Matlab for assignments and projects – a brief guide.

R. W. Moss

Project students wanting guidance on more advanced topics should see also:

(1) Matlab Exercises (RWM) – a detailed course.

(2) Matlab Reference Guide (RWM) – similar to this but a bit more detailed; designed for when you want to quickly look up some details.

 (3) Matlab for Scientists and Engineers, B.D.Hahn (library, e-book & paper)

0. "Crib sheet" – things to remember.

Brackets:

`[ ]` for assembling matrices and function **output** lists.

`{ }` generates a "cell array"

`( )` for everything else i.e. array indexing, BIDMAS and function **input** lists

Symbols:

`:` to construct a uniformly spaced vector or extract all elements of array e.g. `A(:, 3)`

`* / ^` perform <u>matrix</u> multiplication etc

      (for "ordinary" multiplication, use `.*` `./` `.^`)

`'` transpose (converts rows into columns).

`=` for assignment, `==` to test equivalence.

Matrix multiplication

`C = A*B` requires `B` to have as many rows as `A` has columns.

`C(m,n)` is obtained by multiplying pairs of elements from row `m` of `A` and column `n` of `B`, then summing the products.
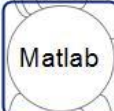
Block statements (`for`, `if`, `while`): all finish with **`end`**.

Use `all` or `any` to clarify meaning when using `if` on matrices.

Examples of valid syntax

```
A = [1 3 7; 2 4 6];
x = A(1, 2);  % first row, second column (=3)
[a, b] = size(A);  % a is number of rows
b = sin(0.1*pi);
f = (12.1 + c)*d;
g = 1:20;
h = g * g';   % h will be a 1*1 matrix
k = g' * g;  % k will be a 20*20 matrix
p = g .* (g+1);  % p will be a 1*20 matrix
for i = 1:10,
     a(i) = g(2*i-1);
end
ad = a(end) – a(1);  % "end" is the final index
if any(any(A>10)), error('Some elements too large'); end
if max(B)==100, disp('B has reached 100'), end
help num2str      % displays "help" for num2str
doc matlab        % opens documentation for Matlab
```

(nb: this is just a collection of valid lines – it does not do anything useful).

Matlab

Data types
- Numeric
- String
- Objects

Arithmetic operations
- function handles
- Complex numbers
- Matrix operations .* ./ \

Array types
- N-dimensional number arrays
- Character arrays
- Cell arrays
- Structures

Plotting
- 3D graphs
  - 2D & 3D graphs
  - Images
  - Surfaces
- Handle graphics
  - Control of graph details
  - Programs with menus and buttons

Root object
- Figures
  - Axes
    - Lines
    - Images
    - Surfaces
    - Lights
    - Text
    - Axes labels
  - Uicontrols
  - Uimenus

Data input & output
- load binary .MAT files
- load ascii data file
- script files
- low level file i/o

parsing
- workspace variables
- current folder m-files
- toolbox and path folders
- built-in functions

Programming language

Syntax
- +-*/^
- [ ], ( ), { }, ';'
- <, <=, ==, >=, >

Flow control
- for loops
- while loops
- if...elseif...else...end
- switch...case...otherwise...end
- try...catch...end

Function calls
- (input, arguments)
- [output, arguments]
- nargin, nargout, varargin

Graphics
- High level commands
- Low level commands
- get, set object properties

MBG-3

Contents

1. Creating matrices; arithmetic operations.

Examples of inputing numbers:

```
a = 1+2
b = 3 * 5 / 2
c = 6.02E23 * 1.6E-19
d = 2 * pi
```

Note that `pi` is a built-in constant that can be over-written (just like `i` and `j`). If you set `pi = 3` your answers won't be very accurate; similarly, avoid using names like `sqrt` for your variables. One of the reasons for using functions is that your code inside the function is isolated from any silly values that might be set by other m-files.
You can examine the value of any variable just by typing its name:

```
>> a
a =

     3
```

```
e = 5 ^ (0.4/1.4)
```
Note the use of brackets! Think whether you want your answer to be like
`5 ^ (2/2)`   (answer 5), or
`5 ^ 2 / 2`   (answer 25/2 = 12.5)

```
f = sqrt(10);
```
Here we are calling a function sqrt. The input arguments are passed in round brackets ().

For other powers, use .^ <u>but be very careful with odd roots of negative numbers</u>, for instance
- -8 .^ (1/3) gives answer -2, but
- [-8, 8] .^ (1/3) picks a different complex root and gives 1.0000 + 1.7321i,   2.0000 + 0.0000i

```
g = sin(0.5)
```
All trigonometric functions in Matlab work in radians. There is no degrees mode. You can of course write a function to convert each way.

```
h = exp(-12.5 * 0.3)
```
exp is the exponential function, whereas 5e3 or 5E3 are just easy ways of writing 5000.

```
k = log10(h)
```
If you use log you get a natural logarithm instead.

Special values
`1 / 0` will give you Inf
`0 / 0` will give you NaN ("Not a Number").

Key points:
- ➢ for scalar calculations, Matlab uses "*" for multiply , "/" for divide (also "\", with special significance) and "^" for powers
- ➢ To input numbers in standard form, use E
- ➢ Standard numbers pi, e, Inf, -Inf, Nan, i and j are built-in but can be over-written e.g. if you do `pi = 4;` then `2*pi` will be 8.
- ➢ NaN stands for "not a number" i.e. Matlab has recognised that the calculation (eg $\frac{0}{0}$) has no sensible answer. This is a very useful feature - NaN and Inf may occur by accident in your calculations (which tells you something about the data) or you can set a value to be NaN so it has some special meaning, such as not plotting on a graph.
- ➢ Calculations give complex number answers wherever appropriate

---

Using a semicolon (;) at the end of each line prevents the result from being shown in the command window, eg:

    a = 1+2;

is neater than `a = 1+ 2`

You can get you code working without the semicolons, then add them so it runs cleanly. (Tip: use the up-arrow key to retrieve previously typed commands).

