MATLAB by Example

G. Chand (revised by Tim Love)

July 24, 2006

1 Introduction

This document[1] is aimed primarily for postgraduates and project students who are interested in using MATLAB in the course of their work. Previous experience with MATLAB is not assumed. The emphasis is on "learning by doing" - try the examples out as you go along and read the explanations after. If you read the *online version*[2] you can paste the scripts into your editor and save a lot of time.

2 Getting started

On the teaching system you can type `matlab` at a Terminal window, or look in the `Programs/ Matlab` submenu of the `Start` menu at the bottom-left of the screen. Depending on the set-up, you might start with several windows (showing documentation, etc) or just a command line. This document will focus on command line operations. In the examples below, >> represents matlab's command-line prompt.

3 The MATLAB Language

The MATLAB interface is a command line interface rather like most BASIC environments. However MATLAB works almost exclusively with matrices : scalars simply being 1-by-1 matrices. At the most elementary level, MATLAB can be a glorified calculator:

[1]Copyright ©1996 by G. Chand, Cambridge University Engineering Department, Cambridge CB2 1PZ, UK. e-mail: tpl@eng.cam.ac.uk. This document may be copied freely for the purposes of education and non-commercial research.

[2]http://www-h.eng.cam.ac.uk/help/tpl/programs/matlab_by_example/matlab_by_example.html

```
>> fred=6*7
>> FRED=fred*j;
>> exp(pi*FRED/42)
>> whos
```

MATLAB is case sensitive so that the two variables `fred` and `FRED` are different. The result of an expression is printed out unless you terminate the statement by a '`;`'. `j` (and `i`) represent the square root of -1 unless you define them otherwise. `who` lists the current environment variable names and `whos` provides details such as size, total number of elements and type. Note that the variable `ans` is set if you type a statement without an '`=`' sign.

4 Matrices

So far you have operated on scalars. MATLAB provides comprehensive matrix operations. Type the following in and look at the results:

```
>> a=[3 2 -1
0 3 2
1 -3 4]
>> b=[2,-2,3 ; 1,1,0 ; 3,2,1]
>> b(1,2)
>> a*b
>> det(a)
>> inv(b)
>> (a*b)'-b'*a'
>> sin(b)
```

The above shows two ways of specifying a matrix. The commas in the specification of `b` can be replaced with spaces. Square brackets refer to vectors and round brackets are used to refer to elements within a matrix so that `b(x,y)` will return the value of the element at row `x` and column `y`. Matrix indices must be greater than or equal to 1. `det` and `inv` return the determinant and inverse of a matrix respectively. The `'` performs the transpose of a matrix. It also complex conjugates the elements. Use `.'` if you only want to transpose a complex matrix.

The `*` operation is interpreted as a matrix multiply. This can be overridden by using `.*` which operates on corresponding entries. Try these :

```
>> c=a.*b
>> d=a./b
>> e=a.^b
```

For example c(1,1) is a(1,1)*b(1,1). The Inf entry in d is a result of dividing 2 by 0. The elements in e are the results of raising the elements in a to the power of the elements in b. Matrices can be built up from other matrices:

```
>> big=[ones(3), zeros(3); a , eye(3)]
```

big is a 6-by-6 matrix consisting of a 3-by-3 matrix of 1′s, a 3-by-3 matrix of 0′s, matrix a and the 3-by-3 identity matrix.

It is possible to extract parts of a matrix by use of the colon:

```
>> big(4:6,1:3)
```

This returns rows 4 to 6 and columns 1 to 3 of matrix big. This should result in matrix a. A colon on its own specifies *all* rows or columns:

```
>> big(4,:)
>> big(:,:)
>> big(:,[1,6])
>> big(3:5,[1,4:6])
```

The last two examples show how vectors can be used to specify which non-contiguous rows and columns to use. For example the last example should return columns 1, 4, 5 and 6 of rows 3, 4 and 5.

5 Constructs

MATLAB provides the for, while and if constructs. MATLAB will wait for you to type the end statement before it executes the construct.

```
>> y=[];
>> for x=0.5:0.5:5
y=[y,2*x];
end
```

```
>> y
>> n=3;
>> while(n ~= 0)
n=n-1
end
>> if(n < 0)
-n
elseif (n == 0)
n=365
else
n
end
```

The three numbers in the `for` statement specify start, step and end values for the loop. Note that you can print a variable's value out by mentioning it's name alone on the line. ~= means 'not equal to' and == means 'equivalent to'.

6 Help

The `help` command returns information on MATLAB features:

```
>> help sin
>> help colon
>> help if
```

`help` without any arguments returns a list of MATLAB topics. You can also search for functions which are likely to perform a particular task by using `lookfor`:

```
>> lookfor gradient
```

The MATLAB hypertext reference documentation can be accessed by typing `doc`.

