrl-C</pre>
--	---

REMEMBER: use CTRL-C to break out of an infinite loop.

## **Booleans and while loops**

---

Using a boolean variable to control a while loop allows us to write more readable code.

### **Syntax**

```
stillLooping = true;
while stillLooping
    some commands
    if some conditions
        stillLooping = false;
    end
end
```

### **Boolean while Example**

```
iPhoneCost = 979;
needMoreMoney = 1;

while(needMoreMoney)
    moneySaved = input('How much money have you saved?');
    if( moneySaved >= iPhoneCost )
        disp('You do not need to save any more money');
        needMoreMoney = 0;
    else
        disp('Keep saving');
    end
end
```

## For loops

---

Recall that when we wanted to write out the squares of all integers less than 50, we used a while loop

```
i = 1;
while i^2 <= 50
    disp(i^2)
    i = i + 1;
end
```

We could just as easily use the following while loop to achieve the same task

```
i = 1;
while i <= 7
    disp(i^2)
    i = i + 1;
end
```

Here the condition has been changed, so that we are checking if the variable *i* is less than 7. This loop will be run exactly seven times, with the body of the loop being run **for each** of the numbers 1, 2, 3, 4, 5, 6 and 7.

MATLAB provides another way of "looping" over the commands called a **for** loop. For loops can be more convenient to use than while loops in many cases (although you can always achieve the same thing with an appropriate while loop). The convenience of a for loop lies in the fact that they take care of the update step for us.

For loops are particularly useful when we know exactly how many times we want to loop through a piece of code or we wish to execute a loop a number of times **for** a given list of values. In the example above we wish to display the square of *i* for each of the values from 1 to 7.

The syntax of the appropriate for loop looks like this:

File: squares4.m for i=1:7 disp(i^2) end	MATLAB command prompt >> squares4 1 4 9 16 25 36 49 >>
---	---

The for loop allows us to execute the disp command for each of the values from 1 to 7.

The first time through the loop, *i* has the value 1. The second time *i* has the value 2 and so on, until the very last time when *i* has the value 7. Recall that 1:7 is another way of writing the array [1,2,3,4,5,6,7]. We can think of a for loop as a loop that executes a given piece of code **for each** element in an array.

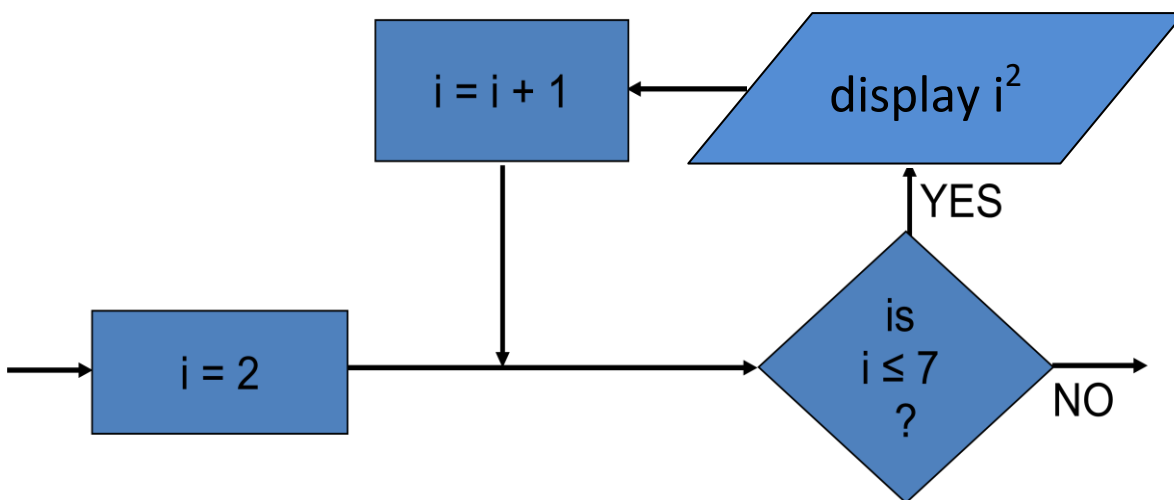
### Pseudocode for our squares example

```
for i = 1 to 7 by 1
    display i2
end
```

