MATLAB TUTORIAL

MARCUS. R. GARVIE
UNIVERSITY OF GUELPH

Note: I wrote these notes a number of years ago as a brief introduction to the different aspects of using and coding in MATLAB for a group of Engineering students. Of course many things have been added since then, but the basic commands discussed in this tutorial should be (mostly) unchanged.

M.R.G. 2009

```
% THE NUTS & BOLTS OF MATLAB
% --------------------------

% Screen input

4

ans =

     4

% Screen output displayed
% Simple Calculations

57 + 39.5

ans =

   96.5000

% Current answer always stored as 'ans'

ans

ans =

   96.5000

% The math symbols are fairly standard.
% Multiplication: *, Exponential: ^, Division: / , etc

% The format of output can be changed. However, all calculations are
% done by Matlab to 15 d.p.

22/7

ans =

    3.1429

% 'format long' allows the diaplay of more d.p.

format long

22/7

ans =

   3.14285714285714

% 'format' returns the output to 5 d.p.

format

% Try also 'format rational' for rational approximation

pi

ans =

    3.1416

format rational

pi

ans =
```

```
   355/113

format

% Notice that Matlab is case sensitive. All commands are done
% in lower case. Also, upper and lower case variables are
% treated differently.

count1=5

count1 =

     5

count2=6

count2 =

     6

totalcount=count1+count2

totalcount =

    11

% Variables can be a mixture of letters, numbers and some
% symbols (lower case '_' is very useful). Variables must
% not start with a number.

% The semicolon ';' after a command suppresses printing. This
% is very useful when we don't want to see all the intermediate
% steps in a calculation.

A=100000;

% (No output shown). But 'ans' still = 100000

% The comma ',' allows us to type several commands on a single
% line (so does ';', but display suppressed).

A=34; B=45; C=23; Total=A+B+C

Total =

   102

% numerical calculations are easily achieved using complex
% numbers. The rules are the same as for ordinary numbers.

sqrt(-1)

ans =

        0 + 1.0000i

u=3+4i; v=5-2i; sum=u+v

sum =

   8.0000 + 2.0000i

% Common functions are listed at the end these pages. E.g.,
% 'exp(x)', 'abs(x)', 'sin(x)', 'sqrt(x)', 'log(x)', 'round(x)', asin(x)
% (Note that Matlab operates in radians, not degrees).

% To identify what variables you have currently assigned:
```

```
who

Your variables are:

A               Total           count2          u
B               ans             sum             v
C               count1          totalcount

% The 'help' command can be placed before any command & provide
% a quick and clear summary of the command.

% for example:

help diary

 DIARY Save text of MATLAB session.
    DIARY file_name  causes a copy of all subsequent terminal input
    and most of the resulting output to be written on the named
    file. DIARY OFF suspends it. DIARY ON turns it back on.
    DIARY, by itself, toggles the diary state.

    Use the functional form of DIARY, such as DIARY('file'),
    when the file name is stored in a string.

% I am currently using a diary to save the current session. Later
% I can open the file from the file menu.

diary off
```

```
% WE NOW LEARN THE BASICS OF CONSTRUCTING & USING VECTORS
% ------------------------------------------------------

% To construct vectors is easy. Use square brackets & put
% spaces between the elements.

x=[1 2 3 4]

x =

     1     2     3     4

% It is traditional to use small letters for vectors and
% capitals for matrices. We have constructed a row vector.
% To get the column vector we take the tranpose. In Matlab
% this is achieved by using a ' symbol after the vector.

x=x'

x =

     1
     2
     3
     4

% Or :

x=[1 2 3 4]'

x =

     1
     2
     3
     4

% There are many efficient ways to construct vectors. The easiest
% has the format: x = start:increment(or step):last.

x=1:1:4

x =

     1     2     3     4

% If the increment is just unity, it can be omitted.

x=1:4

x =

     1     2     3     4

% Another example:

y=1:2:9

y =

     1     3     5     7     9

% It is intuitive how we identify the elements of a vector.

y(3)

ans =
```

```
     5

% Or identify a specific range of elements in a vector.

y(3:5)

ans =

     5     7     9

% We can also change specific elements.

y(3)=89

y =

     1     3    89     7     9

% Another useful way of constructing vectors allows us to specify
% how many elements we want, and the first and last elements. This