7 Programs

Rather than entering text at the prompt, MATLAB can get its commands from a *.m* file. If you type `edit prog1`, Matlab will start an editor for you. Type in the following and save it.

for x=1:10
y(x)=x^2+x;
end
y

The step term in the `for` statement defaults to 1 when omitted. Back inside MATLAB run the script by typing:

```
>> prog1
```

which should result in vector y being displayed numerically. Typing

```
>> plot(y)
>> grid
```

should bring up a figure displaying y(x) against x on a grid. Like many matlab routines `plot` can take a variable number of arguments. With just one argument (as here) the argument is taken as a vector of y values, the x values defaulting to 1,2,..., Note the effect of resizing the figure window.

The Teaching System is set up so that if you have a directory called `matlab` in your home directory, then *.m* scripts there will be run irrespective of which directory you were in when you started matlab.

8 Applications

8.1 Graphical solutions

MATLAB can be used to plot 1-d functions. Consider the following problem:

Find to 3 d.p. the root nearest 7.0 of the equation $4x^3 + 2x^2 - 200x - 50 = 0$

MATLAB can be used to do this by creating file `eqn.m` in your matlab directory:

function [y] = eqn(x)
% User defined polynomial function
[rows,cols] = size(x);
for index=1:cols

```
  y(index) = 4*x(index)^3+2*x(index)^2-200*x(index)-50;
end
```

The first line defines 'eqn' as a function – a script that can take arguments. The square brackets enclose the comma separated output variable(s) and the round brackets enclose the comma separated input variable(s) - so in this case there's one input and one output. The `%` in the second line means that the rest of the line is a comment. However, as the comment comes immediately after the function definition, it is displayed if you type :

```
>> help eqn
```

The function anticipates `x` being a row vector so that `size(x)` is used to find out how many rows and columns there are in `x`. You can check that the root is close to 7.0 by typing:

```
>> eqn([6.0:0.5:8.0])
```

Note that `eqn` requires an argument to run which is the vector [6.0 6.5 7.0 7.5 8.0].

The `for` loop in MATLAB should be avoided if possible as it has a large overhead. `eqn.m` can be made more compact using the `.` notation. Delete the lines in `eqn.m` and replace them with:

```
function [y] = eqn(x)
% COMPACT user defined polynomial function
y=4*x.^3+2*x.^2-200*x-50;
```

Now if you type ' `eqn([6.0:0.5:8.0])`' it should execute your compact `eqn.m` file.

Now edit and save `ploteqn.m` in your matlab directory:

```
x_est = 7.0;
delta = 0.1;
while(delta > 1.0e-4)
  x=x_est-delta:delta/10:x_est+delta;
  fplot('eqn',[min(x) max(x)]);
  grid;
```

disp('mark position of root with mouse button')
[x_est,y_est] = ginput(1)
delta = delta/10;
end

This uses the function `fplot` to plot the equation specified by function `eqn.m` between the limits specified. `ginput` with an argument of 1 returns the x- and y-coordinates of the point you have clicked on. The routine should zoom into the root with your help. To find the actual root try matlab's `solve` routine:

```
>> poly = [4 2 -200 -50];
>> format long
>> roots(poly)
>> format
```

which will print all the roots of the polynomial : $4x^3 + 2x^2 - 200x - 50 = 0$ in a 15 digit format. `format` on its own returns to the 5 digit default format.

8.2 Plotting in 2D

MATLAB can be used to plot 2-d functions e.g. $5x^2 + 3y^2$:

```
>> [x,y]=meshgrid(-1:0.1:1,-1:0.1:1);
>> z=5*x.^2+3*y.^2;
>> contour(x,y,z);
>> prism;
>> mesh(x,y,z)
>> surf(x,y,z)
>> view([10 30])
>> view([0 90])
```

The `meshgrid` function creates a 'mesh grid' of `x` and `y` values ranging from -1 to 1 in steps of 0.1. If you look at `x` and `y` you might get a better idea of how `z` (a 2D array) is created. The `mesh` function displays `z` as a wire mesh and `surf` displays it as a facetted surface. `prism` simply changes the set of colours in the contour plot. `view` changes the horizontal rotation and vertical elevation of the 3D plot. The `z` values can be processed and redisplayed

```
>> mnz=min(min(z));
>> mxz=max(max(z));
```

```
>> z=255*(z-mnz)/(mxz-mnz);
>> image(z);
>> colormap(gray);
```

`image` takes interprets a matrix as a byte image. For other *colormaps* try `help color`.

9 Advanced plotting

Consider the following problem:

Display the 2D Fourier transform intensity of a square slit.

