Introduction to Programming with MATLAB

Course Outline

These web pages provide a reference source to complement the CAPE lectures - "An Introduction to Numeric Computer Programming using MATLAB". Links are provided for each of the major headings and for relevant exercises. You should do the exercises before attempting the course work.

- **What is a program?**
  - Problem definition: steps for solving problem
  - Problem -> Algorithm: formal and precise specification of steps
  - Algorithm -> Program: encoding the steps into a computer language (such as MATLAB)

- **The MATLAB environment**
  - Interactive input and getting HELP
  - Writing programs using m files

Try the [first exercise](#) listed below.

- **An introduction to MATLAB programming**
  - Variables and numbers
  - Expression: Calculations with variables
  - Simple Programs: Order of commands and comments

Try the [second exercise](#) listed below.

- **Multidimensional data structures**
  - Arrays, vectors and matrices
  - Simple tables
  - Mathematical operators (revisited)
Try the exercise for arrays, as listed below.

- **User Procedures**
  - Functions
  - Local and global variables

Try the flow-sheeting exercise which requires the use of procedures.

- **Program control**
  - IF statements
  - Program Loops: FOR and WHILE statements

Try the exercise on Program loops listed below. Also give the exercise on a Taylor series approximation to the Sine function a try.

- **Input/Output**
  - Output to the screen
  - Input from the keyboard
  - Load and saving data

For your own computers

For those of you that may have a computer at home or elsewhere, there is a system, known as Octave, written by John W. Eaton (Department of Chemical Engineering, University of Wisconsin) and others, which is similar to Matlab. For the material covered in this course, Octave is a perfect replacement for Matlab. A free (i.e. as defined by the Free Software Foundation) version for Microsoft Windows is available here, provided by the Matlinks Sourceforge web site. Please note: the download is approximately 7.5MB in size.

For a quicker download, I have placed a copy here and have also written some simple installation instructions.

Exercises

The following are a set of exercises which you should attempt, in order, to gain experience with the
different aspects of Matlab. If you are able to do all these exercises, the hand-in project should pose little difficulty!

1. PC Lab Introduction
2. Estimating distance to horizon.
3. Working with Arrays
5. Program loop exercise.
6. Sine function approximation
Solving problems by computer

Computers are tools which enable us to solve a wide range of problems, both numeric and textual. The purpose of this course is to teach you the rudiments of numerical computing, starting with how to write programs and leading into specific numerical techniques for a variety of problems in Chemical Engineering.

You have already gained experience in using computers. In particular, you have learned how to write documents using a word processor such as Microsoft Word, how to do simple calculations using a spreadsheet calculator such as Microsoft Excel, and how to solve more complex problems using specific tools such as GAMS. This course will help you learn how to develop your own applications, although they will be much simpler than those you have already seen (as they involved large teams of programmers to develop over many years!).

There are three steps in writing a program:

1. Defining the problem you want to solve,
2. Writing an algorithm which describes the steps need to solve the problem, and
3. Writing the actual program to solve your problem.

Each of these steps is now discussed in more detail.

**Problem definition**

The first step, and arguably the most important, is to properly define the problem to solve. It is difficult to write a program if the problem itself has ambiguities. Computers are not particularly good at handling ambiguities: they need to have everything spelled out!

**Example problem**

You've been out all evening celebrating the fact that your first program ran successfully. You've come home and suddenly realize that you are hungry. As it's quite late, and you possibly do not have the coordination for anything more complex, you figure that beans on toast is suitably filling and appetising.

The problem is to make this dish.

What constitutes a precise problem description for this example? As this is a simple problem, we can summarise it as

Prepare a suitable amount of beans and toast, put the beans and toast together on a plate, and find a place to eat them.

As mentioned, this is a simple problem so the problem description is both short and easy to formulate. Real problems will of course be more difficult to state clearly but it is important that this be done before going to the next step.

**Writing an algorithm**

An algorithm is essentially the same as a recipe: it states the steps necessary to solve a particular problem. Algorithms
can vary greatly in the amount of detail presented and the language used to write them. The amount of detail depends on both the problem itself and the person which will be responsible in translating the algorithm to an actual program.

We will continue with our beans on toast example by writing a fairly detailed algorithm which will enable us to make our dinner should the occasion arise.

Algorithm BeansOnToast
"the first step is to locate all the necessary ingredients"
open cupboard and look for can of beans
if no beans found then
    run to corner shop and buy some
end if
open bread bin and find suitable, non-moldy, slices of bread
if no bread found or all of it moldy then
    run to corner shop again and buy some
    buy some milk for tomorrow's morning tea while there
end if
"okay, we've got the ingredients. Now prepare for the actual cooking"
find saucepan for beans
open can of beans and pour into saucepan
find a toaster
if no toaster then
    check for grill
    if no grill either then
        give up and go to sleep hungry
    end if
end if
"everything is ready. Go to it!"
turn cooker on
put beans on to heat up
put bread in toaster (or in the grill)
while beans not done and bread not done do
    if toast ready then
        get butter from fridge (hopefully there is some!)
        butter toast
    end if
    if beans ready then
        turn heat to low
    end if
end while
"everything is now ready"
turn cooker off (including grill if used)
put toast on plate
pour beans on toast
sit in front of TV to eat your beans on toast!
end Algorithm

And you thought that preparing beans on toast was easy!! Well, writing computer programs requires this level of
detail. Computers need to be told exactly what to do. The purpose of an algorithm is to write down all the steps necessary for solving a problem in enough detail to make sure everything is there but without using a computer language. The use of (what is commonly known as) **pseudo-English** should make an algorithm much easier to read than a computer program (although a well written program should not be much more difficult to read than an algorithm, especially if *comments* are used liberally and correctly -- comments will be discussed later).

There are no rules when it comes to writing an algorithm. An algorithm should be written so that you understand each and every step, the order of these steps, and how they interact. If done properly, writing a program is simply a matter of translating each step in the algorithm to one or more lines of program code.