If we wanted to print out all the squares from 1 to 100, instead of 1 to 7, just a small change is necessary:

```
for i = 1:100
    disp(i^2)
end
```

### Flowchart for our squared example



Notice this flowchart is very similar to the one for the while loop that did the equivalent task. This shouldn't be a surprise as the task can be accomplished using either kind of loop.

### More on for loops

At the heart of a for loop is a loop variable, often given the name *i*.

The first time through the loop the variable *i* has a start value.

Each subsequent time the value of *i* is increased by a step value (usually 1).

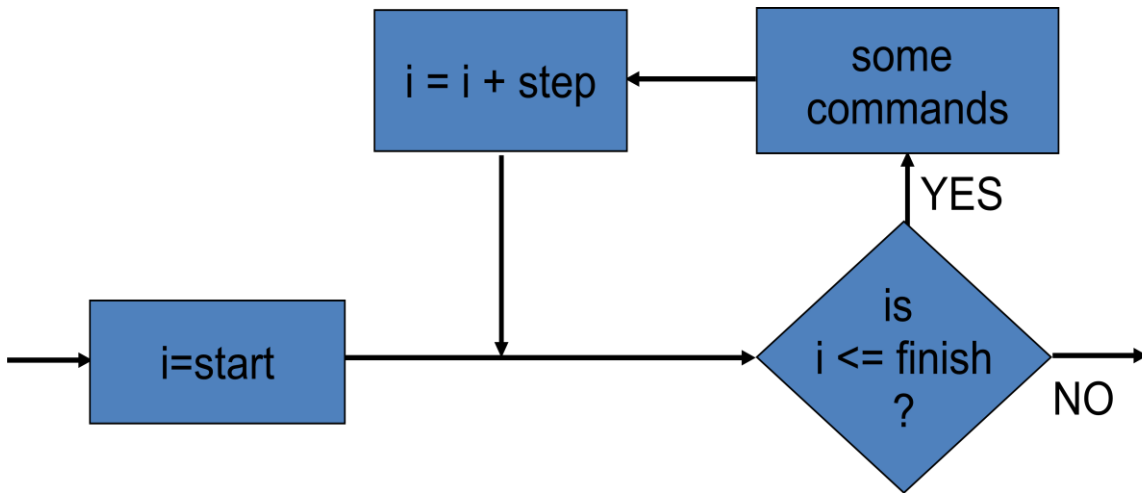
We continue looping until we reach the finish value.

The commands inside the for loop will often use the loop variable (but they don't have to).

### Pseudocode

```
for i = start to finish by step
    some commands
end
```

## Flowchart



## Syntax

```
for variable = start:step:finish  
    some commands  
end
```

If no step is specified it is assumed to be 1.

## Some examples

File: count_to_five.m for i=1:5 disp(i) end	MATLAB command prompt >> count_to_five 1 2 3 4 5 >>
--	--

File: triple_greeting.m for i=1:3 disp('Hello') end	MATLAB command prompt >> triple_greeting Hello Hello Hello >>
--	--

## Different step values

<pre>File: count_time.m for time=0:0.1:0.5     disp(time) end</pre>	<pre>MATLAB command prompt &gt;&gt; count_time     0     0.1000     0.2000     0.3000     0.4000     0.5000</pre>
---	---

<pre>File: countdown.m for i=5:-1:1     disp(i) end disp('blastoff!')</pre>	<pre>MATLAB command prompt &gt;&gt; countdown     5     4     3     2     1     blastoff! &gt;&gt;</pre>
---	--

Note that the for loop stops as soon as the finish value is exceeded.

