Learning from Data 1
Introduction to Matlab

David Barber
dbarber@anc.ed.ac.uk
course page : http://anc.ed.ac.uk/∼dbarber/lfd1/lfd1.html

This is a modified version of a text written by Chris Williams.

For web based help follow the links in
http://www.ai.mit.edu/courses/6.867-f01/matlab.html.
See also http://www.math.unh.edu/∼mathadm/tutorial/software/matlab/

Background

This document has the objective of introducing you to some of the facilities available in MATLAB , including:

1. Using the interpreter and help system.
2. Plotting facilities.
3. Scripts and functions.
4. Matrices.

Section 5 gives information on the NETLAB toolbox for neural networks.

MATLAB is an interpreted language (and as such can be much slower than compiled software) for numeric computation and visualisation. It offers high level facilities for dealing directly with mathematical constructs. The particular benefits that it offers for this course are:

1. Excellent support for linear algebra and matrix operations. The basic type in MATLAB is a double precision matrix. The software was originally developed as a linear algebra package (MATLAB stands for MATrix LABoratory) and has efficient and numerically reliable algorithms for matrix inversion, eigenvalues etc.
2. Visualisation facilities. The in-built graphing and plotting functions are easy to use for both 2d and 3d plots.
3. Ease of extension. Functions and scripts can be written in the MATLAB language (in 'M-files') and these can then be called in exactly the same way as the core functionality of MATLAB . In fact, the 'toolboxes' that extend the functionality of MATLAB to more specialist areas are written in this way. NETLAB (a MATLAB toolbox for neural networks) consists of a set of M-files.
4. Portability. Software written in the MATLAB language is portable to any platform that runs MATLAB , including Unix machines, PCs and Macintoshes.

1 Using the Matlab interpreter and help system

The basic objects that MATLAB works with are *matrices*: 2-d rectangular arrays of double precision (or complex) numbers. Operations and commands in MATLAB are intended to work with matrices just as they would be written down on paper. The MATLAB interpreter can be controlled dynamically from the command line.

1. To enter a matrix, you should follow these conventions:
   - Separate entries with white space or commas.
   - Use a semi-colon `;` to denote the end of each row.
   - Surround the entries with square brackets `[` and `]`.

   The statement

   ```
   A = [8 1 6; 3 5 7; 4 9 2]
   ```

   results in the output

   ```
   A =

        8     1     6
        3     5     7
        4     9     2
   ```

   MATLAB will output the result of every command. To suppress this, you should terminate your command with a semi-colon `;`. Thus, typing

   ```
   A = [8 1 6; 3 5 7; 4 9 2];
   ```

   still sets `A` to be the same matrix, but there is no output.

   The value of any variable, such as the matrix `A`, is retained until it is explicitly modified. If there is no variable on the left hand side of an expression, then MATLAB assigns the result to a built-in variable named `ans`.

   To find out the value of the i, jth element of `A`, type `A(i, j)`. For example, on typing

   ```
   b = A(1, 2)
   ```

   the following is output

   ```
   b =

       1
   ```

   Note that matrix indices start from 1. To obtain the second row of the matrix, type

   ```
   c = A(2, :)
   ```

   which gives the following output:

   ```
   c =

        3     5     7
   ```

   To obtain the third column, type

   ```
   d = A(:, 3)
   ```

   which gives the following output:

   ```
   d =

        6
        7
        2
   ```

2. In some situations, MATLAB gives a special meaning to *vectors*, which are matrices with only one row or column. A particularly useful way of forming row vectors is using the colon operator `:`. The line

   ```
   x = 0:0.1:1;
   ```

   generates a row vector `x` of length 11 containing the values from 0 to 1 with increments of 0.1. This construction is often used to generate the x-axis points in graphs. So the command

   ```
   y = sin(2 * pi * x);
   ```

   generates regularly spaced values from a sine curve.

