Intermediate MATLAB for Beginners

Jas Ellis, London School of Economics
j.j.ellis@lse.ac.uk

Version 0.43, September 12, 2006
This file is maintained at `http://personal.lse.ac.uk/ELLISJ/`; comments and corrections are gratefully recieved. I'm using MATLAB 7.0.

1 First Things

1.1 Help

The MATLAB help is pretty good. Either use the help browser (accessed from the /Help menu) or use:

```
>> help <command name>
```

The help browser can also be accessed using `doc <command name>`

1.2 Halting MATLAB

`CONTROL-C` will cause MATLAB to stop calculating and return to the command prompt.

2 Commands, Scripts and Functions

2.1 The Command Prompt

You can execute any MATLAB command at the command prompt (>>). These commands execute in the *MATLAB Workspace*. That is, the variables you refer to in your command exist (or will be created) in the workspace. You can see them in the *Workspace* panel, accessed from the /Window menu.

```
>> X = [1 2 3; 4 5 6]
```

Matrices and vectors are entered row-wise using `;`s to break rows, so this creates a 2x3 matrix `X`. To suppress the output of your calculation, place a semicolon after it. MATLAB is case-sensitive by default.

```
>> y = [1 2 3];
>> y = y';          % Transpose
>> Z = X*y;         % Matrix multiplication
```

This creates a row-vector `y`, redefines `y` to be its transpose (a column-vector), and defines `Z` by matrix multiplication. An apostrophe (`'`) denotes the transpose. Equivalently:

```
>> y = [1; 2; 3];
>> Z = X*y;
```

Here, `y` is defined as a column-vector to begin with.

To concatenate vectors or matrices, use the square brackets:

```
>> [y y]
ans =
     1     1
     2     2
     3     3
```

If you wish to make several calculations in one go, use `SHIFT-RETURN` at the end of the line. The commands will all be executed when you press `RETURN`. This is essentially the same as a *script*. Commands like `for` will wait for a matching `end`, so using `SHIFT-RETURN` is unnecessary in this case.

To execute a command you have used before, you can either double-click it in the *Command History* window/panel, or press the `UP-ARROW`. Typing a letter or so and then pressing the `UP-ARROW` gives the last command starting with those characters. After typing something, pressing `TAB` will suggest commands and filenames that begin with what you typed.

2.2 Scripts

A *script* is a text file (called an *m-file*) containing a sequence of MATLAB commands. Scripts also execute within the MATLAB Workspace, so a script may overwrite or clear variables you are using. This is their major disadvantage. They are useful when one wishes to perform subsequent calculations in the workspace. You can edit scripts and functions in the MATLAB Editor.

2.3 Functions

A *function* is also an m-file containing a sequence of commands, but differs from a script in two ways:

1. A function has its own workspace. If you have an `x` in the MATLAB workspace, you can use `x` as a variable in a function without overwriting it.
2. Functions take inputs (called ‘arguments’) and return outputs.

```
function [a b] = demo(c)
a = mean(c);
b = std(c);
end
```

`demo` takes a matrix and returns the means and standard deviations of its columns. Most MATLAB commands work with matrices and higher-dimensional arrays. More on this later. Call `demo` from the command prompt like this:

```
>> [mu sigma] = demo(X)
mu =
 2.5000 3.5000 4.5000
sigma =
 2.1213 2.1213 2.1213
```

If you just want the first output:

```
>> mu = demo(X)
mu =
 2.5000 3.5000 4.5000
```

3 Some useful commands

`clear`	Clears the workspace.
`clear <variablename>`	Clears a single variable or list of variables.
`clc`	Clears the command window.
`whos`	Lists all variables, their sizes and their types.
`save <filename>`	Saves all variables in `filename.mat`.
`load <filename>`	
`tic`	Starts a timer
`toc`	Displays the time taken since `tic` was last called. `t = toc` stores the time taken.
`disp`	Displays things on the screen.