<pre>File: count_odd.m for i=1:2:10     disp(i) end</pre>	<pre>MATLAB command prompt &gt;&gt; count_odd     1     3     5     7     9 &gt;&gt;</pre>
---	--



## For Loops and arrays

For loops are ideal for processing arrays. The loop variable can be used as an array index. This allows us to use the same piece of code but run it on each element in an array in turn.

File: check\_weights.m

```
bagWeights = [4.5, 3.4, 5.0, 7.2, 10.0, 4.9, 8.6]
for i=1:length(bagWeights)
    if (bagWeights(i) > 7)
        disp(['Bag ', num2str(i), ' too heavy!']);
        bagIsTooHeavy(i) = true;
    else
        bagIsTooHeavy(i) = false;
    end
end
end
```

MATLAB command prompt

```
>> check_weights
    Bag 4 too heavy!
    Bag 5 too heavy!
    Bag 7 too heavy!

>>
```

The array bagIsTooHeavy will also have been populated

```
>> bagIsTooHeavy

bagIsTooHeavy =

    0    0    0    1    1    0    1
```

Notice how the loop variable has been used as an index into our bagWeights and bagIsTooHeavy arrays. A common programming error is to create a for loop attempts to access elements of an array that do not exist. This will occur if i happens to be zero, have a negative value or have a fractional value. Array indices in Matlab can only be positive integers.

## Chapter 5 Summary Program

---

We can now write programs that use loops.

```
% this script file lets a user enter internal and examination
% marks for the students in a class and then calculates their
% final marks.

% index keeps track of which student we are up to
i = 0;

% boolean variable which is true while there are more students
% to process
moreStudentsToProcess = 1;

while( moreStudentsToProcess )

% increment student number
    i = i + 1;

% ask user to enter internal and exam marks for student
    id(i) = input('Enter student id number:');
    internalMark(i) = input('Enter the internal mark:');
    examMark(i) = input('Enter exam mark:');

    disp('Are there more students to process?')
    response = input('Enter 1 for yes, 0 for no:');

% response of 0 means no more students to process
    if( response == 0)
        moreStudentsToProcess = 0;
    end
end

% calculate final exam mark for each student
for j=1:length(id)
    coursework = internalMark(j);
    exam = examMark(j);
    cmax = coursework + 10;
    emax = exam + 10;
    avg = (coursework + exam)/2;
    finalMark(j) = min([cmax,emax,avg]);
end

disp('Student ids are');
disp(id);
disp('Final marks are');
disp(finalMark);
```

An example of this program running is shown below, with user input in bold.

```
Enter student id number:123
Enter the internal mark:12
Enter exam mark:56
Are there more students to process?
Enter 1 for yes, 0 for no:1
Enter student id number:234
Enter the internal mark:56
Enter exam mark:78
Are there more students to process?
Enter 1 for yes, 0 for no:1
Enter student id number:565
Enter the internal mark:67
Enter exam mark:69
Are there more students to process?
Enter 1 for yes, 0 for no:0
Student ids are
    123    234    565

Final marks are
    22    66    68
```

# Appendix: MATLAB Command Reference

There are many MATLAB features that are not included in the lecture and lab notes. Listed below are some of the MATLAB functions and operators available, grouped by subject area. Use the on-line help facility for more detailed information on the functions.

## General

help	help facility
demo	run demonstrations
who	list variables in memory
what	list M-files on disk
size	row and column dimensions
length	vector length
clear	clear workspace
computer	type of computer
^C	local abort
exit	exit MATLAB
quit	same as exit

## Disk Files

chdir	change current directory
delete	delete file
diary	diary of the session
dir	directory of files on disk
load	load variables from file
save	save variables to file
type	list function or file
what	show M-files on disk
fprintf	write to a file

## Matrix/Array Operators

### Matrix Operators

---

+	addition
-	subtraction
*	multiplication
/	right division
\	left division
^	power
'	conjugate transpose

