Introduction to Programming using Matlab

1. What is programming? Languages, Code, Compilers, Machine Code
2. Code conventions, Comments, etc.
3. Variables, Data Types, Mathematical Operators, Arrays
4. Conditional branching
5. Conditional Loops
6. Functions, defining functions, calling functions, returning values from functions
7. Reading data from files
8. Plotting Data

1. What is programming?

Essentially a computer is a machine that processes data. The act of programming (or coding as it is sometimes known) is the act of creating a set of instructions for the machine to follow.

Early computers were programmed with what is called "machine code" which is essentially a large set of binary digits which form a series of instructions that were fed to the machine either on punch cards or tape.

Luckily for us things have progressed a lot over the last 30 years or so and now most programming languages are far more human readable (and are no longer stored in boxes of punch cards). This makes a programmers life a lot easier. In this course we will be using a language/programming package called Matlab.

Essentially Matlab is programming environment/compiler. A compiler takes the human readable code a programmer writes and converts it into machine readable machine code (IE: binary code).

Using "M" Files

Matlab programs are stored in what are known as "M" files. Essentially these are just text files with a filename ending with ".m" EG: myreallyusefulprogram.m

To run a program in Matlab you can open it from the "file" menu and then "run" it from the debug menu or by pressing F5.

NOTE: Due to the configuration of Matlab on the lab machines your .m files are not automatically saved in the network (N:\) drive. Please make sure that you make a point of saving your work on N:\ as this is the backed up network location. Saving work elsewhere may result in the loss of work.

Matlab Help

The Matlab help function is incredibly useful. If at any time you are having problems with a command or operation merely type "help <command>".
EG: "help plot" will bring up a help file about the use of the plot command.

2. Coding Conventions.

Comments

Programs can get quite long and complex so it is always a good habit to put "Comments" into your code. This stops you forgetting what pieces of your code are for, what version of the code you are looking at, etc. It may seem a bit pointless remarking short pieces of code but it is a good habit to get into for the future. Believe me you'll thank me later. In Matlab you can put comments into your code by using the percentage symbol "%" at the start of a line.

EG:

```
% This is a very simple Matlab program
disp ('Hello world!')
```

The first line will be ignored by the compiler (if it didn't it would come up with an error as "This" is not a valid command). The second line would execute and print "hello World!" at the command prompt.

3. Variables

Almost all programs people write are concerned with the creation storage or manipulation of data (and sometimes all 3). So obviously the first thing we are going to need is somewhere to store the data. This is where "Variables" come in.

Essentially a variable is a name given by the programmer to a piece of computer memory within which the program stores some data.

In Matlab we create a variable (and assign it a value) by using a piece of code like this,

```
IntBigCounter = 1000
```

Here the '=' means 'assign the value' of 1000 to the variable IntBigCounter. The order is important. For example the line '1000 = IntBigCounter' will not work.

Note that whenever an assignement occurs the resultant value will be output to the Matlab report screen. If you wish to suppress these reports simply put a semi colon at the end of the line of code.

```
EG: IntBigCounter = 1000 will assign a value of 1000 to IntBigCounter
and will also output the variable name and value to the Matlab
screen. But IntBigCounter = 1000; will merely assign the value.
```

It is best to use a variable name that has some relevance to what we are using it for. In the example above "IntBigCounter" is going to contain an integer that will be used as a counter and so the name "IntBigCounter" makes sense. While it is perfectly possible to assign variable names like a, b and c it is frowned upon as it makes reading your code and understanding what variables are being used for far more awkward in the long run.

To output a variable to screen use the sprintf command again but this time we need to specify a conversion character.

EG:

```
IntBigCounter = 1000;
str=sprintf ('Big Counter Value=%d', IntBigCounter);
disp(str)

or
IntBigCounter = 1000;
disp(['Big Counter Value=' num2str(IntBigCounter)])
```

Will output "Big counter value= 1000" to the screen. The %d inside the quotes specifies that the value of `IntBigCounter` is to be output to screen as a decimal.

