Laboratory 2 Programming in MATLAB

This laboratory describes the basics of programming in MATLAB. Topics discussed include M-files, control flow, structures and cell arrays, and more on graphics. You should read these notes before coming to the laboratory and be prepared to do the exercises.

M-File Basics

Files that contain MATLAB code are usually referred to as **M-files**. There are two kinds of M-files: **script** files and **function** files.

Script M-files are simply a list of commands such as you would enter interactively in the Command window. They are useful for simple testing of algorithms and performing tasks that need to be done only once, twice, or a few times at most. Variables created when a script file is run reside in the base workspace.

Function M-files provide the foundation for programming in MATLAB. They can accept input arguments and return output arguments. Local variables used within a function are not accessible outside the function. However, a number of functions and the base workspace can share variables if they are defined as global.

MATLAB has a powerful editor/debugger which can be accessed via the **File** menu or via the task bar or using the edit command. When you write a M-file, you must give it a name and save it before it can be run. It will be saved in the current directory. For convenience, it is suggested that you use the directory provided on the local hard disk during the session and save files to your own account, if you wish, at the end of the session.

When you run a M-file, MATLAB looks in the current directory for a file of that name and then in its prescribed search path. It is best to avoid using names which already exist for MATLAB functions.

Here is a simple script M-file which plots a sine wave:

```
% These lines are comment lines which
% will be displayed when the help function is used
% Joe Bloggs  16 Jan 2018
%
x = linspace(0, 5*pi, 100);
y = sin(x);
plot(x, y)    % Comments can also be put here
```

Lines beginning with the percentage sign % are comments ignored by the MATLAB interpreter. The first batch of such lines is displayed in response to the help command used with the filename, a feature which can be very useful. To execute the file, enter the filename (without the .m) in the Command window.

Exercise 1

What does the function linspace do? Type help linspace at the command prompt to get MATLAB to display the help comments. What does the following command do?

```
x = linspace(0,48,1024);
```

Type “edit mysinewave” at the command prompt and enter in the example code above to draw a sine wave. Run the script. What is the length of the array y ?

Here is a simple function **sumcubes** that accepts two array inputs and returns the array consisting of the element by element sum of their cubes:

```
function res = sumcubes(x, y)
% SUMCUBES calculates the sum of cubes
%   RES = SUMCUBES(X, Y) returns the sum of the cubes of X
%   and Y, element by element.
%   Jane Bloggs 16 Jan 2018
%
res = x.^3 + y.^3;
```

Several points are worth noting about this example.

The “sumcubes” in the first line is a dummy; when you run the file you use the *name of the file*, which can be different. To avoid confusion you should make a practice of making the function and file names the same.

The .^ operator has been used so this function will work with vectors and arrays (on an element by element basis). If only a ^ had been used, it would work only for scalars (and also for square matrices but with a completely different result because the matrices, not their elements, are cubed!).

Another example of a function file **decsort** for sorting in descending order:

```
function [y, p] = decsort(x)
% DECSORT  Sort in descending order.
%   For vectors, DECSORT(X) sorts the elements of X in descending order.
%   For matrices, DECSORT(X) sorts each column of X in descending order.
%   [Y, P] = DECSORT(X) returns an array Y with each column sorted
%   plus a set of permutation indices P that can be used to obtain Y from X.
%   Uses SORT. See help on SORT for more information.
%
%  Fulana Detail 18 Jan 2016
%
[y, p] = sort(-x);
y = -y;
```

This function takes one input argument x and returns a sorted array y together with a set of permutation indices p (`x(p)` will produce the sorted array `y`). The MATLAB built-in function `sort` is used. Because `sort` only sorts in ascending order, the negative of the input array is used.

This function will do something sensible when the input is vector or array, real or complex (because of the properties of the `sort` function). If x is a vector, the elements of the vector are sorted. If x is a matrix, the columns are sorted individually. Many MATLAB functions work in this way on array arguments.

A function can have two or more outputs (as here), separated by commas and enclosed in square brackets. The function can be executed without output arguments in which case the first output variable is returned in the variable `ans` and information about the remaining output variables is lost.