(The 2D Fourier transform intensity is the diffraction pattern). Enter the following into square_fft.m :

```
echo on
colormap(hsv);
x=zeros(32);
x(13:20,13:20)=ones(8);
mesh(x)
pause % strike a key
y=fft2(x);
z=real(sqrt(y.^2));
mesh(z)
pause
w=fftshift(z);
surf(w)
pause
contour(log(w+1))
prism
pause
plot(w(1:32,14:16))
title('fft')
xlabel('frequency')
ylabel('modulus')
grid
echo off
```

The `echo` function displays the operation being currently executed. The program creates a 8-by-8 square on a 32x32 background and performs a 2D FFT on it. The intensity of the FFT (the real part of `y`) is stored in `z` and the D.C. term is moved to the centre in `w`. Note that the plot command when given a 3 by 32 array displays 3 curves of 32 points each.

10 Input and output

Data can be be transferred to and from MATLAB in four ways:

1. Into MATLAB by running a *.m* file
2. Loading and saving *.mat* files
3. Loading and saving data files
4. Using specialised file I/O commands in MATLAB

The first of this involves creating a *.m* file which contains matrix specifications e.g. if `mydata.m` contains:

data = [1 1;2 4;3 9;4 16;5 25;6 36];

Then typing:

```
>> mydata
```

will enter the matrix `data` into MATLAB. Plot the results (using the cursor controls, it is possible to edit previous lines):

```
>> handout_length = data(:,1);
>> boredom = data(:,2);
>> plot(handout_length,boredom);
>> plot(handout_length,boredom,'*');
>> plot(handout_length,boredom,'g.',handout_length,boredom,'ro');
```

A *.mat* file can be created by a `save` command in a MATLAB session (see below).

Data can be output into ASCII (human readable) or non-ASCII form:

```
>> save results.dat handout_length boredom -ascii
>> save banana.peel handout_length boredom
```

The first of these saves the named matrices in file `results.dat` (in your current directory) in ASCII form one after another. The second saves the matrices in file `banana.peel` in non-ASCII form with additional information such as the name of the matrices saved. Both these files can be loaded into MATLAB using `load`:

```
>> clear
>> load banana.peel -mat
>> whos
>> clear
>> load results.dat
>> results
>> apple=reshape(results,6,2)
```

The `clear` command clears all variables in MATLAB. The `mat` option in `load` interprets the file as a non-ASCII file. `reshape` allows you to change the shape of a matrix. Using `save` on its own saves the current workspace in `matlab.mat` and `load` on its own can retrieve it.

MATLAB has file I/O commands (much like those of C) which allow you to read many data file formats into it. Create file `alpha.dat` with **ABCD** as its only contents. Then:

```
>> fid=fopen('alpha.dat','r');
>> a=fread(fid,'uchar',0)+4;
>> fclose(fid);
>> fid=fopen('beta.dat','w');
>> fwrite(fid,a,'uchar');
>> fclose(fid);
>> !cat beta.dat
```

Here, `alpha.dat` is opened for reading and `a` is filled with unsigned character versions of the data in the file with 4 added on to their value. `beta.dat` is then opened for writing and the contents of `a` are written as unsigned characters to it. Finally the contents of this file are displayed by calling the `Unix` command `cat` (the '!' command escapes from matlab into unix).

```
>> help fopen
>> help fread
```

will provide more information on this MATLAB facility.

11 Common problems

If you need save data in files data1.mat data2.mat etc. use `eval` in a *.m* file e.g.

```
for x=1:5
results=x;
operation=['save ','data',num2str(x),' results']
eval(operation)
end
```

The first time this loop is run, '`save data1 results`' is written into the `operation` string. Then the `eval` command executes the string. This will save `1` in `data1.mat`. Next time round the loop `2` is saved in `data2.mat` etc.

`num2str` is useful in other contexts too. Labelling a curve on a graph can be done by:

```
>> gamma = 90.210;
>> labelstring = ['gamma = ',num2str(gamma)];
>> gtext(labelstring)
```

`num2str` converts a number into a string and `gtext` allows you to place the label where you like in the graphics window.

Saving a figure in postscript format can be done using `print -deps pic.ps`. Entering `print` on its own will print the current figure to the default printer.

12 More information

The Help System's MATLAB page[3] is a useful starting point for MATLAB information with links to various introductions and to short guides on curve-

[3] http://www-h.eng.cam.ac.uk/help/tpl/programs/matlab.html

fitting[4], symbolic maths[5], etc, as well as contact information if you need help. The handout on *Using Matlab*[6] contains information on the local installation of MATLAB.

[4]http://www-h.eng.cam.ac.uk/help/tpl/programs/Matlab/curve_fitting.html
[5]http://www-h.eng.cam.ac.uk/help/tpl/programs/Matlab/symbolic.html
[6]http://www-h.eng.cam.ac.uk/help/tpl/programs/matlab5/matlab5.html