Eg.

```
>> save demodata X mu
```

4 Calculating Things

4.1 Scalar Functions of Vectors

Remember: MATLAB operates on matrices. Scalar functions of vectors — eg. `mean(X)` — will operate on the columns of `X`. But they will work 'normally' on row-vectors.

```
>> y = [1 2 3];
>> X = [1 2 3; 4 5 6];
>> max(y)
ans =
     3
>> max(y')
ans =
     3
>> max(X)
ans =
     4     5     6
>> max(X')
ans =
     3     6
```

NB. The result of a displayed but unassigned calculation will be stored in a variable called `ans` in the workspace.

Commands to try:

```
max(X)  min(X)   mean(X)  median(X)
sum(X)  prod(X)  std(X)
```

4.2 Scalar Functions *or* Vector Functions of Vectors

Conversely, many MATLAB commands return an array of the same size as the original one:

`abs(X)` `sqrt(X)` `exp(X)` `sin(X)` `cos(X)`

`rand(X)`	Uniform random variable.
`randn(X)`	Normal random variable.
`ones(X)`	A matrix of ones.
`zeros(X)`	A matrix of zeros.

The last four can also take integers as arguments, specifying the size in each dimension:

```
>> ones(2, 3)
ans =
     1     1     1
     1     1     1
```

NB. `rand`, `randn`, `ones`, `zeros` will take a row-vector to be a list of dimensions! They will *not* return a row-vector in this case.

4.3 More functions

`size(X)`	Returns a row-vector giving the size of each dimension of `X`.
`length(X)`	Returns the number of columns of `X`. Or, if `X` is a column-vector, its height.
`eye(n)`	Returns the `n` x `n` identity matrix.

To construct a sequence of numbers in a vector, use the colon (`:`) operator.

```
>> a = 1:10         % Sequence from 1 to 10. (Default step is 1).
a =
     1     2     3     4     5     6     7     8     9    10
>> b = 0:0.2:0.8    % Sequence from 0 to 0.8, step 0.2
b =
         0    0.2000    0.4000    0.6000    0.8000
```

The percent sign is used for comments. Anything on a line following a `%` is ignored by MATLAB. Use comments to explain your code to others (and yourself, later).

4.4 Basic Calculations

`+`, `-`, `*`, `/` and \ all work as expected... on matrices. `^` will work on any square matrix. Scalar multiplication works automatically. Adding a scalar to a matrix *will* work, but adds to every element of the matrix.

```
>> A = [1 2; 3 4];
>> B = A^2
B =
     7    10
    15    22
>> y = [9; 10];
>> x = B\y;         % Division
>> B*x
ans =
     9
    10
```

(This demonstrates to solve linear systems of simultaneous equations). To perform many calculations at once, it is often useful to 'vectorise'. Use a `.` (full stop) to perform an element-wise multiplication, division or exponentiation.

```
>> A .* B
ans =
     7    20
    45    88
```

4.5 Some Constants

`pi`	*How I need a drink alcoholic of course after studying the heavy chapters of quantum mechanics.* (NB. MATLAB probably knows more digits than I do).
`i`	Square root of minus one.
`true`	'one' — stored as a 'logical' (More later).
`false`	'zero' — ditto
`NaN`	'Not a Number'. Any calculation involving `NaN`s will results in a `NaN`.
`Inf`	Positive infinity.

These can be use anywhere in MATLAB. It is possible — though not advised — to give these names to your own variables. Once you clear your variable, you can use the constant again.

```
>> exp(inf)
ans =
   Inf
>> exp(-inf)
ans =
     0
>> sin(inf)
ans =
   NaN
```

4.6 Logical Calculations

Given any two scalars, they may be compared with the `==`, `~=`, <, >, <`=` and >`=` operators. This gives either a `true` or `false` logical result.

```
>> x = 1 >= 2
x =
     0
>> whos x
  Name      Size                    Bytes  Class
  x         1x1                         1  logical array
Grand total is 1 element using 1 bytes
```

A 'logical array' can contain only 0 or 1: 0 is `false` and 1 is `true`.

NB the *test for equality* is `==`. A single equals sign is used for *assignment.*

More complicated logical expressions can be constructed using `&` (*and*), | (*or*) and `~`(*not*). For more, see `help relop`.

```
>> y = true
y =
     1
>> ~x & y
ans =
     1
```

Brackets can be used for more complicated terms.