Exercise 2

Type “edit sumcubes” at the command prompt and enter the code for the function sumcubes. Try running the function sumcubes with different arguments. What is the difference between a script file and a function file in MATLAB? Do MATLAB functions require a return statement in order to produce an output variable? Create the function decsort given above. How many outputs does the function decsort have? What happens when you type the command:

```
>> tmp = decsort([1 3 2 5 9 6]);
```

What happened to the other output? Is it possible to get the output ‘p’ without the output ‘y’?

Control Flow

There are a number of methods for controlling program flow in MATLAB: **for** loops, **while** loops, **if-else-end** constructions, **switch-case** constructions and **try-catch** blocks. They will be easily understandable by those who have worked with languages such as C or Java.

For Loops

The syntax of the for loop is

```
for variable = expr
    statements
end
```

The columns of expr are stored one at a time in the variable and then the statements are executed. Commonly, expr is a row vector and variable is scalar. As an example, suppose that we need the values of the sine function at eleven evenly spaced points $\pi n/10$, for n = 0, 1, …, 10. These numbers can be generated using the following for loop:

```
for n = 0:10
    x(n+1) = sin(pi*n/10);
end
```

Because MATLAB is an interpreted language (not compiled), the for loop incurs substantial computational overhead costs. Thus, it should be used only when non-looping methods cannot be applied. Consider the problem of creating a 10 by 10 matrix A where $A(m,n) = \sin(m)\cos(n)$. Using nested for loops, we could use the following code:

```
A = zeros(10, 10);
for m = 1:10
    for n = 1:10
        A(m, n) = sin(m)*cos(n);
    end
end
```

A loop free version of this code using matrix multiplication might look like this:

```
k = 1:10;
A = sin(k)'*cos(k);
```

The second method is far more efficient. The technique of eliminating loops is known as **vectorization**. It should be used whenever possible. If loop methods must be used

to generate a matrix, it is good practice to create the entire matrix in one step, as in the first command of the first method. Otherwise the matrix is produced incrementally and inefficiently.

While Loops

While loops are used when the programmer does not know the number of repetitions a priori and finishing depends on a logical condition. The syntax is

```
while expr
   statements
end
```

The statements are executed while the real part of expr has all non-zero elements.

Here is a simple problem that requires use of a while loop. Suppose that the number π is divided by 2 and the resulting quotient divided again by 2, and so on, until the quotient is less than or equal to 0.01. What is the largest quotient that is less than 0.01 on the continued division of π by 2 ?

```
q = pi;
while q > 0.01
  q = q/2;
end
```

The result is q = 0.0061.

If-Else-End Construct

The simplest form of syntax for the if-statement construct is:

```
if expr
   statements
end
```

The statements are executed if all elements of expr are True (non-zero, usually 1). A more general construct is

```
if expr1
   statements1
elesif expr2
   statements2
else
   statements3
end
```

As an example, we consider the problem of calculating Chebyshev polynomials of the first kind $T_n(x)$, $n = 0, 1, \ldots$. They can be defined recursively by:

$$T_n(x) = 2xT_{n-1}(x) - T_{n-1}(x), \;\; n \geq 2 \;\text{ with }\; T_0(x) = 1, \;\; T_1(x) = x$$

The following function uses this relationship:

```
function c = cheb(n)
% CHEB calculates coefficients of the Chebyshev polynomial
% of the first kind of order n.
% c is the row vector of coefficients in descending order
%
```

```
t0 = 1;
t1 = [1 0];
if n == 0
   c = t0;
elseif n == 1
   c = t1;
else
   for k = 2:n
      c = [2*t1 0] - [0 0 t0];
      t0 = t1;
      t1 = c;
   end
end
```

The result of `c = cheb(5)` is `c = [16  0 -20  0  5  0]`, ie $T_5(x) = 16x^5 - 20x^3 + 5x$.

You might be tempted to write `cheb` as a recursive function (one which calls itself):

```
function c = cheb(n)
% Recursive version
if n == 0
   c = 1;
elseif n == 1
   c = [1 0];
else
   c = [2*cheb(n-1) 0] - [0 0 cheb(n-2)];
end
```

but this version is much slower because of the overhead in function calls.