Key points:
- ➢ In Matlab, "=" is an assignment operator that transfers the numerical answer from a calculation (to right of the "=") into a variable with a name (to left of the "=").
- ➢ You will later see "==" which tests for equivalent values
- ➢ A semicolon at the end of the line means the calculation performs without writing the answer all over the command window

---

Assignments always transfer values (to right of the = sign) into named variables (to left of the = sign)

```
>> a=1+2
a =
     3
```

The opposite is meaningless and will give an error message

```
>> 1+2 = a
??? 1+2 = a
        |
Error: Missing operator, comma, or semicolon.
```

Key point:
- ➢ Variable names must start with a letter and may contain letters, numbers and underscore "_" with up to 32 characters so `half_width`, `p15`, `Speed` would all be valid names but `1+2`, `a&`, `jimmy!` would be invalid

Assembling numbers into a matrix.

To assemble an array containing all the previous results:

```
ah = [a b c d e f]
```

Note that the spaces between variable names here stack them as elements along a matrix **row** (we could use commas too without changing the meaning).

Semicolons are row separators, e.g.

```
ah2 = [a; b; c; d; e; f]
```

will stack elements in a single **column**.

We could achieve the same effect by pressing RETURN after each variable:

```
ah3 = [ a
b
c
d
e
f ]
```

For simple column vectors it is often neater to transpose a row vector, e.g.

```
ah4 = [a b c d e f]'
```

Note how Matlab knows that you have an open set of brackets (starting with "[") to assemble a matrix. Although you press RETURN, Matlab does not interpret the typed information and return to the command prompt >> until you press RETURN after the closing bracket "]". *If Matlab seems to "hang" and do nothing in later exercises it is probably because your brackets are not paired up properly or it has started a* **for** *or* **if** *block and you have not supplied the closing* **end** *command.* `for`, `while`, `if`, `switch`, `try` *are all block opening commands. If stuck, Control-C should abandon the operation.*

Key points:
  ➢ **Round** brackets () are used in the usual BIDMAS sense to force a particular calculation order eg 5 * (2+3), also for feeding an "argument list" of values into a function eg exp(5)
  ➢ **Square** brackets are used for assembling single values, or matrices, into a (larger) matrix (& also for output lists from functions, see later)
  ➢ You will learn about curly brackets {} later
  ➢ Within square brackets, a semicolon (or new line) starts a fresh matrix row whereas spaces, tabs or commas separate items on the same row.

Matrix size and the transpose operator.

```
Terminology.
Array         a list of 0, 1 or many numbers organised into rows, columns
              layers, rooms, houses, estates, cities etc
Vector        a one-dimensional array (a single row (ah above) or a column (ah2 above))
Matrix        a two-dimensional array (which also includes 1-D arrays, i.e. vectors, as a
              special case and empty matrices [] with zero number of elements as another
              special case).  An array with 3 or more dimensions (a block of numbers
              rather than a flat table) is not a matrix.   Matrix multiplication is a special
              operation defined in section 11.
Variable      an array we have created in Matlab and to which a name has been allocated
              (if a = sqrt(10), a  is a variable, 10 is just a number – Matlab cannot
              change its value – and  sqrt is a function).
Scalar        An array containing just one number.
Workspace     The variables that exist at the moment.
```

Try typing **who** (to show a list of variables in use) and **whos** (to list them and show the size of each matrix).

The size is defined as [(number of rows), (number of columns)] so if you type
```
size(ah)
```
(passing the variable ah into the size function) the answer will be [1 8]

Similarly, if $M = \begin{bmatrix} 1 & 3 & -2 & \pi \\ 11 & -54 & -0.0016 & 83 \\ 32 & 0.5 & -78.2 & 0.02 \end{bmatrix}$, M is a matrix with 3 rows and 4 columns.

You can also see the size of variables by looking in the workspace window.

The **transpose** operator swaps rows and columns e.g. M' would have 4 rows and 3 columns.

Assembling two-dimensional matrices

We can also use square brackets to stack together either numbers or entire matrices as larger two-dimensional matrices with more than one row and more than one column. Try:
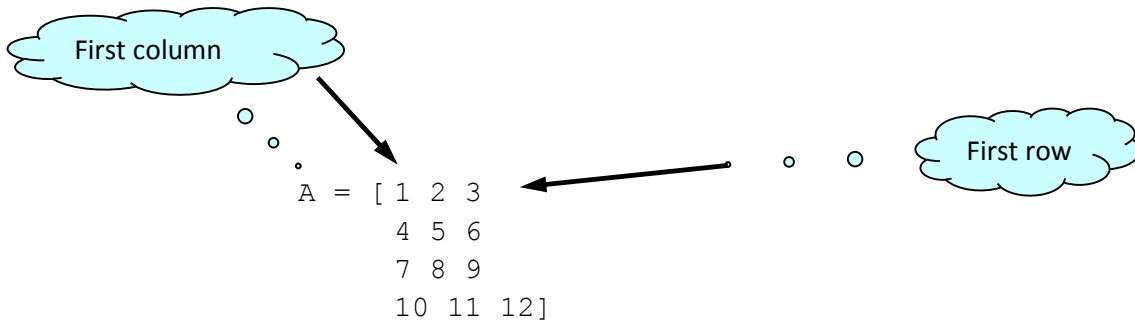
```
D = [1 1; 2 2]
E = [5 10; 5 10]
F = [D E D; E D E]
G = [F, rand(4,1) ];
```
and satisfy yourself as to why these come out as they do. Try also:
```
H = repmat(E, [3 4])
```

Matrix indexing

Consider a matrix

First column

First row

```
A = [ 1  2  3
      4  5  6
      7  8  9
     10 11 12]
```

nb: Remember to press RETURN after each line. You will not get back to the command prompt until you have entered the closing]. If desperate to get out of a mess, press Control-C.

This is held in the computer's memory as a long list of numbers (1 to 12) and I *can* (though we usually *don't*) ask for elements as if it were a vector. Try

```
A(1)
A(2)
A(11)
```

and you will see when you look at the values that it stores all the first column, then all of the second column etc.

Matlab knows that this actually has 4 rows and 3 columns. Usually we would index this using a pair of row and column numbers. Try

```
A(1,1)
A(2,1)
A(1,2)
```

Key points:
  ➢ Indexing goes "the other way" to x-y coordinates - down first, then across
  ➢ We can use indexing to extract the value of an element, or to insert a new value.
  ➢ Some computer languages start their indexing at 0, some at 1, some you can choose either. Matlab always starts at 1.