Scalar reals can also be evaluated as logicals. Nonzero values are `true`, zero is `false`. If a complex number is used, only the real part is used. Conversely, logicals can be used as scalars in the obvious way. The `isnan` and `isinf` commands are useful; notice:

```
>> nan==nan
ans =
     0
```

4.7 Logical Matrices

Usually, for flow control etc., scalar logicals will be required. However, MATLAB also allows logical matrices. The logical operators work component-wise:

```
>> I = eye(3);          % 3x3 identity matrix
>> X = [1 2 3; 4 5 4; 3 2 1];
>> X == I
ans =
     1     0     0
     0     0     0
     0     0     1
```

This highlights a very important point. If `A` and `B` are not scalars, the term `A == B` is not a scalar, and should *not* be used to test the equality of `A` and `B`. Use `isequal(A, B)`. (`isequal` or `strcmp` must be used on strings).

Two commands that can be used on logical arrays are `any` and `all`. They operate column-wise on matrices and row-wise on row-vectors. Eg. `any(all(X))` returns `true` if there exists a column containing only `true`s.

5 Flow Control

5.1 The `if` Command

For controlling what a function or script does, use `if`.

```
if isequal(A, eye(2)), disp('A is the 2x2 identity'); end
```

NB. The comma may be omitted, but improves legibility when the command is given on a single line.

The `if` command is followed by a logical scalar, the commands to be executed, and an `end`. See the MATLAB help for more.

5.2 The `for` Command

For repeating things, use `for`.

```
>> for a = 1:3          % a runs from 1 to 3
disp(a^2);          % Display a squared
end;
     1
     4
     9
```

The syntax is:

FOR variable = expr, statement, ..., statement END

The loop *variable* (`a` in my example) takes the first element (or the first column, if *expr* is a matrix) and executes the statements with the loop using that value. When *END* is reached, this is repeated with the second value, and so on.

The `continue` command tells MATLAB to jump to the next run of the loop. `break` quits the loop.

```
>> for a = 1:10
if a/2 == floor(a/2)       % If a is divisible by 2...
continue;                  % Jump to the next iteration
end;
disp(a);
if a == 7 break; end;      % Quit the loop if a is seven
end;
     1
     3
     5
     7
```

The `while` command also works in the usual way.

6 Deconstructing Matrices

6.1 Rows and Columns

The elements of a matrix can be accessed using parenthesis (rows, columns):

```
>> X = [1 2 3; 4 5 6];
>> X(1, 3)       % Element in the first row, third column of X
ans =
     3
```

The colon operator can be used to specify several elements:

```
>> X(1, 2:3)         % Elements in the 1st row, columns 2 and 3
ans =
     2     3
```

The colon operator is also used also to refer to either a row or column of a matrix.

```
>> X(1,:)         % The first row of X
ans =
     1     2     3
>> X(:,2)         % The second column of X
ans =
     2
     5
```

These methods can also be used to assign values to elements:

```
>> X(1,2:3) = 20
X =
     1    20    20
     4     5     6
```

Elements outside the existing array will be created:

```
>> X(1,5) = 9
X =
     1    20    20     0     9
     4     5     6     0     0
```

It should be noted that resizing arrays can be particularly time-consuming. It is much better to define a full-size array beforehand (eg. using `zeros(n,m)`) than to resize it as your computation runs.

Higher-dimensional arrays are dealt with similarly. These can be easily created using this last method:

```
>> X(:,:,2) = X.^2
X(:,:,1) =
     1    20    20     0     9
     4     5     6     0     0
X(:,:,2) =
     1   400   400     0    81
    16    25    36     0     0
```

`X(:,:,2)` refers to all rows and columns in the 2nd 'layer' of `X`.

Note: column-wise commands still operate on the columns of multidimensional arrays.

```
>> sum(X)
ans(:,:,1) =
     5    25    26     0     9
ans(:,:,2) =
    17   425   436     0    81
```

Arrays *can* have dimensions of size zero.