Other Example conversion characters.

```
%c Single character
%d Decimal notation (signed)
%e Exponential notation (using a lowercase e as in 3.1415e+00)
%E Exponential notation (using an uppercase E as in 3.1415E+00)
%f Fixed-point notation
%o Octal notation (unsigned)
%s String of characters
%u Decimal notation (unsigned)
```

Further control characters for output can also be used , eg:

```
\f Form feed
\n New line
\r Carriage return
\t Horizontal tab
%% Percent character
```

Variables can also be assigned non numerical values such as "strings", eg:

```
StrFirstName = 'David';
```

Or can be assigned values from other variables, EG:

```
IntBigCounter = 1000;
NumLittleCounter = IntBigCounter/2;
```

In the code above "IntBigCounter" is assigned a value of 1000 and "NumLittleCounter" is assigned a value of half whatever "IntBigCounter" is currently set to (IE: 500).

Another example:

Time = 1;
Time = Time + 0.1

Will output the new value of time to the screen as 1.1. As you can see = is not a mathematical equality as that would imply 0 = 0.1 which is clearly not true.

As you can see there are a number of mathematical operators built into Matlab, these are core elements of how a program is written and I will list some of the simpler ones here (there are a large number of more complex ones listed in the online documentation for Matlab.)

Mathematical Operators

+	Addition
-	Subtraction
^	Exponentiation
*	Multiplication
/	Division
Sin	
Cos	
Tan	
Exp	
Log	

To see a list of all of the operators, type "help ops" in matlab.

Arrays

An array is a type of variable list that is used extensively in Matlab programming, an array can be created by using the a *zeros* command to initialise it.

EG:

```
% 2 new variables are used to store
% the no of rows and columns in the array.
RowNum = 5;
ColNum = 3;

% the next line creates the array "FirstArray" and initialises all
its elements to zero
FirstArray = zeros(RowNum,ColNum);

% The next line sets the value stored at Row 3 Column 1 to equal 4.2
FirstArray(3,1) = 4.2;

% the next line displays the new contents of the array "FirstArray"

FirstArray
```

You can change the value of a consecutive set of array elements as follows

```
FirstArray(1:4,2)=3.7;
```

This will change rows 1 to 4 in column 2 to all equal 3.7. There is also a shorthand to change all elements in an array dimension to equal a value.

Alternately you can assign a series of values to array elements like this

```
FirstArray=[5, 10 ,15, 20]
```

Will assign the values 5, 10, 15 and 20 to the first 4 rows of an array or alternately using

```
FirstArray=[5; 10; 15; 20]
```

Will assign the values 5, 10, 15 and 20 to the first 4 columns of the array. Commas are used to denote rows and semi colons are used to denote columns.

```
FirstArray(:,:)=2.5;
```

Will set every element in FirstArray to equal 2.5

The command "size" returns the number of elements in each dimension of an array

```
FirstArraySize=size(FirstArray)
```

Often a particular entry in an array is used in a calculation. EG:

```
RowWant = 4;
ColWant = 2;

SelectedItem = FirstArray(RowWant,ColWant);

FirstArray
%Outputs the full array to screen

SelectedItem
% Outputs the value contained in Row 4, Col 2 of the Array FirstArray
```

Arrays are a natural way to represent vectors (like position and velocity) and the matrices that appear in solving physical problems. Vectors are simply one dimensional matrices. For example, a position in space can be labelled by three coordinates x, y and z which we could write in two ways in MATLAB:

```
a row vector r=[ x, y, z ]
a column vector c=[ x; y; z ]
```

To enter these at the MATLAB prompt you will first have to assign values to variables x, y and z.