Indexing multiple elements

Frequently we will want to extract, or insert, more than one element at once. Try

```
A(1:3, 2)
A([1 3 4], [1 3])
```

Very often what we want to do is to access all the elements in a row or a column. In this case I know there are 4 rows and so I could extract all rows of column 1 using:

```
a = A(1:4, 1);
```

In general this is messy and difficult to read (how do I check that there are only 4 rows?). To extract all the elements it is better to use the colon character (:). Try

```
        a   = A(:, 2)
```
All these commands work equally well for inserting new values, eg
```
        A(:,2) = A(:,1) + 10;
```
If the colon is used without any other indexing it treats the variable as being one long list (as in 1 above). Try:
```
        A(:)
```

There is also a special index **end**. Try:
```
        A(2, end)
```

Key points:
  ➢ ":" indexes all the elements in one direction (all rows, or all columns)
  ➢ The last element is "`end`" so "`3:end`" and "`[-2 -1 0]+end`" are valid index lists.

"**Mask indexing**" is an alternative that specifies whether each row should be selected or not, e.g.
```
        B = rand(5,1)
        C = B(B>0.5)   % B>0.5 returns a Boolean true/false (1/0) "mask"
        D = B(logical([1 0 0 1 0]))  % "logical" turns 1, 0 into True/False
```

"for" loops

It is good practice in Matlab to use matrix and array operators wherever possible; this is much faster than using a loop to calculate each individual value. We tend to use a loop when the operations to be carried out cannot easily be "vectorised" eg:

  • reading and processing data from a large number of similar files
  • plotting a series of graphs
  • calling a function that only takes scalar input values for one of its arguments

The general syntax looks like this:
```
        for i = 1:10,  S(i) = sqrt(i); end
```
(Of course, it would be much quicker in this case to do
```
        S = sqrt(1:10);
```
)

nb: Normally a loop would contain many lines of code. There is no need for it to all be on one line, though it may help in the command window when we want to recall a previous instruction if we can get it all at once. Typically one's code would be spread over several lines and indented within the loop for easier reading; it would look like:
```
        for i = 1:10,
            S(i) = sqrt(i);
            % maybe hundreds of lines of code
        end
```

Any loop that enlarges the variable may run very slowly.
Look at your watch and try timing how long

```
        for i = 1:30000, a(i) = 1; end
```
takes to execute before returning to the command prompt. Each time it adds an element it is having to copy the variable to a new place in memory with room for the additional element.

The solution to this is to create the matrix first, eg by using:
```
        a = zeros(1, 30000);
```
or
```
        a = ones(1, 30000);
```
which make a matrix filled with the value 0 or 1 respectively.   Now repeat the
```
        for i = 1:30000, a(i) = 1; end
```
and you will find it runs much quicker.

Key points:
 ➢ You will come across `for….end` loops, `while….end` loops and `if…else…end` blocks. These "flow control" commands affect the execution of lines of code that would otherwise simply run one after another. A loop makes one line run many times. An if-block controls whether a line of code runs at all.
 ➢ The comma after 10 in the first loop is required to separate statements if they are all on the same line
 ➢ The `%` indicates that the rest of this line is a comment rather than a Matlab instruction. We use comments to make code self-explanatory and easily read.
 ➢ Matlab is optimised to be very fast at "vectorised" commands - those involving one function call.   You can often speed up calculations by minimising the number of operations.
 ➢ For instance, `b = a * (2 / 3);` will take half the time of `b = a * 2 / 3;`
 ➢ You should especially avoid loops that increase the size of large matrices.

Vector creation using ":"

The "1:10" in the **for** loop example above is constructing a vector (one-dimensional list of numbers).
One can create a variable like this:
```
        A = 1:5
        B = 2:2:10
```
Most Matlab functions provide sensible outputs when fed with vectors or matrices (2-dimensional lists) eg
```
        SA = sqrt(A);
```

Key points:
 ➢ : is an array constructor that creates a vector rising in steps of 1 (eg `3:5`) or in other steps, eg `2:2:10`.
 ➢ There are other array constructors such as `ones(), zeros(), rand(), eye()`. Try `ones(4,3), rand(2,10), eye(5)`

> ➢ In a loop such as `for i =a`, Matlab takes each column of matrix `a` in turn and assigns it to the loop variable `i`. (You can use any variable name, but the convention for trivial loops with no special significance is to use i, j and k for the loop variables).
> ➢ You can nest loops within loops – take care to use a different loop variable for each loop, otherwise you will get confused.
> ➢ To create and transpose in one statement, you need `(1:5)'`

Matrix addition and subtraction.

Given a matrix, we can:

(i) add, subtract, multiply by or divide by a scalar. Try (starting with matrices A and B from above):

```
A = A + 1
B = B * 2
```

(ii) Add or subtract a matrix of the same size. Try:

```
C = A + B
C=A+a
```

Key point:
> ➢ Matrix addition is simple. If the matrices are not the same size, you get an error message (unless one is a scalar, in which case your answer is the same size as the other matrix).

Matrix multiplication

When multiplying or dividing two <u>matrices</u>, we need to specify the kind of operation required.

Matlab is a "*Matrix Laboratory*" and performs matrix multiplication (as defined mathematically) by default.

If matrix X is defined as having elements $\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$ and Y is defined as $\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$ then the product of X and Y is $\begin{bmatrix} x_{11}y_{11} + x_{12}y_{21} & x_{11}y_{12} + x_{12}y_{22} \\ x_{21}y_{11} + x_{22}y_{21} & x_{21}y_{12} + x_{22}y_{22} \end{bmatrix}$

Multiply
this row by
this column

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} A & C \\ B & D \end{bmatrix} = \begin{bmatrix} aA+bB \end{bmatrix}$$

to get this number

In other words, elements of <u>row</u> **a** of **X** are multiplied by elements of <u>column</u> **b** of **Y**; you add up each product and this becomes element (a, b) in matrix **C**. This is useful for generating linear combinations of values, for instance in evaluating polynomials and defining simultaneous equations.