## Writing a program

A computer program consists of a series of instructions to the computer. These instructions, which may tell the computer to add two numbers together or to write a message out to the user, are written in a *high level* language. In this course, we will be using a language known as **MATLAB** Following is an example program, one which takes two numbers from the user and prints out the result of multiplying these numbers together.

```matlab
% Simple Program
x = input('Input value of x please ');  % get the first number
y = input('Input value of y please ');  % and the second number
z = x * y;                             % compute product
z                                      % show the result to the user
```

The syntax of MATLAB will be discussed in later lectures. This example is here simply to give you an indication of what a program looks like (so you'll be able to recognise such a beast if you run into one in a dark lane...).

Click [HERE](#) for next section.
The Matlab Environment

MATLAB is an interactive program for numerical computation and data visualisation. It was originally developed in FORTRAN as a MATrix LABoratory for solving numerical linear algebra problems. The original application may seem boring (except to linear algebra enthusiasts), but MATLAB has advanced to solve nonlinear problems and provide detailed graphics. It is easy to use, yet very powerful. A few short commands can accomplish the same results that required a major programming effort only a few years ago.

THE MATLAB WORKSPACE

The prompt for a user input is shown by the double arrow:

>>

MATLAB has an extensive on-line help facility. For example:

>> help exp

EXP    Exponential.
EXP(X) is the exponential of the elements of X, e to the X.

See also LOG, LOG10, EXPM, ARITH, POW2.

By simply typing:

>> help

MATLAB will provide a list of commands that are available. If you do not know the exact command for the function you are after, another useful command is lookfor. This command works somewhat like an index. If you did not know the command for the exponential function was exp, you could type:

>> lookfor exponential

EXP    Exponential.
EXPM    Matrix exponential.

The command demo gives an overview of the available demo programs. These might be useful if you wish to know more about certain aspects of MATLAB.

Simple calculations

Simple calculations can be carried out in MATLAB. For example by typing

>> 2+2

MATLAB gives the (correct!) response:

ans =
However, most calculations are carried out using variables. This is illustrated below:

```
>> x = 2
>> z = x+x
```

The response is:

```
x =  
   2
z =  
   4
```

Actually we can suppress MATLAB from echoing to the screen by adding a semicolon (;) after each line. For example

```
>> x = 2;
>> z = x+x
```

This time the response is:

```
  z = 
   4
```

**Managing your variables**

You can view the variables currently in the workspace by typing:

```
>> who
```

```
Your variables are:
ans      x         z
```

More detail about the size of the matrices can be obtained by typing:

```
>> whos
```

```
     Name      Size      Elements   Bytes  Density   Complex
ans      1 by 1      1           8      Full      No
x        1 by 1      1           8      Full      No
z        1 by 1      1           8      Full      No

Grand total is 3 elements using 24 bytes
```

Sometimes it is desirable to clear all the variables in a workspace. This is done simply by typing:

```
>> clear
```
More frequently you may wish to clear a particular variable, such as \( x \):

\[
>> \text{clear } x
\]

**Programming in MATLAB**

Thus far we have shown the interactive features of MATLAB by entering one command at a time. Though possible it would be a very bad idea to carry out complex calculations by typing interactively at the MATLAB command prompt (What if you make a mistake? or want to do the calculation again tomorrow?). Instead MATLAB code should be written in m-files and saved for later use.

There are two ways of generating your own MATLAB code: 1) script files and 2) function routines. Function routines will be considered later in this course. We will start by initially just considering script files:

**Script files**

A script file is simply a sequence of commands that could have been entered interactively. When the sequence is long, or must be performed a number of times it is much easier to generate a script file. The following example is a very simple program used to solve an equation. If we name the file `myprog.m` then it is run by simply typing `myprog` in the MATLAB command window.

```matlab
% My first program
x = 2.0;
y = 3*sqrt(x) -7
```

**Getting started**

Get started on MATLAB by trying the exercise associated with this section.

Click [HERE](http://www.ucl.ac.uk/~ucecesf/local/e862/matlab/envir.html) for next section.
A Simple Program

To create your program, from the file menu select New: M-File. This will open a Notepad window where you can create your first MATLAB program. Type, exactly as shown, the following *program*:

```matlab
% My first program
x = 4.0;
y = 3*sqrt(x) - 7
% End of program
```

This is an example (albeit a very simple one) of a MATLAB program. You will be writing many more during the course of the rest of your degree! Save the result (which must be exactly as shown above) into a file called *myprog.m*. We will be using this program in the next section.

You should notice the use of the semicolon (;) at the end of the first line, this suppresses MATLAB printing this line to the screen. Often we only want to know the final answer from the program not the intermediate steps taken. Therefore, you will be using the semicolon (;) at the end of most lines in your programs.

Running MATLAB Programs

Running a MATLAB program is very straightforward, simply type the program name at the command prompt, in this case, type *myprog*. The commands you have listed in your file will be executed sequentially. If there are any errors in your file, MATLAB will tell you which line the error is on, and indicate what type of error has occurred.

If you get the answer $y = -1$ then you have successfully completed this first exercise.
An Introduction to MATLAB Programming

A program consists of a sequence of instructions to the computer. These instructions are used to describe the types of data to be used, the manipulations desired of this data, and how results are presented to the user of the program.

In MATLAB the commands can either either be entered directly into the workspace or they can be saved to a file - when the file is called (at the MATLAB command prompt) the instructions are executed automatically. This latter method is potentially much more powerful.

Variables

Variables are used to hold values while the program is running (similar to a memory in a calculator). Most variables can be given new values at any time. That is, the following extract of code is perfectly valid:

```matlab
... x = 1.0 y = 2*x x = 10.0 z = x/2.0 ... 
```

Code is much more readable if reasonable names are given to all variables and numeric constants. For example, the following code

```matlab
Pi = 3.1416 ... radians = degrees / (2*Pi) ... 
```

is much more readable and intuitive than

```matlab
... radians = degrees / 6.2832 ... 
```

Note that MATLAB is case sensitive, e.g. Pi is a different variable name to pi.