An apostrophe transposes from one to the other in MATLAB:

```
r=c' and c=r'
```

Two-dimensional arrays are matrices with rows and columns. For example:

```
amat = [ x1, x2, x3 ; y1, y2, y3 ]
```

sets up a matrix with 2 rows (the x and y coordinates) of 3 points (each one represented by a column).

Note: another way to set up bigger vectors and matrices is to use the zeros command:

```
array = zeros( NRows, NCols )
columnvector = zeros( NRows, 1 )
rowvector = zeros( 1, NCols )
```

Worked example using a rotation matrix

(the following example will be relevant to Fluid Dynamics, Course MT24A)

Define two points on a flat surface:

```
x = [ 0, 1 ]
y = [ 0, 2 ]
```

In MATLAB you can plot the line between these points readily using the command:

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

Now create a 2 x 2 matrix which contains the coordinates of both points:

```
a = [ x; y ]
```

The line between the points can be rotated by an angle theta by multiplying by the rotation matrix:

```
m = [ cos(theta), -sin(theta) ; sin(theta), cos(theta) ]
```

Note that angles must be in radians (not degrees) in programs. The result of rotation is in matrix:

```
b = m*a
```

Use theta = 0.25*pi and try it in MATLAB.

Check the results by multiplying the matrices yourself on paper. Note that the original points in matrix a can be obtained from b by multiplying it by the inverse of m (see MA1VM notes). MATLAB does this for you using the backslash operator:

```
a = m\b
```

which in this case is equivalent to rotating backwards. The new coordinates are given by the two rows of the b matrix:

```
xnew = b(1,:)
ynew = b(2,:)
```

where the : means all elements along that row.

```
hold on (to overplot on the previous graph)
plot(xnew,ynew)
```

You should see the line has rotated anti-clockwise by the angle specified.

Other Basic Matlab Commands

`Disp`	Used to output strings or variables to the screen
input	Prompts for user input
fprintf	Used for the output of messages and numbers to file.
help	The basic help command
pause	Pauses program or script until user presses any keyboard key.
pause(n)	Pauses program or script for n seconds.
waitforbuttonpress	Pauses program or script until a mouse button or key press.
strcmp	In its basic form, compares two strings EG: strcmp('Thursday','birthday') returns a value of 0.
str2num	Converts a string to a number. EG: x = str2num('digitstring')
num2string	Converts a number to a string. EG: numstring = num2str(numvar)
plot	Plots two series of values against each other EG: plot(time,tempvals)
xlabel	Adds a label to the X axis of a plot EG: xlabel('t (minutes)')
ylabel	Adds a label to the Y axis of a plot EG: ylabel('T (K)')
title	Adds a title label to a plot EG: title('Temp Over Time Plot 1')

An extensive list of Matlab commands can be found at
http://www.engin.umich.edu/group/ctm/extras/commands.html

Exercise 1: (20% of marks)

A. Write a short program that puts the numbers 10,20,30,40,50 in an array and asks a user for a numerical value to be added to these numbers. The program should add this user input value to each item in the array and output the new values to the screen.

Note: By default Matlab creates 2D arrays as does the zeros command. So you will need to make sure you specify a 1D array.

B. Modify the steps in the above vector rotation example to include a third point at position (x=3, y=2) and plot all three points. Rotate the shape obtained anti-clockwise by 0.125*pi radians and plot the results. Rotate the shape by a further 0.125*pi radians using the same matrix and overplot again. Experiment by adding more points and different rotations.

4. Conditional Branching

Most programs are going to require that different operations are performed dependant on the current status of variable in the program. This is where the concept of "conditional branching" comes in. There are several types of conditional branching the two most common of which are "if" and "switch".

The IF statement

If is used when a piece of code is to be executed if a set condition is true.

For example you would only send someone birthday card when it was their birthday, below is a piece of code that simulates this kind of behaviour.

```
if strcmp(stringvar1,'birthday')
        stringvarsendcard = 'yes'
end
```

You'll notice that I have ended the if "branch" with an "end", the code between "if" and "end" is only executed if the condition on the initial branch line is met.