6.2 Tips for Using Multidimensional Arrays:

`squeeze(X)`	Eliminates singleton dimensions.
`permute(X,a)`	Permutes the dimensions of `X` according to a vector of integers, `a`.

```
>> X = permute(X, [3 2 1])
X(:,:,1) =
 1 20 20 0 9
 1 400 400 0 81
```

```
X(:,:,2) =
 4 5 6 0 0
 16 25 36 0 0
```

6.3 Linear and Logical Indexing

Sometimes it is useful to treat an array simply as a list of elements. One might want to perform a task on every element, for example. This can be done by referring to `X` with just one integer, and similarly for sequences.

```
>> X(5)         %5th element of X
ans =
 20
>> X(5:10)         %5th through 10th elements of X
ans =
 20 400 0 0 9 81
```

`numel(X)` gives the number of elements of `X`. This will help building generalised sections of code. For example:

```
>> for n=1:numel(X)
a = X(n) - 6;                % a is X(n) minus 6
if a/10 ~= floor(a/10)       % If a is not divisible by 10...
continue;                    % ...jump to the next iteration
end;
disp(X(n));                  % Display X(n)
end
 16
 6
 36
```

Logical indexing is a related concept which is even more useful. If `X` is an array, a logical array, `A`, can be used to refer to elements of `X` (non-logical arrays will be treated as logical arrays - see the previous section on this). `A` must be the same size as `X`.

The last example can be also be implemented this way:

```
>> a = X-6;                  % a's elements are X-6
>> A = a/10 == floor(a/10)   % Test for multiples of 10
A(:,:,1) =
 0 0 0 0 0
 0 0 0 0 0
A(:,:,2) =
 0 0 1 0 0
 1 0 1 0 0
>> X(A)                       % Elements of X according to A
ans =
 16
 6
 36
```

Logical indexing can be used to assign elements too:

```
>> X(A) = inf
X(:,:,1) =
 1 20 20 0 9
 1 400 400 0 81
X(:,:,2) =
 4 5 Inf 0 0
 Inf 25 Inf 0 0
```

7 Using the Editor

The editor works like any standard text editor, but it is also integrated with the MATLAB engine. The following tips will speed up the writing and testing of MATLAB scripts and functions.

1. `F5`
   Saves and runs your current m-file. It is equivalent to typing the name of the file at the command prompt, without giving any arguments.

2. `F12`
   Sets (or clears) a breakpoint in the code. MATLAB will halt when it reaches a breakpoint. You can then use the command prompt to check and edit variables. This is 'debug mode'. If a command in your program causes an error and you can't work out why, set a breakpoint before it. Check that all your variables are as you expected. Press `F5` or `F10` to continue

3. `F10`
   Steps through a single line of code. See `F12`, above.

4. `...`
   For long lines of code, use an ellipsis to tell MATLAB to continue on the next line. Anything after the elipsis is regarded as a comment.

5. *Hovering over a variable name*
   If you are in debug mode, the contents of a variable in the workspace will pop up if you hover over a variable name.

6. `TAB`
   Use the `TAB` key to indent your code. Code that exists between an if or for and the corresponding end should be indented. This will help both you and others understand your code.

7. *Default parameters*

Consider a function that requires an input argument (or several). Running at the command prompt without arguments (or pressing F5) will give an error. To implement default values, or simply to make testing quicker, add the following:

```
if ~exist('x', 'var')   % If the variable 'x' does not exist...
 x = 15; end;           % ...set the x to be 15
```

8 Graphics Example

```
>> a = [0:0.1:2*pi];   % Points between 0 and 2*pi, spaced at 0.1
>> b = sin(a);         % Sines of the above
>> plot(b)             % Plot b against the index of b (1 to 63)
>> plot(a,b)           % Plot b against a
>> c = cos(a);
>> B = [b;c];          % First row is b, second row is c
>> plot(a,B)           % Plots the two series on the same graph
>> xlabel('a')
>> title('The Sine and Cosine')
>> legend('Sin(a)', 'Cos(a)')
>> axis tight          % Fits the axes tightly round the graph
```

If you have a function in the workspace, you can create various plots from the workspace panel in MATLAB.