One major property of real numbers (otherwise known as floating point numbers) on a computer is that many can not be represented exactly. Computers use binary numbers internally so only numbers that are some exact multiple of small powers of 2 can be represented accurately. For example, 1/2 = 0.5, 1/4 = 0.25, 1/8 = 0.125, etc can be represented exactly. However, numbers such as 1/3 = 0.3333 and (maybe surprisingly to people used to working in a decimal system) 1/10 = 0.1 can not be represented exactly on a computer. For example:

```matlab
>> format long >> x = 0.1 >> y = 1 >> z = y - (x+x+x+x+x+x+x+x+x+x) 
```

The output is:
z =
1.387778780781446e-16

Though the value of $z$ is very small the answer should have been zero. This error is known as \textit{rounding error} (or round-off error) and can, in some sequences of calculations, accumulate quickly, sometimes leading to completely erroneous results.

Rounding error is more likely to appear when similar values are subtracted from each other.

**Expressions**

Calculations are performed by specifying \textit{expressions}. An expression is equivalent to a mathematical equation where the left hand side is a single variable which will contain the value of the right hand side. The equals sign (=) is an assignment (as opposed to a mathematical statement of equality) so that the expression

$$x = x + 1$$

means that the variable $x$ is given (assigned) the value that results from adding 1 to $x$. This is different from a mathematical equation which would allow us to subtract $x$ from both sides of the equation, ending up with "0 = 1" which is definitely not true!

The following standard mathematical operators are all available in MATLAB:

+  add two values together
-  subtract second value from first
*  multiply
/  divide
^  raise first value to second value

In addition many standard mathematical functions (\textit{sin}, \textit{cos}, \textit{log}, \textit{exp}, ...) are available. \textbf{Note}: the \textit{log} function returns the natural log (ln). If you want the base 10 log, use the function \textit{log10}. Functions are used by putting the argument in ()'s, as in "\textit{sin(x)}."

Operators have different precedence. This means that some operators are used before others when evaluating an expression. For example: "2+3*4" will evaluate to 14 and not 20. This is similar to many calculators, If 20 were the desired result, you would type "(2+3)*4". The power operator (^) has the highest precedence (i.e. it's evaluated first), followed by multiplication and division, and last come addition and subtraction. A more detailed explanation of the use of mathematical operators is given in a later section.

**Simple programs**

A MATLAB program consists simply of the name and then the actual steps you wish the program to perform. This can be seen in the example program shown below. The execution of a program proceeds from the first executable statement through to the end of the program (there are, however, several exceptions to this rule. See section on program control). Each statement is executed in turn.

```matlab
% Velocity
% This program calculates the average speed of an object

time = 4.7;                          % The variable "time" is assigned a value
distance = 2*time^3 - 5*time + 18.3; % Distance form origin is calculated
velocity = Distance/time            % The average velocity is calculated

% End of program
```

http://www.ucl.ac.uk/~ucecesf/local/e862/matlab/introduction.html (2 of 3) [9/21/01 3:35:00 PM]
You'll have noticed in the example program that there is a description attached to almost every line of the program. These descriptions, known as *comments*, are indicated by a starting percentage sign (%). Comments can appear anywhere after the end of a command or by themselves on a line. Everything after the percentage sign is ignored by MATLAB. In MATLAB any comments at the beginning of the file will be called if the user asks for help on the program.

Comments are intended to help the person reading the code understand the function of the code. You should always put comments in your programs, even if you are the only one that will be looking at that code. It is a fundamental law of nature that a program doesn't appear as intuitively obvious to even the programmer an hour after it was written! Comments are absolutely necessary and their liberal use is highly recommended.

Related to comments is the actual indentation of each line. You should also consider indenting certain lines to make it clear how they fit in with the rest of the code. The use of indentation will become clearer when we discuss controlling the flow of execution in a program.

**An Exercise**

You should now attempt the exercise associated with this section. You should have already tried out the introductory exercise.

Click [HERE](http://www.ucl.ac.uk/~ucecesf/local/e862/matlab/introduction.html) for next section.
Estimating distance to horizon

Assume that you are sitting at the beach enjoying the lovely weather we had this summer... You look eastwards and see a ship heading out to sea. The purpose of this exercise is to estimate the distance to the ship just as it disappears below the horizon. For the purposes of this exercise, assume that your eyes are 1 metre above the ground when sitting and that the radius of the earth is 6000 km.

Write a MATLAB program (as a script file) that calculates and prints out the distance to the ship. All constants should be declared as parameter variables.

The theoretical answer is 3464 m.
Multidimensional data structures

Although many problems can be solved through the use of simple variables, as described previously, it is sometimes convenient or necessary to group data together in the form of vectors or matrices. For example, the list of component fractions for a mixture is best described as a set of real numbers $x_i$, $i=1,\ldots,nc$. This section describes the declaration and use of arrays, the entities in MATLAB that allow us to work with vectors and matrices. The introduction of these means that we need to look again at the use of standard mathematical operators in MATLAB.

1. Arrays, vectors, matrices and tables
2. Mathematical operations (revisited)

An Exercise

You should do the exercise associated with this section.

Click HERE for next section.
Arrays

An array is an ordered collection of numbers. Each number can be accessed directly using an index (much like the subscript notation used in mathematics to refer to an element of a vector or a matrix). An array is entered into MATLAB as a list of numbers e.g.

\[
x = [1,3,5,7,9]
\]

or instead of using commas we can use blanks

\[
x = [1 3 5 7 9]
\]

This creates a variable called \( x \) which has elements \( x(1), x(2), ..., x(5) \). Each such element is a real number and can be used directly. To access the third element we type:

\[
\ldots
\]
\[
z = x(3)
\]
\[
\ldots
\]

This code would result in the variable \( z \) assuming a value of 5.

Arrays become useful when one wants to perform the same action on every element of an array. For example, supposed we have an array describing the flows of a set of four components in a mixture. We can find out the total flow by simply adding up the individual flows. One way would obviously be to write a statement of the form

\[
\ldots
\]
\[
\text{Totalflow} = x(1) + x(2) + x(3) + x(4);
\]
\[
\ldots
\]

Though, luckily, MATLAB has a built in function \textit{sum} which can be used in this instance. We write

\[
\ldots
\]
\[
\text{Totalflow} = \text{sum}(x);
\]
\[
\ldots
\]

Vectors
The elements of a horizontal array may be used to represent a row vector. Consequently it may be necessary to define a column vector or vertical array.