To perform <u>this</u> operation in Matlab, use **∗**

Try
```
D = [1 10; 100 1000];
E = [1 2; 3 4];
DmE = D * E
```
and satisfy yourself that the values obey the above rules.

One of the basic rules of matrix multiplication states that the result obtained when multiplying the transposes of two matrices (see part 5) is defined mathematically by $(DE)^T = E^T D^T$ and therefore

$$DE = \left(E^T D^T \right)^T$$

Check that this is true for your variables D and E, i.e. that if
```
EtDt = E' * D'
```

then `EtDt` is the transpose of `DmE`.

Note that for matrix multiplication to be possible the number of columns in the first matrix <u>must be the same</u> as the number of rows in the second matrix.
If you try multiplying A and B for instance, you will get an error message telling you that **you have tried to do something stupid**:
```
>> A*B
??? Error using ==> *
Inner matrix dimensions must agree.
```

Check that $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = 14$ and $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$. You will have to think how to enter and multiply these matrices in Matlab.

<u>Key Points:</u>
  ➢ Matrix multiplication is one of the reasons why matrices are so important in Mathematics. Just as $\sum$ gives us an easy way to write the sum of many terms in a series, matrix multiplication gives us a short way of writing a sum with many (possibly thousands) of cross-products. See part 11 below.

- To do matrix multiplication, the second dimension (number of columns) of the first matrix and the first dimension of the second matrix must be equal. The size of the answer = [number of rows from first, number of columns from second matrix] i.e. the other two dimensions.

- Multiplying by an appropriately-sized identity matrix with 1's on the diagonal eg $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

  leaves a matrix unaltered. (This matrix is `eye(3)`).

Array multiplication.

Quite often though we have two matrices of the same size (eg A and B) and want to simply multiply together pairs of elements (the multiplication equivalent of matrix addition). These might for instance be corresponding pairs of voltage and current measurements.

For this Matlab has a special operator `.*` (called "dot times"). For instance
```
F = [1 2 3] .* [1 10 100]
```
gives an answer [1 20 300].

Array division and indices

When dividing or raising to a power we must similarly select between `/` and `./` or between `^` and `.^`

Unless you are trying to solve a matrix equation, the "dot" form is almost certainly the one you want. For instance
```
G = [1 20 300] ./ [1 10 100]
```
will give the answer `[1 2 3]`.

Key points
- Use the .*, ./ and .^ operators whenever you do not want to perform matrix multiplication.
- Using .* etc, both matrix must be the same size (or one scalar), just as with addition.
- Note the difference: if $F = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, `G = F^0.5` attempts to find a matrix `G` such that `G*G` equals F (a matrix square root) whilst `H = F.^0.5` simply takes the square root of each element.

All you have to remember is that in Matlab   *, / and ^ perform **matrix multiplication** and division and *if that is not what you want*, you must use a **different operator** .* or ./ or .^

Using left divide to solve simultaneous equations

We can write a simultaneous equation in terms of matrix multiplication (nb this is mathematical notation, not Matlab code).

Suppose we have a pair of simultaneous equations,

$$p + 2q = 5$$
$$3p + 4q = 11$$

I can write this as a matrix equation

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$$

which is the same as writing $Ax = b$ where

matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, vector $x = \begin{bmatrix} p \\ q \end{bmatrix}$ and vector $b = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$.

The left divide operator solves this using Gaussian elimination (much quicker than finding the inverse of A):

```
x = A \ b
```

Of course, one can have more than two unknowns. Try 1000 unknowns:
```
A = rand(1000,1000);
b = rand(1000,1);
tic; x = A \ b; toc
```
("tic, toc" is to time how long it takes).

Tip:
> You can also **use left divide to do curve fitting**. See the "Matlab Exercises" document, exercise C.

Finding the roots of a polynomial

To solve polynomials (quadratics, cubics etc), use the `roots` function.

The polynomial $p(x) = x^2 + 4x + 3$ can be expressed in Matlab as a vector of coefficients
p = [1 4 3];
The roots function will solve the polynomial with these coefficients:
x = roots(p)

Similarly one could solve the quintic $2x^5 + 10x^4 - 7x^3 - 60x^2 + 4x + 50 = 0$ using:
```
x = roots([2 10 -7 -60 4 50])
```

Useful Matlab functions

Matlab comes with tens of thousands of functions. I have listed some of the more useful ones in the *Matlab Reference Guide*. Beginners sometimes find it hard to remember all the names, so here is an abridged list:

Elementary Mathematical Functions

abs	Absolute value and complex magnitude
acos, acosh	Inverse cosine and inverse hyperbolic cosine
acot, acoth	Inverse cotangent and inverse hyperbolic cotangent
acsc, acsch	Inverse cosecant and inverse hyperbolic cosecant
angle	Phase angle
asec, asech	Inverse secant and inverse hyperbolic secant
asin, asinh	Inverse sine and inverse hyperbolic sine
atan, atanh	Inverse tangent and inverse hyperbolic tangent
atan2	Four-quadrant inverse tangent
ceil	Round toward infinity
conj	Complex conjugate
cos, cosh	Cosine and hyperbolic cosine
cot, coth	Cotangent and hyperbolic cotangent
csc, csch	Cosecant and hyperbolic cosecant
exp	Exponential
fix	Round towards zero
floor	Round towards minus infinity
imag	Imaginary part of a complex number
log	Natural logarithm
log10	Common (base 10) logarithm
mod	Modulus (signed remainder after division)
nchoosek	Binomial coefficient or all combinations
real	Real part of complex number
round	Round to nearest integer
sec, sech	Secant and hyperbolic secant
sign	Signum function (-1, 0 or 1)
sin, sinh	Sine and hyperbolic sine
sqrt	Square root
tan, tanh	Tangent and hyperbolic tangent

Basic Matrix Operations

inv	Matrix inverse
max	Maximum elements of an array
mean	Average or mean value of arrays
min	Minimum elements of an array
polyarea	Area of polygon
repmat	"Tile" a small matrix into a bigger one

sort	Sort elements in ascending order
std	Standard deviation
sum	Sum of array elements
trapz	Trapezoidal numerical integration