The general structure for an if statement is a follows

```
if (condition == true)
        execute all code that is here
else
        execute all code that is here
end
```

The "else" statement allows the execution of alternate code (between else and end) if the condition is untrue.

The Switch statement

The Switch statement is more complex and is used when there are several possible values for a variable are to be evaluated.

For example there may be a variable called strcardtype which contains a value set elsewhere in the program to a card type (e.g.: Birthday, Christmas, Wedding, New baby boy, New baby girl) the switch statement will evaluate the variable and then execute the code assigned to that case (or value of the variable).

```
strcardtype = 'Wedding';

switch (strcardtype)
   case {'New baby boy','New baby girl'}
      disp('Send a new baby card')
   case 'Birthday'
      disp('Send a Birthday card')
   case 'Christmas'
      disp('Send a Christmas card')
   case 'Wedding'
      disp('Send a Wedding card')
   otherwise
      disp('no need to send a card')
end
```

This code will out put “Send a Wedding card” to the screen using the “disp” statement. Notice that the “otherwise” statement is used to catch all non enumerated possibilities. You will notice that I have only used the “==” or “is equal to” evaluation statement in the above examples. But there are many possibilities available for use.

Please note that switch statements only execute the first appropriate bit of code they come to.

NOTE: Unfortunately the switch statement in Matlab only copes with string values and so cannot be used as a way of switching based on a numeric value.

Relational operators	**Description**
<	Less than
>	Greater than
==	Equal to (Don't confuse with the assignment operator =)
~=	Not equal to

Logic operators	**Description**
~	NOT
&	Logical AND as used in evaluation expressions
|	Logcial OR as used in evaluation expressions

5. Conditional Loops.

While Statement

The 'if' statement allows us to execute a series of commands conditional on some logical expression. The 'while' statement adds repetitive execution to the 'if' statement. The format of the while statement is 'while *condition statement* end' and the statement is repeatedly executed while it is the case that the condition evaluates as true. In other words the statement is executed over and over again until the condition becomes false. Be warned that an error in such a statement might lead to your program looping continuously. If your program is looping, press [Ctrl/c] in the command window to cancel it. The while statement is useful when you do not know in advance how many times an operation needs to be performed. For example, this code finds the smallest power of two which is larger than the number n:

```
n=50;
p=1;

% This while loop exits when the value of p is greater than 50
while (p < n)
    p = 2 * p;
end

%The next line displays the final value of 'p' in this case 64
disp(p);
```

The break statement is sometimes useful to cancel a while loop in the middle of a number of statements:

```
while (1)
    req = input('Enter sum or "q" to quit : ','s');
      % Note that the 's' is needed to return the entered
      % string as a text variable
        if (req=='q')
        % If the input value is q then we break from the while loop
        break;
        end

     % The next line evaluates and displays the entered string
     % as a numeric value
    disp(eval(req));
end
```

For Statement

The 'for' statement is useful when you *do* know how many times you want to repeat a statement. It is just a special form of the while loop. To execute a statement 10 times with a while loop would require statements such as;

```
i=1;
while (i<=10)
    disp(i);
```

```
    i = i + 1;
end
```

This can be written more succinctly with the *for* statement; its basic syntax is 'for *var=sequence statement* end'. For example, the loop above could be written:

```
for i=1:10
    disp(i);
end
```

Note that in a for loop, the variable is set to each value in the sequence in turn and retains that value when it is referred to inside the loop. You can create sequences with increments different to one in the usual way with '*start***:***increment***:***stop*'. You can even loop through the values in an array. For example:

```
for even=2:2:100 …
for primes=[ 2 3 5 7 11 13 17 19 23 ] …
```

For loops are executed efficiently without actually building an array from the sequence. This means that a loop of 1 million repetitions doesn't require an array of size one million. You can use the *break* statement in for loops as well as in while loops.