A column vector is entered into MATLAB as a list of numbers separated by semi-colons e.g.

\[
y = [2; 4; 6; 8; 10]
\]

\[
y = \\
 2 \\
 4 \\
 6 \\
 8 \\
10
\]

**Matrices**

Thus far we have only introduced one dimensional arrays, however many engineering and mathematical calculations require the use of matrices, which are two dimensional arrays. These are entered in a similar way

\[
>> A = [1 2; 3 4]
\]

\[
A = \\
 1 2 \\
 3 4
\]

We can determine the size of any matrix, using the `size` function

\[
>> size(A)
\]

where the output is

\[
ans = \\
 2 2
\]

This tells us that \(A\) is a matrix with two rows and two columns. Actually, all variable types in MATLAB (scalar variables, and row and column arrays) are in fact matrices of different sizes. So if we determine the size of our row vector, \(x\)

\[
>> size(x)
\]
we are told

\[
\text{ans} = \\
\begin{array}{c}
1 \\
5 \\
\end{array}
\]

So \( x \) is a matrix 1 row by 5 columns.

## Tables

Matrices also provide the easiest way to present data in tabular form. For example we may want to tabulate data showing the relationship between speed and journey time:

\[
\begin{align*}
\text{speed} &= [20 \ 30 \ 40 \ 50 \ 60 \ 70] \\
\text{time} &= [5 \ 3.3333 \ 2.5 \ 2 \ 1.6666 \ 1.4286]
\end{align*}
\]

A simple program that produces two alternate tables from this data is shown below. From the program you should note how to transpose vectors and matrices using the special ' operator. This is particularly useful when you want to convert row vectors into column vectors. The program also illustrates how to merge vectors (of the same size) into a large matrix.

```matlab
% Create table - data in rows
row1 = speed;
row2 = time;
Table1=[[row1;row2]] % Form a matrix from two row vectors

% Create table - data in columns
col1 = speed'; % ' is used to transpose data from row vector to a column vector
col2 = time'; % row vector to a column vector
Table2=[[col1 col2]] % Form a matrix from two column vectors

%End of program
```

The output from the program is

\[
\text{Table1} = \\
\begin{array}{cccccccc}
20.0000 & 30.0000 & 40.0000 & 50.0000 & 60.0000 & 70.0000 \\
5.0000 & 3.3333 & 2.5000 & 2.0000 & 1.6666 & 1.4286
\end{array}
\]
Table2 =

<table>
<thead>
<tr>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.0000</td>
<td>5.0000</td>
</tr>
<tr>
<td>30.0000</td>
<td>3.3333</td>
</tr>
<tr>
<td>40.0000</td>
<td>2.5000</td>
</tr>
<tr>
<td>50.0000</td>
<td>2.0000</td>
</tr>
<tr>
<td>60.0000</td>
<td>1.6666</td>
</tr>
<tr>
<td>70.0000</td>
<td>1.4286</td>
</tr>
</tbody>
</table>

Click HERE for next section.
Matrix arithmetic operations in MATLAB

When earlier we considered the multiplication and addition of simple (scalar) variables, we used normal mathematical operators. However, for calculations with matrix variables we will see that there are two types of arithmetic operations:

- **Matrix arithmetic** which is based on the rules of standard linear algebra. The standard operators are used (+, -, *, /, \, ^).
- **Array arithmetic** which is carried out element-by-element. To distinguish from matrix arithmetic the standard operators are preceded by a full-stop (.* ,./ ,.^).

The results of the operations are generally quite different, and therefore, it is essential to determine which option you require before writing your program.

Matrix arithmetic

Addition and subtraction

Under the standard rules of matrix arithmetic, normal addition and subtraction are carried out on an element-by-element basis. For example

\[
\begin{align*}
\text{>> } & \text{ A = [1 2; 3 4];} \\
\text{>> } & \text{ B = [2 4; 3 5];} \\
\text{>> } & \text{ C = A + B}
\end{align*}
\]

gives the output

\[
C =
\begin{bmatrix}
3 & 6 \\
6 & 9
\end{bmatrix}
\]

and of course the operation C - A gives B.

Multiplication

Matrix multiplication is illustrated by the following program:

\[
\begin{align*}
A & = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}; \\
B & = A * A
\end{align*}
\]
C = B * 2

The output of the program is

B =
  7  10
 15  22

C =
  14  20
  30  44

Notice that the multiplications are carried out under the standard rules of matrix arithmetic and thus the result of the first operation is not simply the square of all the elements but for this example is calculated as below:

\[
B = (1\times1+2\times3) \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}
\begin{pmatrix} 1 & 3 \\ 3 & 4 \end{pmatrix}
\]

Division

In MATLAB both left and right division are possible. So if we wish to divide two scalars then two results are possible:

\[
\begin{align*}
&\text{>>}2\backslash3 \\
&\text{ans} = \\
&1.5 \\
&\text{>>}2/3 \\
&\text{ans} = \\
&0.6667
\end{align*}
\]

Of course we can also use matrix division to reverse a multiplication e.g.

\[
A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix};
\]

\[
C = \begin{bmatrix} 14 & 20 \\ 30 & 40 \end{bmatrix};
\]

\[
D = C/2
\]

\[
E = D/A
\]

this program gives

D =
In general if A is a square matrix then we can say that

\[
A \backslash B = \text{inv}(A) \times B \\
A / B = B \times \text{inv}(A)
\]

Solving linear equations

Perhaps the most useful application of matrix division is in the solution of linear equations. In fact in MATLAB a system of n linear equations in n unknowns can be solved easily using left division (\). (The maths for over or under specified systems of equations is much more complicated and thus we won't consider it.)

Consider a system of two simultaneous linear equations containing two unknowns:

\[
\begin{align*}
2x_1 + 5x_2 &= 3 \\
4x_1 + 5x_2 &= 11
\end{align*}
\]

This can be written in matrix form

\[
A \times X = B
\]

where

\[
A = \begin{bmatrix} 2 & 5 \\ 4 & 5 \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad B = \begin{bmatrix} 3 \\ 11 \end{bmatrix}
\]

Using left division we can write a program that solves this system of equations:

```matlab
% Express system of equations in matrix form
A = [2,5;4,5];
B = [3;11];

% Now solve the system of equation
X = A \ B
% End program
```
The result of the program is

\[
X = \\
\begin{bmatrix}
4 \\
-1
\end{bmatrix}
\]

Hence we have solved the equation to get \(x_1 = 4\) and \(x_2 = -1\).