9 Function Concepts

9.1 Function Handles

A function handle is a reference to a function. They are created using the @ sign.

```
>> x = @sqrt;
>> whos x
 Name Size Bytes Class
 x 1x1 16 function_handle array
Grand total is 1 element using 16 bytes
```

x is now an alias for the square root function:

```
>> x(4)
ans =
 2
```

This seems a bit pointless! (The ‘pointer’ pun was not originally intended). However, this allows us to use a function as an argument in another function. This will prove to be very useful for various things.

For example, suppose you have two functions with different versions of an algorithm. You have another function that uses one of the algorithms:

```
function y = sumfunc(x)    % Does exactly the same as 'sum'
y = 0;
for n=1:numel(x)
 y = y + x(n);
end;
end
```

And:

```
function t = triangular(n, alg)\qquad
% Calculates a triangular number when 'n' is an integer
% and 'alg' is a function that sums vectors.
t = alg([1:n]);
end
```

Create function handles to your two algorithms, and simply tell `triangular` which one to use:

```
>> s1 = @sum;
>> s2 = @sumfunc;
>> triangular(5, s1)
ans =
 15
>> triangular(5, s2)
ans =
 15
```

This *still* might seem a waste of time, but you might have two algorithms that are efficient for different types of problem. Then you could use the best one without having to re-write your implementing function. Also, lots of options have been opened up:

```
>> p = @prod;
>> triangular(5, p)
ans =
 120
>> c = @cos;
>> triangular(5, c)
ans =
 0.5403 -0.4161 -0.9900 -0.6536 0.2837
```

9.2 Anonymous Functions

The second aspect to function handles is the 'anonymous function'. For simple functions, it isn't worth writing an m-file. Anonymous functions exist only as function handles, but they can be used like any other function.

```
>> a1 = @(x) 1 - 4*x + 2*x.^2;        % A quadratic polynomial
>> a1(0)
ans =
 1
>> a1(0:0.1:0.5)
ans =
 1.0000 0.6200 0.2800 -0.0200 -0.2800 -0.5000
>> a2 = @(x,y,alpha) x.^2 + x .* y + y.^2 + x + alpha*y;
```

`a2` is a quadratic in `x` and `y` with parameter `alpha`.

```
>> a2(1,1,14)
ans =
 18
>> a2([1 2 3], [4 5 6], 14)
ans =
 78 111 150
>> ss = @(x) sum(x.^2);               % Sum of squares
>> ss([1 2 3])
ans =
 14
```

9.3 Function Solving and Minimisation

MATLAB has a range of functions that will solve nonlinear systems and minimise functions. Several are presented here. Function handles will be essential here.

```
>> [x a1x] = fminsearch(a1, 0)
x =
 1.0000
a1x =
 -1
```

Here, we minimises `a1`, with 0 as the starting point. `fminsearch` returns the minimising value and the function evaluated at that value. The results of the search algorithm can also be obtained. Check the MATLAB help for details. I'm using anonymous functions I defined earlier, but these methods will also work with function handles that point to m-files. To find the root of the polynomial `a1`:

```
>> fsolve(a1, 0)
Optimization terminated: first-order optimality is less than options.TolFun.
ans =
 0.2929
>> fsolve(a1, 2)
Optimization terminated: first-order optimality is less than options.TolFun.
ans =
 1.7071
```

This brings us to the other big use for anonymous functions: To serve as a 'wrapper' for other functions. `fminsearch` and `fsolve` only give functions a single argument - the minimising vector. Many functions take several arguments, like `a2`, which I defined earlier. To recap:

```
>> a2
a2 =
 @(x,y,alpha) x.^2 + x .* y + y.^2 + x + alpha*y
```

Suppose we want to find the minimum of `a2` with `alpha` equal to 14. Use the following:

```
>> a2wrapper = @(X) a2(X(1),X(2),14);
```

`a2wrapper` is a function handle pointing to `a2`. It takes just one argument, `X`. The first element of `X` becomes the first argument of `a2`. Similarly for the second argument. The third argument is 14. We can now give this to `fminsearch`.

```
>> fminsearch(a2wrapper,[0 0])
ans =
 4.0000 -9.0000
```