6. Functions

Up until this point, every line of code we've shown you has done a simple task, such as performing an arithmetic operation, or checking a Boolean condition, or assigning to a variable. Functions allow you to do a whole lot in one line of code. Instead of performing a simple task, a single line of code can call another previously written .m file which can execute any number of commands at once. This allows you to split complex problems up into "Black boxes" to perform specific operations, thus helping to break the problem down into more manageable chunks.

The first line of a function m-file must be of the following form.

function [output_parameter_list]= Function_name(input_parameter_list)

The first word must always be ``function''. Following that, the (optional) output parameters are enclosed in square brackets [ ]. Note also that the "=" in the function definition is also only included if there are one or more output parameters.

If the function has no output_parameter_list the square brackets and the equal sign are also omitted. The function_name is a character string that will be used to call the function. The function_name **must also be** the same as the file name (without the ``.m'') in which the function is stored. In other words the MATLAB function, ``func1'', must be stored in the file, ``func1.m''. Following the file name is the (optional) input_parameter_list.

There can be exactly one MATLAB function per m-file.

Here is a trivial function, addtwo.m

```
function addtwo(x,y)
%  addtwo(x,y)  Adds two numbers, vectors, whatever, and
%               print the result = x + y
x+y
```

Here is another simple function "traparea.m" with three input parameters and one output parameter. Since there is only one output parameter the square brackets may be omitted.

```
function area = traparea(a,b,h)
%  traparea(a,b,h)   Computes the area of a trapezoid given
%                    the dimensions a, b and h, where a and b
%                    are the lengths of the parallel sides and
%                    h is the distance between these sides
%  Compute the area, but suppress printing of the result
%  notice that at the end of the function the value of "area" is
%  returned to the calling program
area = 0.5*(a+b)*h;
```

Calling Functions

A generalised version of a Matlab function call looks like this:

```
[output1, output2, ...,outputN] = functionname(in1, in2, ..., inN)
```

Where "output1, etc" are the returned values from the function (there may also be no value or multiple values returned dependant on the functions definition), "functionname" is the name of the .m file, and "in1,in2,etc" are the input values supplied from the main program.

EG of a how a function call might look in your program:

```
L=3;
W=4;
H=1;
BoxAreaResult = BoxArea(L,W,H);
```

Exercise 2: (60% of marks)

Write a short program that asks a user to input their birthday in an appropriate format and then tells them their age and star sign. Make sure the program catches incorrect input and asks the user to input valid data. Consider month lengths and leap years during the date validation process.

7. Reading data from files into Matlab

The easiest way to get data into Matlab is to use the LOAD function.

```
load mydata.dat;          %  read data into mydata matrix
month = mydata(:,1);      %  copy first column of mydata into month
precip = mydata(:,2);     %  and second column into precip
```

8. Plotting Data

Data can be plotted by simply using the "plot" command; plots can further be visually improved using various properties of the plot command such as labels, line types and thicknesses, etc. For a full rundown on the plot command simply type "help plot" or look at the online documentation.

Exercise 3: (20% of marks)

Take data file "aug03_temp.csv" from the website. Read it into Matlab and programmatically plot the temperature data correctly on a time scale (Time is in decimal hours 0 -23) with appropriate labelling and colouration. Show knowledge of the plot function and its other properties (title, legend, grid, LineWidth, xlabel, ylabel, etc).

MT12c Marking Structure

Percentage of total marks per exercise

- Exercise 1: 20%
- Exercise 2: 60%
- Exercise 3: 20%

Each exercise is marked according to the following criteria

- Handing in computer files (.m text files) 10%
- Proper Commenting of Code 10%
- Program Structure and Readability 20%
- Program functionality 30%
- Testing of program (Making sure all outcomes have been covered) 30%

Handing In instructions

The exercises be handed in by the Friday of week 4.

- All ".m" program files should be sent as attachments to D.S.J.Cunningham@rdg.ac.uk