**Array arithmetic**

**Addition and subtraction**

Matrix addition and subtraction are in fact carried out on an element-by-element basis and so there is no need for separate array addition and subtraction operations.

**Multiplication**

How can we square all the elements in matrix \(A\)?

If we write a program that calculates \(A^2\) (this is equivalent to carrying out the matrix operation \(A \times A\)) then we might write:

\[
A = \begin{bmatrix} 1 & 2; \\ 3 & 4 \end{bmatrix}; \\
B = A^2; \quad \text{or} \quad B = A \times A
\]

however, this gives the output

\[
B = \\
\begin{bmatrix}
7 & 10 \\
15 & 22
\end{bmatrix}
\]

So we need to specify that we want to carry out the operation on an element-by-element basis, hence:

\[
A = \begin{bmatrix} 1 & 2; \\ 3 & 4 \end{bmatrix}; \\
C = A. \times^2; \quad \text{or} \quad C = A \times A
\]

\[
C = \\
\begin{bmatrix}
1 & 4 \\
9 & 16
\end{bmatrix}
\]
In general, the operation \( A(i,j) \times B(i,j) \) will result in a matrix with elements \( a_{ij} b_{ij} \).

**Division**

In a similar way to multiplication if we want to carry out division on an element-by-element basis then we used the array operations. Thus, the operation \( A(i,j) \div B(i,j) \) will result in a matrix with elements \( a_{ij} \div b_{ij} \). Hence

\[
A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}; \\
C = \begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}; \\
D = C \div A
\]

\[
D = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}
\]

**Note:** For both array multiplication and division, unless either variable is a scalar, then both matrices being operated on must be the same size.

If you don't understand the rules of matrix and array arithmetic then it may help to refer to your maths notes or a standard maths textbook.

Click [HERE](http://www.ucl.ac.uk/~ucecesf/local/e862/matlab/arrays-b.html) for next section.
Exercise 3

Working with Arrays

This exercise asks you to write a program to give you some experience of working with arrays, vectors and matrices in MATLAB. However, you must make sure your program is reusable, and you should ensure that it can be used to solve systems with different numbers of components.

Calculating the properties of streams

A stream consists of a set of components for which we have the following information:

- \( f_i \) = the molar flow, in kmol/s, of the i-th component
- \( mw_i \) = the molecular weight, in kg/kmol, of the i-th component

You are to write a program which, given this information for a particular stream, calculates and outputs the following properties:

1. \( F \), the total flow, in kmol/s, of the stream where:
\[
F = \sum_{i=1}^{n_c} f_i
\]
and where \( n_c \) is the number of components in the stream.

2. \( w_i \), the mass flow of each component in kg/s, where:
\[
w_i = f_i \times mw_i
\]

3. \( W \), the total mass flow:
\[
W = \sum_{i=1}^{n_c} w_i
\]

4. and \( x_i \), the mole fraction of each component, where:
\[
x_i = \frac{f_i}{F}
\]

Try this program out for the following 5 component stream:

1. 0.1 kmol/s of Propane, molecular weight 44.1 kg/kmol
2. 0.2 kmol/s of Butane, molecular weight 58.1 kg/kmol
3. 0.08 kmol/s of Pentane, molecular weight 72.2 kg/kmol
4. 0.3 kmol/s of Hexane, molecular weight 86.2 kg/kmol
5. 0.025 kmol/s of Heptane, molecular weight 100.25 kg/kmol

Now run your program to solve the same problem but for a 6 component stream which is the same as the one above but with one other component: 0.13 kmol/s of Water, molecular weight of 18.

**Hint**: You should use the `input` function to request the mass flowrates and molecular weights of each component.
Just as in mathematics, it is convenient to be able to define logical parts of a program as a single entity. This can be because it becomes easier to understand a large program when it is composed of small easily identified segments or because a small piece of code is going to be used in several places in a program.

We have already met a small number of the built-in functions found in MATLAB. However, much of the power of MATLAB is that you can extend the language by writing your own functions. (MATLAB functions are similar to sub-routines found in other programming languages such as FORTRAN)

### Mathematical functions

In mathematics, one can define a **function**, as for example:

\[ f(x) = 3x + 5 \]

This function can then be used by specifying the name of the function and an **argument**. The argument is used to replace the variable \( x \) in the definition of the function by the value that results in evaluating the argument. So, for example, the following arguments give the results shown:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>2+3</td>
<td>20</td>
</tr>
</tbody>
</table>

and so on. Functions can have more than one argument, as in:

\[ g(x, y) = x^2 - y + 10 \]

In this case, using the function requires specifying two arguments where the first one is used in lieu of \( x \) and the second for \( y \). So, for example, the following table shows the result for a variety of argument values:

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g(0,0) )</td>
<td>10</td>
</tr>
<tr>
<td>( g(1,0) )</td>
<td>11</td>
</tr>
<tr>
<td>( g(0,1) )</td>
<td>9</td>
</tr>
<tr>
<td>( g(2,1) )</td>
<td>13</td>
</tr>
</tbody>
</table>

Note, in particular, the difference in result for \( g(1,0) \) and \( g(0,1) \).

### Functions in MATLAB

In MATLAB, one can define functions much as we define them in mathematics. Like script files these are also stored with the extension **.m** but the first line of the file must contain the word **function**. For example, the following code segment defines a MATLAB function which corresponds to the mathematical definition:

```matlab
function g(x, y) = x^2 - y + 10
```

http://www.ucl.ac.uk/~ucecesf/local/e862/matlab/function.html (1 of 4) [9/21/01 3:36:00 PM]
function y = equ(x)
% Any comments immediately following the function declaration
% are used for the HELP file
    y = 3.0*x + 5.0;

In this case, the function equ takes a local variable x and returns a real result, y. The result of the function is whatever value y has when the end of the function is reached.

Note: The name assigned to the function by MATLAB is actually the name given to the saved m-file not that given on the function declaration line. Of course, in most cases, it makes sense to use the same name.