### Array Operators (Element wise)

---

+	addition
-	subtraction
.*	multiplication
./	right division
.\	left division
.^	power
.'	transpose

## Relational and Logical Operators

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal
&	and
	or
~	not

## Control Flow

if	conditionally execute statements
elseif	used with if
else	used with if
end	terminate if, for, while
for	repeat statements a number of times
while	do while
break	break out of for and while loops
return	return from functions
pause	pause until key pressed

## Special Values

ans	answer when expression not assigned
pi	pi
inf	infinity
NaN	Not-a-Number
clock	wall clock
date	date

## Special Characters

=	assignment statement
[	used to form vectors and matrices
]	see [
(	arithmetic expression precedence
)	see (
.	decimal point
...	continue statement to next line separate subscripts and function arguments
,	
;	end rows, suppress printing
%	comments
:	subscripting, vector generation
!	execute operating system command

## Programming and M-files

input	get numbers from keyboard
keyboard	call keyboard as M-file
error	display error message
function	define function
eval	interpret text in variables
feval	evaluate function given by string
echo	enable command echoing
exist	check if variables exist
etime	elapsed time
global	define global variables
startup	startup M-file
getenv	get environment string
menu	select item from menu

## Special Matrices

diag	diagonal
eye	identity
magic	magic square
ones	constant
rand	random elements
zeros	zero

## Trigonometric Functions

sin	sine
cos	cosine
tan	tangent
asin	arcsine
acos	arccosine
atan	arctangent
atan2	four quadrant arctangent
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangent
asinh	hyperbolic arcsine
acosh	hyperbolic arccosine
atanh	hyperbolic arctangent

## Elementary Math Functions

abs	absolute value or complex magnitude
angle	phase angle
sqrt	square root
real	real part
imag	Imaginary part
conj	complex conjugate
round	round to nearest integer
fix	round toward zero
floor	round toward -infinity
ceil	round toward infinity
sign	signum function
rem	remainder
exp	exponential base e
log	natural logarithm
log10	log base 10

## Command Window

clc	clear command screen
format	set output display format
disp	display matrix or text
fprintf	print formatted number

## Graph Paper

plot	linear X-Y plot
loglog	loglog X-Y plot
semilogx	semi-log X-Y plot
semilogy	semi-log X-Y plot
polar	polar plot
mesh	3-dimensional mesh surface
contour	contour plot
meshgrid	domain for mesh plots
bar	bar charts
stairs	stairstep graph
errorbar	add error bars

## Decompositions and Factorisations

chol	Cholesky factorization
eig	eigenvalues and eigenvectors
hess	Hessenberg form
inv	inverse
lu	factors from Gaussian elimination

## Nonlinear Equations and Optimisation

fmin	minimum of a function of one variable
fmins	minimum of a multivariable function
fsolve	solution of a system of nonlinear equations
fzero	zero of a function of one variable

## File formats (used with fscanf, fprintf etc)

%c	characters
%d	decimal numbers
%e, %f, %g	floating-point numbers
%i	signed integer
%o	signed octal integer
%s	series of non-white-space characters
%u	signed decimal integer
%x	signed hexadecimal integer

## Graph Annotation

title	plot title
xlabel	x-axis label
ylabel	y-axis label
grid	draw grid lines
hold	hold plot on screen

## Column-wise Data Analysis

max	maximum value
min	minimum value
mean	mean value
median	median value
std	standard deviation
sort	sorting
sum	sum of elements
prod	product of elements
cumsum	cumulative sum of elements
cumprod	cumulative product of elements
hist	histograms

## Elementary Matrix Functions

expm	matrix exponential
logm	matrix logarithm
sqrtn	matrix square root
poly	characteristic polynomial
det	determinant

## Differential Equation Solution

ode23	2nd/3rd order Runge-Kutta method
ode45	4th/5th order Runge-Kutta-Fehlberg method