Solvers and engineering functions

diff	Differences and approximate derivatives
eig	eingenvalues and eigenvectors
fft	discrete Fast Fourier Transform
fminbnd	find the minimum of a function
fzero	find where a function is zero
ode23, ode45	solve ordinary differential equations
pdepe	solve 1D partial differential equations
pwelch	estimate power spectral density
roots	find roots of a polynomial

Character String Functions

deblank	Strip trailing blanks from the end of a string
findstr	Find one string within another
int2str	Round matrix to integers and convert to character string format
lower	Convert string to lower case
num2str	Convert matrix to character string format (about 5 sig. figures)
strcmp	Compare strings
upper	Convert string to upper case

Things to avoid

| Matlab code	Explanation
`2 pi`	Matlab knows what 2 is and it knows what pi is – but what do you want to do? Add them? List them out? Multiply them? You need an operator.
`for i = 1:10,` `  s(i) = sqrt(1)`	Now it sits there doing nothing. This is deliberate. It is waiting for you to complete the **for** loop with a final "**end**" – then it will execute.
Try	`try…catch…end` is a Matlab flow control block that traps any errors and executes code in the "catch" section to deal with them. If you type "try", Matlab will just sit there waiting for you to add the closing "end". Press Control-C to kill it. Other block commands are if…elseif…else…end switch…case…otherwise…end for…end while…end

2. Using script and function m-files

Matlab "m-files" are plain text (ascii) files of Matlab commands. There are 2 kinds:

- ➢ "script" files
    - o These are good for quick, one-off calculations and plotting where you may want to experiment, make changes and re-run it until you are happy.   Run the file by typing its name in the command window or by pressing the Run button in the editor. Run bits of it using the right mouse button.
- ➢ "function" files, which start with the keyword "function"
    - o Functions are good for writing re-usable programs that perform some particular task that may be useful in future. They are much more powerful as they allow a complicated task to be broken into many simple, well documented stages that can even be written by different members of a team.
    - o Function files can have further functions hidden within them

There are many ways of opening the editor, for instance:

(a) In the main Matlab window, open the editor by clicking on the white (left hand) icon:
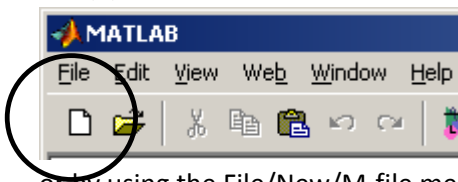
or by using the File/New/M-file menu, or

(b) type

```
edit test1
```
and it will open file plot1.m or create it if it does not exist, or

(c) double click on an m-file in the "Current folder" part of the main Matlab window.

(but please, avoid double-clicking an m-file in Windows Explorer as it may start a whole new Matlab session or even get confused and start a Mathematica session).

Making a simple script file

In the editor, type:

```
for i = 1:3,
A = rand(4,4);
    b = rand(4,1);
    x(:,i) = A \ b;
end
x
```

Save the file as `test1.m` and run it by clicking on the editor run button or by typing `test1` in the command window. It just solves 4 random simultaneous equations.

Making a function file

Let's make a simple function to convert degrees to radians.

Type

```
edit d2r
```

in the command window to create a new m-file (d2r.m).

```
function y = d2r(x)
% y = d2r(x),   Converts degrees to radians
y = x * (pi/180);
```

Now save the file as d2r.m

Back in the command window, check that it works:
```
>> d2r(180)
ans = 3.14159
```

You can now use this in expressions like
```
>> sin(d2r(30))
```

Argument lists for functions

You can have any number of input variables and output variables. When you call the function, <u>it is the order in the list that matters, not the name of the variable</u>.

For instance I could create a file `copy3.m`:

```
function [x, y, z] = copy3(a, b, c)
% function [x, y, z] = copy3(a, b, c)
% This just passes a into x, b into y etc

x = a;
y = b;
z = c;
```

Note the initial comment. It is good practice to explain what the function does. In this case it is pretty trivial!

Now in the command window, let's type

```
A = 1; B = 2; C = 3;
[X, Y, Z] = copy3(A, B, C)
```

Variable **A** (or it might be a number) in the top-level workspace becomes variable **a** inside the function workspace; its value is put into variable x which is then "returned" to the "calling"

workspace using the output argument list in square brackets []. In the main workspace the returned value is put into variable **x**.

You might like to try

```
[X, Y, Z] = copy3(C, B, A)
```

or

```
[Y, Z, X] = copy3(A, B, C)
```
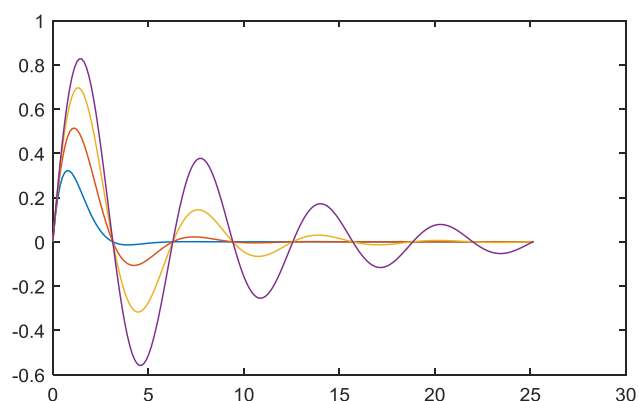
and think what is happening.

Another example

Make a function expsine.m:

```
function v = expsine(t, T);
% function v = expsine(t, T);
% This is a function for generating a decaying sine wave

v = exp(-t./T).*sin(t);
```


Then in the command window, type:

```
x = [0:200]'*(8*pi/200);
for T = [1 2 4 8],
  plot(x, expsine(x, T))
  hold on
end
```

You should get 4 sine waves with different decay rates:

Matlab's **sort** function is a good example that uses more than one output argument:

```
>> X = [9 11 2 3.14 -3 0];
```

```
>> [YS, I] = sort(X)
YS =
   -3.000  0  2.000  3.1400  9.000  11.000
I =
      5      6      3      4      1      2
```
(YS is the sorted version of X, I tells us which elements of X were used for each element in Y i.e. the first element of YS comes from the 5th element in X).