3. The MATLAB interpreter allows you to edit your command line. This makes it much easier to correct small typing mistakes or to repeat sequences of nearly identical commands without error. You can use emacs-style command line editing, e.g. Ctl-p gives you the previous command, and this can be repeatedly used to go back in the command history. You can then edit a command line using the usual delete/backspace key, and move around in the command line using Ctl-b (moves backwards) and Ctl-f (forwards).

   Alternatively, the up and down arrow keys (usually on the right of your keyboard) can be used to move forwards or backwards through the command history, while left and right arrow keys position the cursor on the command line. Typing carriage return anywhere on the line will then submit the command to the interpreter.

   For example, to modify the variable `x`, type up arrow twice. This will give you the same line as you typed two commands ago, namely:

   ```
   x = 0:0.1:1;
   ```

   By moving the cursor, deleting the characters between the two colons, and typing `0.05`, modify this to read

   ```
   x = 0:0.05:1;
   ```

   and type carriage return to submit the command. Now the vector `x` has length 21 and contains the values from 0 to 1 with increments of 0.05. However, `y` still has length 11, and its values no longer correspond correctly with those in `x`. (Unlike with a spreadsheet, we cannot link variables in MATLAB so that they are automatically updated.) To rectify this, the simplest solution is to repeat the definition of `y` by typing up arrow twice and submitting the command.

4. To save data, you can use the command `save`. For example, to save the variable `A` in ASCII format into a file `A.dat`, type:

   ```
   save A.dat A -ascii
   ```

   Type `ls` to get a listing of the files in your current directory, which should include `A.dat`. (Note that normally one can simply use `save A.dat A` which stores the data in an efficient format. The data can be reloaded using `load A.dat`)

5. Of course, most of the time the datasets we will be analysing are far too large to type at the command line without error. Instead, we can read in the matrices from external files. These files can be in a MATLAB format, or stored as text, with each row of the matrix on a new line. To read in some data which is stored in MATLAB format, type (assuming that you have a MATLAB file data.mat in your current working directory)

```
load data
```

You can find out what variables you currently have by typing:

```
whos
```

6. MATLAB is a very powerful tool with hundreds of built-in functions. Even the most hardened MATLAB veteran may occasionally forget the name of a function or the order of its arguments. Fortunately, there is an on-line help facility that provides information on most topics. To get help on a specific function, you need only type `help` followed by the function name. So, to find out about the function `mean`, you should type:

   ```
   help mean
   ```

   A more general information search is provided by `lookfor` (which is similar to the Unix `apropos` command). This enables you to find out what functions MATLAB provides for certain operations. For example, to find out what functions there are related to calculating covariances, type:

   ```
   lookfor covariance
   ```

2 Simple use of two dimensional plotting facilities

1. A dataset for a toy regression problem (the noisy `sin`) can very easily be plotted using the vector `x` we created earlier. The simplest form of the `plot` function takes the x vector as its first argument, and the y vector as its second argument. Different line styles (or scatter plots) can be defined with a third argument. (For more detail, just type `help plot`.) To see the data, type

   ```
   plot(x, sin(2*pi*x) + 0.1*randn(size(x)), '+')
   ```

   which will give you a scatter plot of the data with yellow plusses for each point. The function `randn` generates samples from a standard normal distribution (i.e. a Gaussian with zero mean and unit variance).

   Notice that the function `sin` can take a vector (or indeed a matrix) as input, and return the sine of each element.

   Say we also wanted to plot another curve on the same axes, e.g. $\cos 2\pi x$. We need two new ideas: (i) the use of `hold on`, which permits multiple plots to be made on the same axes and (ii) the choice of different colours for different plots. This can be achieved by a third argument to the `plot` function (for more detail, just type `help plot`).

   ```
   plot(x, sin(2*pi*x))
   hold on
   plot(x, cos(2*pi*x), 'g')
   hold off
   ```

   The `'g'` option plots the cosine curve in green. `hold off` cancels the `hold on` command.

   It is possible to add extra information to the plot using the `title`, `xlabel`, `ylabel`, and `text` commands, and to control the scaling of the axes using the `axis` command. Note that these commands must *follow* the `plot` to which they refer. Axes with non-linear scaling can be obtained with the `loglog`, `semilogx` and `semilogy` commands.