Switch-Case Construct

The syntax of the switch-case construct is:

```
switch switch_expr
   case case_expr,
        statements1
   case {case_expr1, case_expr2, case_expr3,...}
        statements2
    otherwise,
         statements3
end
```

The switch statement executes the set of statements where the switch expression matches the case expression. The second case expression shown is a cell array (described below); here statements 2 are executed if any of the elements of the cell array match. If there is no match, statements3 are executed.

Try-Catch Blocks

Try-catch blocks in MATLAB provide error trapping capabilities (similar to Java). The syntax is:

```
try
```

```
    statements1
catch
    statements2
end
```

Normally, only statements1 are executed with control passing then to the `end` statement. If an error occurs in statements1, however, statements2 are executed.

Exercise 3

Write a MATLAB function that takes a one-dimensional array of numbers (either a row or column vector), and removes all of the neighboring duplicated numbers. For example, the array [1 2 2 2 3 0 1 0 0 4] becomes [1 2 3 0 1 0 4]. The function should return the answer as a one-dimensional array of numbers in the same format as the input. Your program should use a loop command.

Exercise 4

Write a MATLAB function that takes two row vectors *a* and *b*, not necessarily of the same length, and returns the row vector obtained by interleaving the two input. For example, if the first vector is [1 3 5 7 0 0] and the second is [-2 -5 6], the output vector is [1 -2 3 -5 5 6 7 0 0]. Your function should work for an empty vector. Program loops are allowed.

Exercise 5

Because MATLAB is an interpreted language, programs with for loops (or any type of loop for that matter) run extremely slow. Writing MATLAB code that gets around using for loops is known as “vectorizing” the program code. You will now practice reverse engineering. Explain in detail and in your _own_ words why the following code with no program loops solves Exercise 3.

```
function x = ex3(y)
% Exercise 3  Laboratory 2
[nr nc] = size(y);
if min(nr, nc) ~= 1
    error('Input must be a vector');
end
n = max(nr, nc);
z = y(:)';              % make sure we have a row vector
a(2:n) = z(1:n-1);      % shift one to right. a(1) is set to 0
b = a ~= z;             % b is row vector, =1 if elements not equal
b(1) = 1;               % need this for case y(1) = 0
x = y(b);               % ok even if y is a column
```

Exercise 6

Explain in detail and in your _own_ words why the following code with no program loops solves Exercise 4.

```
function x = ex4(a, b)
% Exercise 4  Laboratory 2
na = length(a);
nb = length(b);
n = min(na, nb);
```

```
x(1:2:2*n-1) = a(1:n);
x(2:2:2*n) = b(1:n);
x = [x a(n+1:na) b(n+1:nb)];
```

Structures and Cell Arrays

Structures and **cell arrays** provide a way of organising dissimilar but related types of data into one data entity. In a structure, field names are used to organise and access the data. A cell array is an array of cells, each of which acts a container for data; cells are accessed by array indexing operations.

Structures

A structure can be created by assigning data to individual fields. In the following example we create a data structure called wheat:

```
>> wheat.current_price = 100;
>> wheat.cash_transactions = 25000;
>> wheat.futures = 30;
>> wheat.contact_name = 'Ellie Barley'  % no semi-colon so will display
wheat =
              current_price: 100
          cash_transactions: 25000
                    futures: 30
              contact_name: 'Ellie Barley'
```

A list of a structure's field names can be obtained using the function fieldnames:

```
» fieldnames(wheat)
ans =
   'current_price'
   'cash_transactions'
   'futures'
   'contact_name'
```

The fields of a data structure are accessed using the field names, for example:

```
» x = wheat.cash_transactions
x =
     25000
```

A field can be deleted from a structure:

```
» wheat = rmfield(wheat,'futures')
wheat =
        current_price: 100
    cash_transactions: 25000
        contact_name: 'Ellie Barley'
```

A structure can also be built using the struct function.

You can have arrays of structures; the component structures must all have the same number of fields with the same names.

Cell Arrays

Cell arrays can be created by using assignment statements, using either **cell indexing** or **content indexing**. The following example shows the differences between these two techniques.