Functions can be called by other functions, script program files or interactively at the command prompt. For example we may have a program that uses the defined function:

% test program
d = input('Please input the value for d ');
Answer = equ(d); % Calls function which carries out the calculation
Answer % Print output to screen

Look carefully at this example and note the following:

- The variables used in calling the function do not have the same names as the variables within the function. The value of d when the function is called is used to replace all occurrences of x in the function body. There is a one-to-one mapping between the arguments given when the function is used and the arguments in the declaration of the function itself.

**Number of Inputs and Outputs**

There is pretty much no limit on the number of inputs and outputs for a MATLAB function. For example, if we consider the multi-variable function above, we have:

\[ g(x, y) = x^2 - y + 10 \]

function z = multiin(x,y)
    z = x^2 - y + 10.0;
end

The above example takes two inputs and calculates a single output. Whereas the following function has multiple outputs but only a single input:

function [y,t] = multiout(x)
    y = 2*x^2;
    t = y - x;
end

In fact, if we applied the above function on array variable, x, this would create two outputs (equivalent to y and t) which were also arrays.
There doesn't even need to be an input or an output. For example, consider the following *unhelpful* error message function, which when called, displays an output to the screen:

```matlab
function errormessage
    disp('There is an error')
    disp('Please rectify')
```

### When to use functions

When should one use a *function*?

In general, a function should be used when there is any piece of code that forms some type of logically cohesive unit (a set of calculations that pertain to one type of quantity for example) or when a piece of code is going to be needed in several places in a program (it's always best to minimise the amount of code you have to repeat as it's easier to fix something if it's only in one place).

### Update on format of a program

With the introduction of functions, the format of a program discussed earlier has changed. A complete program now includes a *main* program (most likely a script file) which will then call a number of functions (which may themselves call other functions) to manipulate the data.

```matlab
%This is the main program

% Assign variables values
x1 = input('Please enter first value ');
x2 = input('Please enter second value ');

% Calculations
a = equ(x1); % call function
b = equ(x2); % call function
answer = a - b;

% Display output
answer
```

Note that the function *equ* (defined earlier) and the MATLAB function *input* are called twice in this program. This type of programming is called *structured programming*.

### Local or Global Variables

The variables used within the body of a function are *local* variables. This means that variables used by a MATLAB function are generally separate from those of other functions and those in the base MATLAB workspace.

This is in contrast to variables created in a script file (or entered at the MATLAB command prompt) which are *global* variables and will remain in the MATLAB workspace until cleared. Global variables can be used by other non-function
An Exercise

Now try the exercise associated with this section.

Click HERE for next section.
Writing a flowsheeting system

Having been introduced to modular programming in the form of MATLAB functions, this exercise is meant to give you a chance to get comfortable with the use of subroutines for decomposing problems into logical pieces.

Consider the (very simple and somewhat silly) flowsheet shown in the figure. It consists of two splitters and a mixer along with 6 streams. The purpose of this exercise is to write a simple flowsheet function that determines what the contents of each stream are, given the feed stream and the split ratios of each of the two splitters.

Write two functions, one to model a splitter and one to model the mixer. These functions should be written so that they can be used to model the flowsheet shown in the figure.

In other words, write a function called `Splitter` which has two arguments and returns two results:

```matlab
function [S1, S2] = Splitter( splitratio, Sinput )
```

where

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sinput</td>
<td>The input feed stream for the splitter. This stream will simply be a list of molar flows.</td>
</tr>
<tr>
<td>splitratio</td>
<td>The amount of the input stream to send to the first output stream (the top one for each splitter in the diagram above). This amount is specified as a ratio, a value between 0 and 1.</td>
</tr>
<tr>
<td>S1</td>
<td>The first output stream of the splitter. This will consist of a list of molar flows.</td>
</tr>
<tr>
<td>S2</td>
<td>The other output stream.</td>
</tr>
</tbody>
</table>
Note that the component flows of $S_1$ and $S_2$ should of course add up to the component flows of $S_{input}$!

Also write a function called \texttt{Mixer} which has the following declaration:

\begin{verbatim}
function Soutput = Mixer(input1, input2)
\end{verbatim}

with a similar interface to the \texttt{Splitter} (but not exactly the same!).

Then write a main function which uses the functions defined above, in the appropriate order and the correct number of times, to determine the contents of each stream in the flowsheet.

\section*{An Algorithm}

The algorithm for this problem is shown below:

\begin{verbatim}
Algorithm Flowsheet
  Given: feed stream
  split1 - split ratio for first splitter
  split2 - split ratio for second splitter
  apply splitter to feed stream using split1 to generate $S_1$ and $S_2$
  apply splitter to $S_1$ to generate $P_1$ and $S_3$
  apply mixer to $S_2$ and $S_3$ to get $P_2$
  output all streams in tabular form
end Algorithm
\end{verbatim}

\section*{The problem}

Use the program you've written to solve the flowsheet assuming that the first splitter has splits the stream exactly in half (a split ratio of 0.5) and that the second sends two thirds to the top stream. The program should be written so that these values are part of the input to the main function, as should be the description of the feed stream. Try this out with a feed of 5 components with flows \begin{verbatim}[ 10 20 13 2 45 ]\end{verbatim}.

The output should list the component flows of all the streams (feed, products, and internal streams), in a well organized and easy to understand tabular form. See the two hints below, these may help you manipulate your data into a tabular form.

\section*{HINTS}

1. Often data is best represented in columns, you can transpose a row vector to a column vector using the
operator ' . e.g.

\[
\begin{bmatrix}
\end{bmatrix}
\]

2. In MATLAB two vectors can easily be combined to give a matrix - this is the most basic way to produce a table. For example:

\[
\begin{bmatrix}
\end{bmatrix}
\]
Making Decisions and Repeating Calculations

The most straightforward way of writing a program consists of typing in a sequence of instructions. Sometimes, however, decisions may have to be made according to certain specified requirements on what formula to use in the calculations, for instance, or even whether it is possible to continue for the given data. MATLAB has a selection of commands that will allow the user to take decisions and transfer the control to other parts of the program accordingly. Often we also find that there are instructions we want to repeat a number of times. Having to write the same instructions over and over while changing the data becomes very tedious but MATLAB has a very helpful command for that purpose. There are three control structures provided by MATLAB to enable this type of behaviour:

1. IF statements
2. Program Loops: FOR and WHILE statements

Exercises

There are two exercises associated with this section after reading about IF statements and Program loops you should try them both. Exercise 5 and Exercise 6.