A plot can easily be saved in a variety of formats. For example, to save a plot as an encapsulated postscript file called `myplot.eps`, type

```
print -deps myplot.eps
```

2. We can also easily make histogram plots. First, create some data by typing

   ```
   z = randn(500,1)
   ```

   This will create a vector of 500 (pseudo)random samples from a zero-mean unit-variance Gaussian distribution. We will now plot a histogram of this data, specifying 15 bins by typing

   ```
   hist(z, 15)
   ```

   Does it look how you expect it should?

3 Elementary use of scripts and functions.

In some exercises you will need to edit existing scripts and functions, either to test the effect of varying parameters, or to make it easier to run a sequence of instructions multiple times (in case of typing errors). This exercise is designed to give you some practice at this.

Scripts automate a fixed sequence of instructions. Functions are more flexible, in that they can take arguments and return values, which allows the user to abstract away from particular variable names. Both functions and scripts are text files (called M-files) which have the name `foo.m`, where `foo` is the script or function name. Functions and scripts are invoked in MATLAB by typing `name` with the relevant arguments enclosed in round brackets.

1. Copy the file `http://anc.ed.ac.uk/~dbarber/lfd1/whiten.m` to an appropriate directory.
2. Start up a text editor.
3. Open the file `whiten.m`. The purpose of this function is to preprocess data by applying a linear transformation so that all the variables are zero mean and unit variance. This file assumes that the data is in the form where each column of the matrix represents a datapoint. This is the most natural mathematical way to represent a set of datapoints. (Unfortunately, the usual MATLAB convention is that each column represents a variable and each row represents an observation! We are therefore going against MATLAB convention, but I think towards a more consistent convention for our course). The first line of the file

   ```
   function [y,mu,s] = whiten(x)
   ```

   defines the function interface. The argument `x` is the input data matrix, while the return value `y` is the processed data (and therefore has the same dimensions as `x`). The variables `x`, `y` and all other variables in this function are *local* to this function and are quite distinct from the variables we have already defined in this session.
4. Note that the `%` symbol indicates that the rest of the line is a comment and will be ignored by the MATLAB interpreter. The line

   ```
   n = size(x, 2);
   ```

   finds the number of columns in the data matrix. After this, the lines

```
mu = mean(x')';
s  = std(x')';
```

calculate the mean and standard deviation of each column of the data (the transposes are due to MATLAB 's convention of a datapoint being a row). The next step is to subtract the mean of each column. While it would be possible to do this using MATLAB 's `for` loops, it would be very slow. It is much more efficient to use matrix operations. What is required is to subtract a matrix where every entry in the ith row is the mean of that row. This is done by the lines:

```
e = ones(1,n);
y = x - mu*e;
```

Now `y` is a data matrix where each variable has zero mean. We just need to divide each column by the standard deviation. To do this, we use the element-wise division operator `./` (the usual division operator `a/b` essentially multiplies `a` by the matrix inverse of `b`).

```
y = y ./ (s*e);
```

5. You are now ready to run the function. Switch back to the MATLAB command window. First, we can generate some random data:

   ```
   foo = rand(2,10);
   ```

   The function `rand` generates samples from a random variable which is

   uniformly distributed on the interval $[0, 1]$. This dataset has 2 rows and 10 columns and represents a set of 10 datapoints, each of which is two-dimensional. Type

   ```
   m = mean(foo')'
   s = std(foo')'
   ```

   to find out the mean and standard deviation of each variable. Now we can normalise the data with our function. Type

   ```
   bar = whiten(foo);
   ```

   and find out the new mean and standard deviation. To within rounding error, they should be zero and one respectively.

4 Matrices

4.1 Matrix addition and multiplication

Enter a matrix by typing

```
A = [8 1 6; 3 5 7; 4 9 2]
```

This results in the output

```
A =

     8     1     6
     3     5     7
     4     9     2
```

We can also enter matrices `B` and `C` by typing `B = [1 4 7; 2 5 8; 3 6 0]` and
`C = [1 2 3; 4 5 6]`. Does the statement `D = A + B` do what you would expect? How about `E = A + C`?