Using cell indexing:

```
>> A(1,1) = {[1 2; 3 4]};
>> A(1,2) = {1+j};
>> A(2,1) = {'Jane Bloggs'};
>> A
A =
   [2x2 double]    [1.0000+ 1.0000i]
   'Jane Bloggs'                []
```

Note that the content of cell (2,2) is automatically set to the empty array [].

Using content indexing:

```
>> A{1,1} = [1 2; 3 4];
>> A{1,2} = 1+j;
>> A({,1} = 'Jane Bloggs';
>> A
A =
   [2x2 double]    [1.0000+ 1.0000i]
   'Jane Bloggs'                []
```

One way to look at the difference is that round brackets () identify cells and curly brackets {} access the contents. Observe the difference between

```
>> x = A(1,1)
x =
   [2x2 double]
```

where x is a 1×1 cell array containing a 2×2 double array, and

```
>> x = A{1,1}
x =
    1    2
    3    4
```

where x is a 2×2 double array.

MATLAB normally displays a condensed form (only simple contents are displayed in full). The command `celldisp(A)` will display the full contents.

The cell array above could also have been created using

```
>> A = {[1 2; 3 4], 1+i; 'Jane Bloggs', []};
```

or by creating an empty cell array using `A = cell(2)` or `A = cell(2,2)` and assigning the contents afterwards.

Cell arrays are useful in implementing function input arguments that can contain items of different types. For example, the following function calculates the RMS value of a signal or a cell array of signals:

```
function rms_value = rms(sig)
% RMS  Root Mean Square
```

```
%   RMS_VALUE = RMS(SIG) returns the root-mean-square value of the
input.
%   The input can be either a cell array of signals or a single signal. Each
%   signal is a 1D array.
%   FDT 19/3/17
%
if (iscell(sig) == 1)    % iscell returns 1 if sig is a cell array
   for ii = 1:length(sig)
      rms_value(ii,1) = sqrt(mean(sig{ii}.*sig{ii}));
   end
else
   rms_value = sqrt(mean(sig.*sig));
end
```

We can input a cell array containing signals of different lengths:

```
>> sig{1} = randn(1, 100);
>> sig{2} = randn(1, 1000);
>> sig{3} = randn(1, 500);
>> rms(sig)
ans =
   1.0025
   1.0190
   1.0181
```

Exercise 7

Create a structure variable in MATLAB that contains your name, height, and email address.

Exercise 8

Create a cell array variable in MATLAB that contains the following three arrays:

[1 2 3 4],

$$\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix},$$

$$\begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}.$$

More on Functions

Function Arguments

For serious programming, it is often a good idea to carry out checks on the input arguments. Inside any function, the pre-defined function nargin can be used to determine the number of input arguments:

```
ni = nargin;   % ni = number of inputs
```

Checks are also often carried out on the types of input arguments, etc.

Here is the function **sumcubes** modified to check that there are two inputs and that they are the same size:

```
function res = sumcubes(x, y)
%
if nargin ~= 2
   error('Need two inputs');
end
if ~all(size(x)==size(y))
   error('Inputs must be the same size');
end
res = x.^3 + y.^3;
```

The error function causes the function to terminate and the specified error message to be displayed.

The expression all(size(x)==size(y)) evaluates to 1 (true) if and only if x and y are the same size. If you have trouble understanding this expression, note that: size(x) and size(y) are 1×2, so size(x)==size(y) is equal to [1 1] if these two are equal, and all returns 1 if all entries of size(x)==size(y) are non-zero. An alternative way to write this line is

```
if size(x,1)~=size(y,1) | size(x,2)~=size(y,2)
```

because size(•,1) and size(•,2) return the first dimension (number of rows) and second dimension (number of columns) respectively.

A function can accept a **variable number of input arguments** if varargin is specified as an input (if there are other inputs, varargin must be the last in the list). varargin is a pre-defined cell array whose ith cell is the ith argument starting from where varargin appears. For example, suppose a function is specified by funcname(n, varargin). If it is called using funcname(10, 0.2, [1 3]), varargin{1} = 0.2 and varargin{2} = [1 3]. (Before varargin appears the variables have been specifically named—in the example n=10.)