Click HERE for next section.
Conditional Execution

When writing a program we may be faced with the need to use different equations, for instance, according to what the limiting conditions of validity are for each of them. This is achieved in MATLAB (and in many other programming languages) by using the IF statement.

An IF statement will execute equations subject to a logical condition

Logical Expressions

Statements within an IF Statement are executed subject to a logical condition being satisfied. For example the following logical expression would be TRUE if x was greater than 1, and FALSE if x was less than (or equal to) 1.

\[ x > 1 \]

The list of relational operators and their meanings are:

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>~=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

Expressions can be combined using the logical operators AND, OR, and NOT. The MATLAB symbols for these are:

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td>:</td>
<td>OR</td>
</tr>
<tr>
<td>~</td>
<td>NOT</td>
</tr>
</tbody>
</table>

- The AND operator returns the TRUE value if the expressions on both sides of the operator are true, otherwise it returns FALSE.
- The OR operator returns TRUE if either or both of the expressions are true.
- The NOT operator is applied to a single expression and returns TRUE if the expression is false,
and vice versa.

Consider the following example; if we assume that \( x > y \) is true (i.e. \( x \) is greater than \( y \)) and that \( x < 10 \) is false, the following expressions return the values indicated:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &gt; y ) &amp; ( x &lt; 10 )</td>
<td>\text{FALSE}</td>
</tr>
<tr>
<td>( x &gt; y ) : ( x &lt; 10 )</td>
<td>\text{TRUE}</td>
</tr>
<tr>
<td>( \sim x &gt; y )</td>
<td>\text{FALSE}</td>
</tr>
<tr>
<td>( x &gt; y ) &amp; ( \sim x &lt; 10 )</td>
<td>\text{TRUE}</td>
</tr>
</tbody>
</table>

**IF Statement**

The \texttt{IF} statement is used to determine, when the program is running, whether a block of statements should be executed. It can also be used to execute alternative sequences of statements, depending on alternative conditions. An example of the simplest form of \texttt{IF} statement is shown below:

```java
... 
if x > y
    'X is greater than Y and difference is '
    diff = x-y 
end 
... 
```

The \texttt{IF} statement takes as its argument a logical expression, as described above. If that expression evaluates to \text{TRUE}, the statements that immediately follow the \texttt{IF} statement are executed. If the condition is not true (i.e. it is \text{FALSE}), execution proceeds with the statements after the \texttt{END} statement. In the example, if \( x \) were greater than \( y \), a difference would be calculated and a message printed out. If \( x \) were less than or equal to \( y \), execution would proceed immediately to the rest of the code indicated by the second set of ...'s.

A more general form of \texttt{IF} statement includes a second block of statements which are executed if the logical expression is false.

```java
... 
if x > y
    'X is greater than Y. The difference is '
    diff = x - y 
else 
    'In this case, X is less than Y. The difference is'
... 
```
This case differs from the previous example in that if x is not greater than y, a different method is used to calculate the difference and a slightly different message is printed out to the user. In general, if the argument to the IF statement is FALSE, execution proceeds to the statement immediately following the ELSE statement and continues until the END statement is reached.

A slight variation on this form of IF statement is one in which a series of conditions is tested. An example of this is shown below where the user is asked for the pressure (in atmospheres) of a vessel. This value is then checked against some criteria and if unsuitable, the user is asked to re-enter the data.

```
...  
P = input('Please input the pressure of the vessel ');

if P < 1.0 
 'Sub-atmospheric pressure not allowed.'
 P = input('Please re-enter the pressure of the vessel ');
elseif P > 10.0
     'Pressure is too high!'
 P = input('Please re-enter the pressure of the vessel ');
else 
     'Pressure appears to be within valid range.';
end
...
```

The tests of the logical expressions are made one after the other until one is true. The corresponding block of statements is then executed and the control is passed to the END statement ignoring any other tests that have not been done yet. If none of the conditions is true, the final ELSE clause (if such is present) is executed.

Finally we can have nested IF statements but each has to be fully contained in a block of the next outer IF statement. For example, to compare three quantities:

```
if x > y
    if x > z
        'X would appear to be the largest value.'
    else
        'Z is the largest value.'
    end
end
```
This example is not complete. What code do you need to write to cover all the cases? How can you rewrite this code using logical expressions to make it shorter (and hopefully easier to read!)?

Click [HERE](http://www.ucl.ac.uk/~ucecesf/local/e862/matlab/control-if.html) for next section.
Repetitive tasks

Loops allow the user to repeat a calculation for a number of times. Recall that in a previous section, we showed a very contrived example for demonstrating the effect of rounding errors:

```matlab
format long
x = 0.1
y = 1
z = y - (x+x+x+x+x+x+x+x+x)
```

Note the line where `z` is calculated. It is very tedious to have to write the sum of `x` 10 times in this way, let alone if we want to sum it even more times. It is also difficult to ensure that we've typed the right number of `x`'s! Luckily, MATLAB allows us to replace a line like:

```
z = y - (x+x+x+x+x+x+x+x+x+x)
```

by

```matlab
xsum = 0.0;
for i = 1:10
    xsum = xsum + x
end
z = y - xsum;
```

These two code segments produce identical results. The advantage of the second is that we don't have to manually count out how many `X` values we want to add together. It also means that if we decided to add 20 values, all we would have to do is change the 10 in the `FOR` statement to a 20.

In practice, you will encounter two forms of the program loop: a deterministic one in which the relevant code is executed a predetermined number of times (as in the example above) and one in which the number of times through the loop depends on the behaviour within the loop (for example, loops which terminate when a certain convergence criterion is met such as when using Newton's method for solving a system of nonlinear equations).

Deterministic loops - `FOR`

In a deterministic loop the limits of repetition are defined in advance. To do this we use a `FOR` loop. An example of such a loop was given above and another (slightly less trivial) example is given here:
The FOR statement, in this form, expects an iteration variable or *FOR loop index*, which in the example is the variable \( k \). The rest of the line tells the FOR loop what values this loop index must take. In this case, we have told it to start with the value of 1 and perform each iteration successively using values of 2, 3, and so on until 20.