To transpose a matrix in MATLAB we use the `'` operator (a single backquote). For example, try `F = A'` .

Suppose we define a column vector by `g = [-1 0 2]'`. Then we multiply this vector by a matrix by typing `h = A*g`. Suppose we create a row vector, e.g. `u = g'`. What happens if we then try `v = A*u` ? Why ?

MATLAB will, of course, also multiply matrices of the appropriate size. For example
`A2 = A*B` is valid, as is `A3 = C*A`. What about `A4 = A*C` ?

4.2 Solving systems of linear equations

Suppose we want to solve the system of linear equations represented by

$$A\mathbf{x} = \mathbf{g}.$$

MATLAB provides a special division operator for this purpose; we can simply write

```
x = A \ g
```

and the solution,

```
x =
   -0.0194
    0.2722
   -0.1861
```

is printed out. You can check that `x` is the solution to the system of linear equations by calculating `A*x`.

4.3 Manipulating the elements of a matrix

It is possible to cut parts out of matrices and perform quite complex operations cutting-and-pasting matrix entries in MATLAB . We will consider some simple examples. What is the effect of

```
A
A(2:3,1:2)
```

Note the syntax above: `2:3` specifies the rows wanted, and `1:2` specifies the desired columns. Rows and columns can be manipulated as complete entities. Hence `A(1,:)` gives the first row of `A`, and `A(:,2)` gives the second column.

4.4 Special matrices

Some special matrices that MATLAB knows about are `ones(m,n)`, `zeros(m,n)`, `rand(m,n)`, `randn(m,n)` and `eye(n)`. `ones(m,n)` generates a matrix of size $m \times n$ with all the elements equal to 1. Try typing `ones(4,2)`. `zeros(m,n)` acts similarly but fills all of the elements with 0. This can be useful for initializing a matrix before computations. Try `zeros(3,3)`.

`rand(m,n)`, `randn(m,n)` generate random matrices of size $m \times n$. `rand` choses (pseudo)random numbers in the interval $(0.0, 1.0)$, while `randn` generates them according to a Normal distribution with zero mean and unit variance. Try `randn(5,3)`.

`eye(n)` generates a $n \times n$ identity matrix. Type `eye(5)` to check this.

4.5 Element-by-element operations

Sometimes we don't want to do matrix operations, but elementwise operations instead. These are achieved by using a dot `.` to precede the operator. For example, say we have the vectors

```
y = [1 2 3];    z = [4 5 6];
```

What is the effect of

```
y .* z
y ./ z
y .^2
```

These "dot" operations can also be performed on matrices; try `B./A`

4.6 Output formatting

The name and value of a variable can be output just by leaving the semicolon off the end of the line. However, it is possible to produce tidier output by using the function `disp`. For example

```
disp('the values in matrix A are');
disp(A);
```

It is also possible to use C-like syntax with the command `fprintf`. The `format` command specifies the output format. For example type

```
pi
format long
pi
```

5 Netlab neural network software

The Netlab neural network software is a toolbox for Matlab, written by Ian Nabney and Christopher Bishop.

It consists of a library of Matlab functions and scripts based on the approach and techniques described in Neural Networks for Pattern Recognition by Christopher M. Bishop, (Oxford University Press, 1995). The Netlab homepage can be found at

`http://www.ncrg.aston.ac.uk/netlab/index.html`

The software (should have been!) installed on the School machines, so that upon typing the command `matlab` from the command line the netlab functions are ready for your use. If an error occurs, probably the routines have not been installed. Hoever, you can do this simply by downloading the routines from the NETLAB homepage and placing them in your working directory (or in any of your directories and adjusting the MATLAB path so that MATLAB can find the routines). The command `help netlab` issued within MATLAB will give a listing of the various functions and demos available, and the command `demnlab` will bring up a GUI interface to the demos.