We can rewrite **sumcubes** to handle a variable number of inputs as follows:

```
function res = sumcubes(varargin)
%
ni = nargin;   % number of inputs
% Check that there is at least one input
if ni < 1
   error('Need at least one input');
end
% Check that all inputs are the same size
dim = size(varargin{1});
for n = 2:ni
   if ~all(size(varargin{n}) == dim)
      error('Inputs must be same size');
   end
end
% Calculate result
res = 0;
```

```
for n = 1:ni
   res = res + varargin{n}.^3;
end
```

Subfunctions

A function M-file can contain more than one function. The first function is the **primary function** invoked by the M-file name. Any other functions are **subfunctions** which are visible only to the primary function and other subfunctions in the same file.

```
function c = mylcm(a, b)
%   C = MYLCM(A, B) Returns the least common multiple (lcm) c of
%   two integers a and b. Checks that a and b are integers using the
%   subfunction ISINT.
if isint(a) & isint(b)
   c = a.*b/gcd(a,b);
else
   error('Input arguments must be integers');
end
%
function k = isint(x)
% Checks whether or not x is an integer.
% If it is an integer, a 1 is returned. If not, a 0 is returned
k = abs(x - round(x)) < realmin;
```

Using Feval

At times it is useful to pass a function to another function for evaluation. MATLAB supplies the function feval for this purpose. One way feval can be used is with a character string name of the passed function. For example

```
>> a = feval('myfunc', x, y)
```

is equivalent to

```
>> a = myfunc(x, y)
```

An alternative is to use **function handles**. A function handle can be constructed using the character @ before the function name. The following command is equivalent to the previous two:

```
>> a = feval(@myfunc, x, y)
```

Here is the code for a function **plotlots** which accepts a function handle and a data array and calculates and plots values of the function:

```
function plotlots(fhandle, x)
plot(x, feval(fhandle, data))
```

You could plot $\exp(x)$ for $0 \le x \le 2$ using with

```
Plotlots(@exp, 0:0.01:2)
```

Feval can also be used with **in-line** functions. In-line functions are not defined in the usual way, but using the function inline. In the following example, myfunc is an in-line function and can be called by feval as shown.

```
>> myfunc = inline('x^2 + (1-y)^2', 'x', 'y')
```

```
myfunc =
    Inline function:
    myfunc(x,y) = x^2 + (1-y)^2
>> res = feval(myfunc, 1.0, 5.0)
res =
   17
```

It can also be used without feval:

```
>> res = myfunc(1.0, 5.0)
res =
   17
```

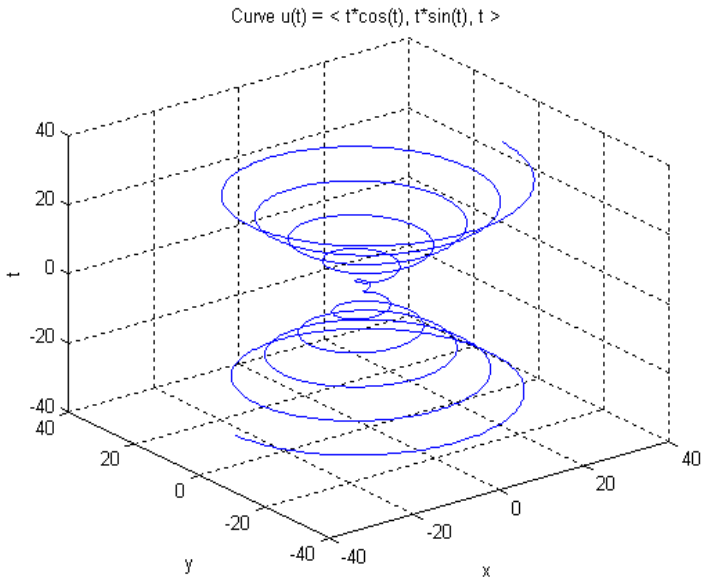
More on Graphics

The topics covered here include 3D graphics. For more information type `help graph3d`.