In general, the starting statement of a *deterministic loop* has the form

\[
\text{FOR variable} = \text{start} : \text{step} : \text{end}
\]

The code between the FOR and END statements will be repeated with values of \( \text{index} \) starting with \( \text{start} \), incremented by \( \text{step} \) for each successive iteration until it exceeds the value of \( \text{end} \). Note, if the step is given a negative value, the FOR loop will continue until the index is less than \( \text{end} \). However, if the step is emitted it will automatically a step value of 1. For example:

\[
\begin{align*}
\text{for } i &= 5:15 \quad \% \text{do for between 5 and 15 with increments of 1} \\
\text{for } j &= 1:2:19 \quad \% \text{do for all odd numbers from 1 to 19} \\
\text{for } k &= n:-1:1 \quad \% \text{start at } n \text{ and count down to 1 inclusive}
\end{align*}
\]

Note that any of \( \text{start} \), \( \text{end} \), and \( \text{step} \) may be expressions as in:

\[
\begin{align*}
\cdots \\
\text{for } i &= 1:2:2n+1 \\
\cdots
\end{align*}
\]

where the end of the loop depends on the value of \( n \).

Also note that it is possible that no iterations through the FOR loop will happen. For example, if \( n \) above had a negative value, it would evaluate to something less than 0 and since the starting value is greater than 0 (with a positive step size), the terminating condition for the FOR loop is met before we even start. In this case, execution would proceed immediately to the code *after* the END.

The value of the loop variable index is incremented at the end of the loop iteration for use in the subsequent iteration. As a result, upon exit from the FOR loop, the value of the index variable will be
some value greater than END (how much greater depends on the step...) It is useful to bear this in mind when subsequent use of the index variable is required. In practice, it is better to not assume that the index variable has any specific value after leaving the FOR loop and hence should not be used.

**Non-Deterministic loops - WHILE**

The second form of loop is when there are no specified limits for the loop parameters. The condition to end the loop calculation is only satisfied during the execution of the loop itself. To do this we use a WHILE loop.

```matlab
% starting time
t_start = 0.0 ;
% end time
% end time
% time step
% time step
| t = t_start; |
| i = 1 ; |

% calculate something using the time t
y = 2*t^3 - 5*t;
% Record output in matrix
tabulate(i,1) = t;
tabulate(i,2) = y;
| i = i +1; |
% increment time using the time step value
| t = t + delta_t; |
end % end while

% Output results to screen
disp(' t    y')
disp(tabulate)
```

Although this example could have been done using a FOR loop, calculating the number of times we expect to go through the loop before hand, it is often easier and more readable to simply let the important variable (the time in this case) control the execution of the loop.

The WHILE loop is terminated when the appropriate condition is met (we reach the value of \( t_{\text{end}} \)).

**Important:** This form of WHILE loop, without an explicit statement of how many times to perform the loop itself, can be dangerous if not used carefully. One **MUST** always, in a case like this, use some counter that ensures that a WHILE loop will eventually terminate. The example above is safe as the time variable will eventually reach and exceed the value of \( t_{\text{end}} \). In the following example, however, there
is no such guarantee:

```matlab
x = 1.0
while x > 0  % leave when value becomes negative
    ! estimate new value of x
    x = x^2 - 10*x + sin(x)
end
...
```

Depending on what the right hand side of the assignment statement evaluates to, the condition for exiting the WHILE loop might never occur. In a case like this, one must always include a mechanism to ensure the WHILE loop is exited, as in the following, safer, alternative:

```matlab
max_counter = 50;
counter = 1;
x = 1.0;
while x > 0     % leave when value becomes negative
    % estimate new value of x
    x = x^2 - 10*x + sin(x);

    % Check to see if we've exceeded maximum iterations allowed
    counter = counter + 1;
    if counter > max_counter
        disp('Maximum iterations allowed has been exceeded!')
        break
    end  % if
end    % while
...
```

This WHILE loop will now always terminate by the time 50 iterations are performed (it may, of course, terminate much sooner, depending on the value of x). Note how we've informed the user that something may have gone wrong. In practice, the error message would hopefully be a lot more informative!!

**Nesting loops**

Just as we've seen an IF statement inside a FOR loop, we can have FOR or WHILE loops inside other such loops. For example:

```matlab
y = 0;
x = 0;
for i=1:10
    y = y + i^2;
```
for j = 1:i
    x = x + j;
end
end

x

y

Here we see an inner FOR loop inside another.
CAPE COURSE: Tabulation of values

The vapour pressure-temperature relationship for many organic substances is reasonably well approximated by the following equation:

\[
\ln P^* = 10\left(1 - \frac{T_b}{T}\right)
\]

where \(P^*\) is vapour pressure in bar or atmospheres, and \(T\) is the Kelvin temperature. \(T_b\) is the 1bar or 1atm boiling point, also in Kelvin.

Now, for mixtures of butane and pentane, write some programs to try out this relationship. Where \(T_b\) at 1atm for butane is 272.5K, and for pentane it is 309.4K.

1. Define a function that calculates the vapour pressure for a given temperature and boiling point. Check the function works by calling it at the MATLAB command prompt.

2. The next objective is to write a program that calculates the vapour pressure of pentane at \(n\) equally spaced points between temperatures \(T_1\) and \(T_2\), where \(n, T_1, T_2\) are variables to which values respectively of 10, 260K and 340K are assigned in your program. There are two ways to this, you should try both of them and compare the results:
   - Firstly use a WHILE or FOR loop for this. Calling the function that you defined above to calculate the vapour pressure of pentane for each temperature. Consider storing the results of each function evaluation in a vector of increasing size each time the loop is executed.
   - Secondly create a row vector of temperatures and use the function to create a vector of vapour pressures. See the hint for a quick way to create a row vector.

3. Select your favoured method from 2 and adapt your program to tabulate together the vapour pressures of both butane and pentane between their normal boiling points.

HINT

- In MATLAB there is a quick way to create a row vector. Try the following, which creates a vector containing values between 1 and 7 by increasing each value by the increment 2.

\[
x = 1:2:7
\]

\[1 \ 3 \ 5 \ 7\]