To consider a different value for `alpha`, redefine `a2wrapper`:

```
>> a2wrapper = @(X) a2(X(1),X(2),19);
>> fminsearch(a2wrapper,[0 0])
ans =
 5.6667 -12.3333
```

10 Structures

A structure is a MATLAB object that contains other objects. This can be useful when one wants to keep variables together, eg. sets of parameters. A full stop separates the name of the structure from the name of the element in the structure:

```
>> pianist1.firstname = 'michael'
pianist1 =
 firstname: 'michael'
>> pianist1.lastname = 'garrick';
>> pianist1.instrument = 'piano';
>> pianist1
pianist1 =
 firstname: 'michael'
 lastname: 'garrick'
 instrument: 'piano'
>> bio = @(x) disp([ x.firstname ' '...
x.lastname ' plays the ' x.instrument]);
```

```
>> bio(pianist1)
michael garrick plays the piano
>> bio(pianist2)
mccoy tyner plays the piano
```

11 Strings

A string is an array of characters. When defining a string, use single quotes.

```
>> alph = 'the quick brown fox jumped swiftly over the lazy dog';
>> whos alph
 Name Size Bytes Class
 alph 1x52 104 char array
```

Think of a string as a row-vector. To join strings, use the concatenation operator, `[]`:

```
>> a = 'how long';
>> b = [a ' is a piece of string?']
b =
how long is a piece of string?
```

Numbers can be converted to strings using `num2str`:

```
>> bl = ['''' b ''' is ' num2str(length(b)) ' characters long.']
bl =
'how long is a piece of string?' is 30 characters long.
```

Use two consecutive `'`s to produce a single ' within a string. Thus, `''''` is a string containing just one '. `'''` gives a ' at the start of a string. To compare strings, use `isequal`, *not* `=`.

Character arrays can have multiple columns, or more than 2 dimensions. Remember however, that each column must have the same number of elements, so strings must then be padded with spaces to fit. It is better to use cell arrays for this.

12 Cell Arrays

Cell arrays, like structures, can contain any type of MATLAB object. The difference is - perversely - that cell arrays have a structure. Instead of labelling the elements inside, cell arrays are organised like a matrix. Use curly braces ({}) to define and refer to elements in cell arrays.

```
>> cells{1} = bl;
>> cells{2} = eye(2);
>> cells
cells =
 [1x55 char] [2x2 double]
>> cells{2,1} = rand(2,2);
>> cells
cells =
 [1x55 char ] [2x2 double]
 [2x2 double] []
```

Cell arrays are particularly useful for storing strings.

```
>> numbers = {'one' 'two' 'three' 'four'};
>> for i = 1:length(numbers)
% For each element in 'numbers'
save(numbers{i}, 'i');
% Save the variable 'i' in a file called eg. "one.mat"
end;
```

13 The eval command

`eval` is the most powerful command in MATLAB - it is used to execute other commands. `eval` takes a single string as an argument and executes the string as a MATLAB command.

```
>> x = 'prod([1 3 5])';
>> eval(x)
ans =
 15
```

It is possible to assign things within `eval`, but apparently this is not recommended. On the other hand, I've done it without problems. There may be times when it can't be avoided.

```
>> y = eval(x)                      % RECOMMENDED
y =
 15
>> x2 = 'z = prod([1 3 5]);';       % NOT RECOMMENDED
>> eval(x2)                         % NOT RECOMMENDED
```

Why use `eval`? Well, often you won't need to. But it can make complicated operations much easier sometimes. It's a very useful command to know. I'll just give two examples.

```
>> vars = who;  % Cell array containing all variable *names*
>> for n = 1:length(vars)
% For each variable...
v = vars{n};
% ...v contains the name of the variable as a string...
eval([v '2 = ' v ';']);
% Equivalent to typing "<contents of v>2 = <contents of v>;"
end
```

For every variable in the workspace - eg. `x` - a variable `x2` is created, equal to `x`. The next example computes means:

```
>> for n = 1:length(vars)
v = vars{n};
eval([v '_mu = mean(' v ');']);       % Eg. 'x_mu = mean(x);'
end
```