3D Line Graph

The function plot3 is similar to plot except it takes triples instead of pairs and plots in a 3D space. As an example, the graph of $r(t) = (t\cos(t), t\sin(t), t)$, a parameterized space curve, can be plotted over the interval $(-10\pi \le t \le 10\pi)$ using:

```
t = -10*pi:pi/100:10*pi;
x = t.*cos(t);
y = t.*sin(t);
plot3(x,y,t);
title('Curve u(t) = < t*cos(t), t*sin(t), t >')
xlabel('x')
ylabel('y')
zlabel('t')
grid
```



Plotting Functions of Two Variables

Suppose we wish to display a function of two variables $z = f(x, y)$. The plot of z versus x and y is a surface in 3 dimensions.

To plot such a function, we need to generate a matrix of x and y values. The `meshgrid` function is supplied for this purpose. Suppose that wish to plot over a grid with x and y ranges as defined by:

```
» x = [0 1 2];
» y = [10 12 14];
```

Using meshgrid:

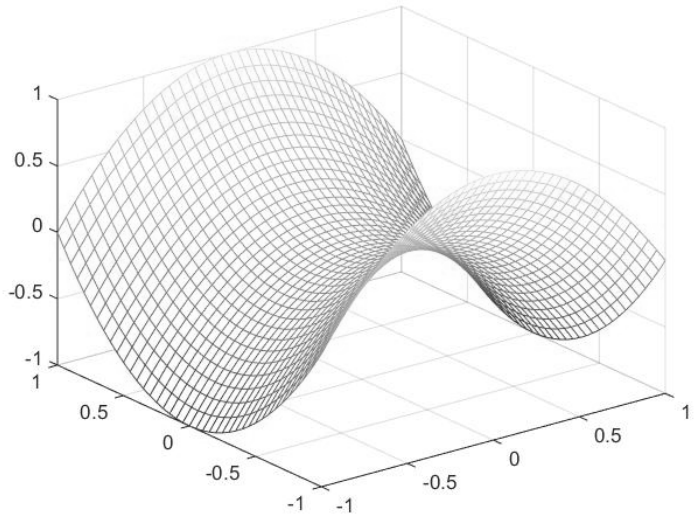
```
» [xi, yi] = meshgrid(x,y)
xi =
     0     1     2
     0     1     2
     0     1     2

yi =
    10    10    10
    12    12    12
    14    14    14
```

Note that the matrix xi contains replicated rows of the array x while yi contains replicated columns of y. The z-values of the function can be computed from the arrays xi and yi.

As an example, to plot the hyperbolic paraboloid $z = y^2 - x^2$ over the square $-1 \le x \le 1, -1 \le y \le 1$:

```
x = -1:0.05:1;
y = x;
[xi, yi] = meshgrid(x,y);
zi = yi.^2 - xi.^2;
mesh(xi, yi, zi)
```



As shown in the plot, the function mesh joins points with straight lines.

Other functions for plotting these type of data include meshc (mesh plus contour lines), surf (uses coloured surfaces), surfc (surface plus contour lines), surfl (surface with lighting effects)

Handle Graphics

A lot can be done with the basic graphics techniques already covered. This section is included for interest.

Every component of a visual entity of MATLAB is an **object** which has an identifier or **handle** associated with it. Low level graphics functions allow the **properties** of an object to be modified. The objects are arranged in a hierarchy. The root object is the computer screen; a window is a child of the root; axes are children of a window; and lines and text are children of axes.

Suppose, for example, that we wish to create a surface plot using the xi, yi and zi of the previous example. The commands

```
figure          % create a window
hfig = gcf;     % use get current figure function to get the handle
set(hfig,'Color','w');   % use the handle to set the background colour
```

set up a figure window and set the background colour to white. The three commands could be rolled into one:

```
hfig = figure('Color','w');
```

Now we can plot and insert a title, and change the font size of the title to 14 point

```
surf(xi, yi, zi);
title('Hyperbolic paraboloid z = y^2 - x^2')
h = get(gca, 'Title');
set(h, 'FontSize', 14)
```

gca returns a handle to the current axes, so h is a handle to the title of these axes. The last command changes the font size of the title. Similarly, we could get a handle to the x-axis label:

```
xlabel('x')
h = get(gca,'xlabel');
```