Key points:
- ➢ A function file starts with the word **function**.
- ➢ The first non-comment line must look like
  ```
  function [A, B, C] = name(a, b)
  ```
  where a, b are input arguments and A, B, C are output arguments. "name" is the m-file name unless you are making sub-functions.
- ➢ Functions are much more useful than script files because they can be written without worrying about whether one file will accidentally change the value of a variable used by another. They provide "reusable code" that will reliably work for years.
- ➢ There are "debugging" tools in the editor that let you pause execution at a line so that you can check values before continuing. You can also look up into the workspace from which the function was called.
- ➢ Matlab will only run your m-files if they are in its current folder or in a folder listed in the Matlab PATH. When it reads a word, Matlab asks itself:
  - o is it a variable in the workspace?
  - o is it a file in the current folder?
  - o is it on the path
  - o is it a built-in function?

  (in that order). If it cannot be found, you get an error message.

Typical usage of script and function files

It is *good programming practice* to separate a calculation from the data that goes into it. That way you can keep one version of the calculation code and run it with multiple data sets. (Just think, if you had multiple versions of the calculation code you would have to really carefully update every one of them whenever you made a change to the calculation method - *yuck!*).

We do this by creating one **script file** to assemble the data. When we run the script file it:

- Defines the data
- Automatically passes it into one or more **functions** that do the calculation

We can have multiple versions of the script file to model different scenarios or even one with a loop to increment through a list of data values (e.g. expsine on the previous page – the function is called inside a loop).

Alternatively one can start with a function instead of a script file. This is commonly done when writing menu-driven apps; if not doing this, the script-driven format is easier.

3. Flow control: choosing which code to run.

So far we have told Matlab exactly what to do.

It is often useful to let Matlab choose what to do. Enter the following code as script file **iftest.m**. What output would you expect? Now run it and see what it gives.

```
for i = 1:5,
     if i==1,
                disp('i = 1')
     elseif i==2,
          disp('i = 2')
     else
          if i<4,
               disp('i = 3')
          elseif i<3.5,
               disp('This line never executes')
          elseif i>4,
               disp('i = 5')
          else
               disp('i = 4')
          end
     end
end
```

This is a trivial example: in calculations there are often times when we must decide how or whether to do a calculation (e.g. are the roots real? Which one do we want?). The indenting makes it easier to read (Select text, right-click and "Smart indent") but does not affect the result.

Key points:
- There is a huge difference between = and ==
  - = performs an assignment i.e. it puts numbers into a variable.
  - == tests whether two numbers are equal and returns a value of 1 if true and 0 it untrue (try just typing 3==3 and 1==2 into the command window and looking at the result).
- We can "nest" one if…end block inside another. Here we are starting a second if-block on line 7. Note that the if here in else if starts a new if-block whereas elseif (one word) is simply part of an existing if-block. If your code does not run, you may well have opened an if-block and not finished it with end.
- If you select your code, right click and pick "Smart indent" it will align it nicely on the page. Check that each if-block finishes where you expect it to.
- Most people use a comma to end each line starting with if or elseif. This is not strictly necessary but makes the code more easily readable.
- disp is a function that displays a number or character string in the command window.

<u>Further niceties</u>

When using `if` blocks we must take care when comparing matrices. In the command window, create a matrix

```
a = [0 1 2; 1 2 3]
```

and then try :

```
if a>1, disp('a is > 1'); else disp('a is not > 1'); end
```

You will get the second message because **not all** of the elements of `a` are >1.

You can see how it works if you type:

```
        b = a>1
        whos
```

Here the "=" assigns the result of the > comparison to the "logical" variable `b` which shows 1 where the `a>1` test returns true and 0 in other positions. It is good practice to express precisely what we mean an **if....end** block to achieve by using the functions `any` and `all` to reduce `b` to a scalar 0 or 1.

Try:

```
        c = any(b)
        d = all(b)
        e = any(any(b))
```

(Functions such as `sum` that operate on a matrix operate on each column individually. Here `b` is a 2*3 matrix and `any(b)` is a 1*3 row vector containing the results for each column, a "1" where <u>any</u> of the column elements is non-zero. Similarly, `all(b)` will give a column result of 1 only if <u>all</u> the column elements are non-zero).

A more clearly readable form of the first expression would thus be something like:

```
if any(any(a>1)), disp('Some elements of a are > 1');
      else disp('No elements are  > 1'); end
```

<u>Key points:</u>

> ➢ Good code makes it very clear how *if*-blocks execute.
> ➢ When testing the values in a matrix, you should always use all or any to ensure that the code behaves as expected.
> ➢ The logical operators ==, >, >=, <, <=, ~= return a logical true/false (1/0) "mask". If needed, `find` will identify the "true" elements e.g.
> ```
>         a = rand(6,1)
>         b = a>0.5    % 1 if true
>         c = find(b)
> ```
> `a(b)` and `a(c)` are equivalent ways of getting those elements of `a` that are >0.5.
>
> Try also: `a=[0 1 2; 1 2 3],  [R, C] = find(a>1)`

4. Other data types: strings (char), structures, cell arrays. Using `disp`

Strings

We can make a string by enclosing it in inverted commas (not speech marks):

```
mystring = 'Time since midnight (minutes) '
xlabel(mystring)
whos          % whos shows mystring as a "char" data type
```

We can convert a number or matrix to a string, e.g.

```
        int2str(pi)
```

gives us the answer '3' (string format and rounded to an integer), while

```
        num2str(pi)
```

gives us the answer '3.1416' (string format and rounded to about 5 s.f.).

So we could use some code like:

```
        c = clock;  % see "help clock" for what this does
        secs = sum(c(4:6) .* [3600 60 1]);
        mystring = ['Time since midnight (minutes) = ',…
                        int2str(secs)]; % … continues the line
        title(mystring)
```

Note that the square brackets are concatenating the "Time since…" string and the "seconds" string.

disp

`disp` will display the bare contents of a variable (without "name = ") in the command window e.g.

```
        disp(mystring)
        disp(clock)
```

disp only takes a single input argument but we can use [ ] to assemble several matrices <u>with the same number of rows</u> into one large matrix, e.g.

```
        volts_amps = rand(20,2)*10;
        M = [repmat('Wire ', [20 1]), int2str((1:20)'),...
                repmat(':  ' , [20 1]), num2str(volts_amps)];
        disp(M)
```

Structures

Structures are created whenever we assign to a variable with a name containing a ".", eg:

```
        >> S.a = rand(22,2);
        >> S.b = 'some text';
        >> S.c.x = 1;
        >> S.c.y = 2;
        >> S.c(2).x = 3;
        S =
            a: [22x2 double]
            b: 'some text'
            c: [1x2 struct]
```

One can also use the struct function:

`S = struct('field1',VALUES1,'field2',VALUES2,...)` creates a structure array with the specified fields and values (see the help).

In use, one simply specifies within a calculation the indexed part of the structure that returns a conventional double, cell or char array, eg:

```
S.a = S.a + 1;
```

(since `S.a` is a 22*2 double).

One can have any number of fields and levels (structures within structures), though you may crash the system if you create more then 2000 levels! There is complete flexibility e.g. in the above variable, `S(1,2).a = 1;` would be allowed. `S(2).a` is then a different size to `S(1).a` – in fact it may even be a different type, though this would be **bad style**.

Examples of how one might use structures

(1) Structures are useful for handling and identifying data e.g.
```
load weather_2003.mat rain sun temp
weather(1).year = 2003;
weather(1).rain = rain;
weather(1).sunshine = sun;
weather(1).temperature = temp;
weather(1).readme = …
    'Monthly mm rain, hours sunshine and mean deg Centigrade';

% now add some more data:
clear rain sun temp
load weather_2004.mat rain sun temp
weather(2).year = 2004;
weather(2).rain = rain;
etc
save all_weather.mat weather
```
 Now we have a structure which is self-explanatory if you load it from a file.

(2) Passing arguments into a function.
Looooking at the input argument list for a function, the variable names *inside* the function are allowed to be different to what they were *outside* (in the calling workspace) **but the structure's field names do not change**.   So instead of having a large number of individual input arguments (with risk of getting the order wrong!) we can use just one or two structure variables.

For instance:
(a) I could create a function to look up the properties of a variety of heat transfer fluids and make the function return these properties as a structure:

```
>>  fp = fluid_properties('water', [], 50)

fp =
          c: 4182
          k: 0.6387
         mu: 5.4710e-04
        rho: 988
    th_diff: 1.5459e-07
    th_prod: 1.6245e+03
         Pr: 3.5821
         TC: 50
       Type: 'water'
       Conc: []
```

Fairly self-explanatory? I can now pass these properties into a function to analyse the performance of a heat exchanger:

```
effectiveness = heatx(fp, mass_flow, area)
```

which is a lot neater than

```
effectiveness = heatx(c, k, mu, rho, Pr, mass_flow, area)
```

(b) If I were designing a bridge, I could define the design parameters as a structure and pass it into a function to calculate the strength:

```
design1.length = 50;    % length in metres
design1.height = 4;     % height in metres
design1.cable = 0.02;   % cable diameter in metres
safe_load = bridge_calculation(design1);
              % pass these numbers into a calculation function
```

(c) If I were analysing the performance of a racing car, I could have a structure "`car`" with all the design properties (`car.mass, car.power, car.drag_coeff` etc) and a structure "`race`" with all the race track features (`race.length, race.altitude, race.temperature, race.wind` etc). I could write a function that updated the `race` structure with a field `race.lap_time` (see Matlab course sheet F for detailed example):

```
race = race_predictions(car, race);
```

Cell arrays

Each element in a cell array is a "cell" that may contain an array of any data type (and if the cell content is itself a cell, one can go on ad infinitum). Cell arrays are indexed in two ways, depending on whether we are interested in the cell or its contents.

Empty cell arrays can be created using the `cell` function (equivalent to `zeros`):

```
>> C = cell(1,3)
C =
    []    []    []
```

and one can then insert elements, eg

```
C{1} = 'hello world';
% note curly brackets – we are inserting the cell contents
Or
C(2) = { [1 2 3] };
% note curly brackets – we have created a cell and then
% put it in the array.
```

In terms of assembling cell arrays,

```
>> C = {'hello world', [1 2 3], S}
```
and
```
>> C = [ {'hello world'}, {[1 2 3]}, {S} ]
C =
    'hello world'    [1x3 double]    [1x1 struct]
```

are equivalent.

The most common use of a cell array is to hold lines of text. If they were all the same number of characters they would fit in a char array; if not, we put each line in a separate cell, eg:

```
>> a = dir('m*')  % this gets a list of files in your current folder
a =
7x1 struct array with fields:
    name
    date
    bytes
    isdir

>> names = {a.name}'
      % (take all instances of this field and put into a cell array)
names =
    'Matlab Function Help.doc'
    'MatlabExercises.doc'
    'MatlabNotes.doc'
    'MatlabTalk'
    'mortgage1.m'
    'mortgage2.m'
    'Mortgage3.m'
```

Alternatively could use:

```
names = cell(7,1);
[names{:}] = deal(a.name);
```

5. Loading and saving data

There are two general classes of Matlab file:

- files with a `.m` extension, which contain Matlab code (script or function m-files)
- files with a `.mat` extension, which contain data in a binary format

Create some data:
```
a = 1;
b = 'Monday';
c = rand(3,5);
```

Now save these variables as a binary data file:
```
save data1.mat a b c
```

This file is basically an exact copy of these variables, just transferred from the workspace to the disk. Try typing:
```
clear
whos
```
(Now the workspace is empty). Then
```
load data1.mat
whos
```
and you see that the all variables have been read from the file.

Now try:
```
clear
S = load('data1.mat')
```

All the variables in the file are created as fields of a structure variable S. This is really useful – you can for instance very easily save a number of different variables (e.g. data acquired in LabView: voltage, time, transducer names, date) to a Matlab file and then load them "bundled together" into a structure.

Just for interest

The save command is a strange function that knows both the name of each variable and the numbers inside it. Matlab has what it calls "command & function duality" which means that
```
save data1.mat a b c
```
(a "command") and
```
save('data1.mat', 'a', 'b', 'c')
```
(a function call) mean exactly the same thing. You might expect that the function only sees the character strings 'a', 'b', 'c'. In fact it (effectively) has a command (`evalin`) for retrieving the variable values from the workspace which calls it.

6. Plotting

The `plot` function takes pairs of input arguments eg `plot(x, y)` where `x` is a matrix of *x*-coordinates and `y` is a matrix of *y*-coordinates; one can also specify a line style, eg `plot(x, y, '-+')`. Try the following:

```
plot(1, 2)
```

(the little spot in the middle is the coordinate plotted).

```
plot(1, 2, 'x')
help plot
```

Read the list of colours, point styles and line types. Try some of the different point styles (^, >, o, + etc)

Key points

> In Matlab, a function does not necessarily simply "convert an input into an output" as in ordinary mathematics. It can have any number of input and output arguments (including none) and its purpose may be to "do something" such as creating a graph.
> The `plot` function identifies numerical inputs as the x- or y-halves of coordinates and string inputs (supplied in quotes eg 'b') as settings for line or marker colour and style.

If we plot more than one coordinate, Matlab will draw straight lines between them.

```
plot([1 2], [3 4])
```

Note that this is a line from coordinates (1,3) to (2,4).

Try plotting a sine wave:

```
x = [0:10] * 2*pi/10;
plot(x, sin(x))
```

Now repeat using 200 points to give a smoother curve. (Matlab never draws curved lines – to draw a curved line we simply supply a large number of coordinates such that the straight line segments between them are very short).

Note that if you simply typed

```
plot(sin(x))
```

you would get x-axis numbers 1, 2, 3 etc corresponding to the number of elements in `sin(x)`.

Across the top of the figure window you will see a toolbar. Click on the (+) symbol for the zoom-in tool:
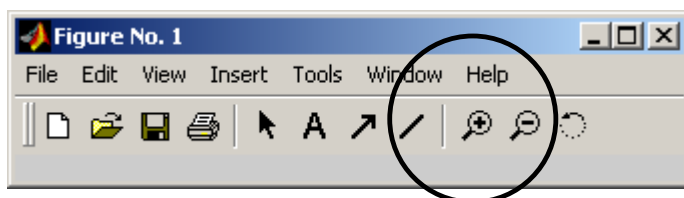
Clicking within the axes box will now enlarge the view. You can also hold down the left mouse button while dragging the mouse to draw a box – it will then expand the part of the curve within the box to fill the axes. Click with the right mouse button to reverse the effect (or use the (-) tool), or select the arrow button to turn off zooming.

The right-hand of these three tools is for 3D rotation. Select this tool and try rotating the axes. Now try

```
surf(peaks(30))
```

and see how you can rotate a 3-D plot.

Go back to the two-dimensional sin(x) plot. Now try expanding the region drawn using the axis function:

```
plot(x, sin(x))
axis([0 3 0 1])
```

This specifies the x-axis limits (0 to 3) and the y-axis limits (0 to 1).

---

nb.   The **axis** function sets x- and y-axis limits
        (but is reset by `plot`, so call <u>after</u> the `plot` command)

        The **axes** function creates a whole new axes box – *not the same thing at all*!

Plotting more than one line.

There are several ways of creating multiple lines on a graph:

(a) we can plot a matrix, in which case each column becomes a new line

(b) we can lengthen the plot command to draw them all at once

eg   `plot(x1, y1, x2, y2, x3, y3)`

(c) we can draw one line, "`hold`" the plot so that it does not get automatically deleted, then plot additional lines.

We will plot the curve $y = e^{\frac{-t}{T}} \cos t$ against time t for different values of the "time constant" *T*. This curve represents the motion of something vibrating (a mass on a spring, or current in a tuned circuit) that gradually loses energy; the amplitude decreases by a factor $\dfrac{1}{e}$ each time *t* increases by the time constant *T*. (See graph on page 20).

(a) <u>Plotting all the columns of a matrix:</u>

```
t = [0:200]' * 8*pi/200;
y = cos(t);
T = [1000 50 20 5];
V = zeros(201, 4);
for i = 1:4,  V(:,i) = y.*exp(-t/T(i)); end
plot(t, V);
grid
xlabel('Time (seconds)')
ylabel('Voltage')
legend({'T = 1000', 'T = 50','T = 20', 'T = 5'})
```

The quotation marks here signify that the words between them are a character string (a special kind of variable) – without these Matlab would try looking for a function `Time` and a variable `seconds` to pass into it.

The curly brackets {} in the legend definition are setting up a cell array – this gets around the problem with matrices that each row would have to contain the same number of characters.

➤ Try dragging the legend to another position on the graph.

Note that Matlab is "case sensitive" so that *T* and *t* are different variables. Look at the code carefully:
  o How do I know I will need 201 rows for `V`?
  o What does the `:` do in the for-loop?
  o Why do I need `.*` but not `./` in the for-loop line?

(b) Using multiple inputs to the plot command
Create a new figure window `>> figure`
(You can have as many as you want, but more than 10 will get confusing).
```
plot(t, V(:,1), t, V(:,2), t, V(:,3), t, V(:,4))
```
It should look the same as the first figure. This syntax is useful if your vectors contain different numbers of points eg
```
plot(t, cos(t), [0 2], [0 2]).
```
What is the solution of $t = \cos(t)$? Zoom in to the point of intersection, type
```
xy = ginput(1)
```
and click on the point. The coordinates will appear in the command window.

(c) Using `hold`.
Try typing:
```
plot(t, V(:,1))
```
and then
```
plot(t, V(:,2))
```

The second plot over-writes the first! Now try:
```
plot(t, V(:,1));
hold on
for i = 2:4, plot(t, V(:,i)); end
hold off
```
This tells Matlab to keep the previous line while adding new ones.

Key points:
  ➤ You can plot all the lines at once (by plotting a matrix, or with lots of pairs of x, y input arguments to the plot function)
  ➤ Alternatively, you can "hold" your first plot so that subsequent plots add to it rather than starting again and deleting the previous one
  ➤ See also: `plot3, surf`