% has the structure: g=linspace(first,last,number of elements).

g=linspace(0,10,6)

g =

     0     2     4     6     8    10

h=linspace(3.5,7.3,8)

h =

  Columns 1 through 7

    3.5000    4.0429    4.5857    5.1286    5.6714    6.2143    6.7571

  Column 8

    7.3000

% ( Notice that Matlab considers a vector to be just a 1-by-n matrix.)

h=linspace(3.5,7.3,7)

h =

    3.5000    4.1333    4.7667    5.4000    6.0333    6.6667    7.3000

% Inner Product / Dot Product (x.y). This is best done in Matlab via
% matrix multiplication. I.e., we first multiply a column vector x
% by the transpose of a column vector y, and then take the square root
% (sqrt in Matlab). Consider the dot product of x with itself (norm):

x=[3 4];
sqrt(x*x')

ans =

     5

% This obviously corresponds to the 3,4,5 triangle.

% You will no doubt remember from your lecture notes the equation
% for the angle between two vectors, based on the dot product. I.e.,
% the angle theta between two vectors v and w is given by
% arccos{(v.w)/norm(v).norm(w)}. In Matlab for arccos we use 'acos'.

e1=[1 0 ];
```

```
e2=[0 1];
% The norms of these vectors are obviously unity.
acos(e1*e2')      % Notice the transpose symbol with e2
ans =
    1.5708
% This answer is in radians. To convert to degrees:
ans=ans*180/pi
ans =
    90      % degree
% As expected. Lets do a more general example.
v=[5 2]; w=[3 5];
norm_v=sqrt(v*v'), norm_w=sqrt(w*w')
norm_v =
    5.3852

norm_w =
    5.8310
rads=acos((v*w')/(norm_v*norm_w))
rads =
    0.6499
theta=rads*180/pi
theta =
   37.2348      % degree
% We can use this data to find the area of the parallelogram
% spanned by the vectors v and w:
area=norm_v*norm_w*sin(rads)
area =
    19          % units squared
% We can use vectors as the argument of a function.
x=1:8, y=2*x
x =
     1     2     3     4     5     6     7     8

y =
     2     4     6     8    10    12    14    16
```

```
% A very powerful facility allows us to have arguments between
% vectors, where the operations act 'element by corresponding
% element'. With addition/subtraction and scaler multiplication
% this is exactly what happens, but when we multiply two vectors
% together, or we have a more complicated operation we need to be
% able to tell Matlab what to do. In Matlab we simply put a
% . (dot) in front of the operation(s) concerned. E.g., using
% the above vectors x and y:

z=x.*y

z =

     2     8    18    32    50    72    98   128

% Or even (!):

d=sin(x)./(y.*x)

d =

  Columns 1 through 7

    0.4207    0.1137    0.0078   -0.0237   -0.0192   -0.0039    0.0067

  Column 8

    0.0077

% Traditional programming structures would a require a double loop
% in order to achive this (more on this later).

diary off
```

```
% WE NOW LEARN HOW THE BASICS OF CONSTRUCTING & USING MATRICES
% ------------------------------------------------------------

% We can construct matrices from vectors if we wish. Given two
% vectors v and w we form the matrix, where the columns of the
% matrix are the column vectors v and w. The syntax for this is
% intuitive. E.g. :

v=[1 3]',u=[4 5]'

v =

     1
     3


u =

     4
     5

A=[v u]

A =

     1     4
     3     5

% Try to use upper case letters to denote matrices.
% Another example would be A=[e1 e2]=I.

% To construct matrices from scratch we enter the rows of
% elements separated by semi-colons. Again, put a space
% between each element.

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

A =

     1     2     3
     4     5     6
     7     8     9

% We can identify elements in this matrix easily.

A(2,3)

ans =

     6

% Or change the value of elements.

A(3,2)=78

A =

     1     2     3
     4     5     6
     7    78     9

% A useful feature is the facility to pick out specific
% columns (or rows). We achieve this with a colon (:).

A(:,2)

ans =
```

```
     2
     5
    78

% The colon tells Matlab that as no specific row is indicated
% we need ALL elements in column 2. Of course you can use this
% to change the column/row of a matrix. There are a whole host
% of clever ways to amend + construct matrices. Some specialized
% ones are shown here.

ones(2,2)

ans =

     1     1
     1     1

zeros(3,2)

ans =

     0     0
     0     0
     0     0

eye(5)

ans =

     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

% Notice that as identity matrices are always square, the eye
% command has only one input number.

rand(2,2)

ans =

    0.9501    0.6068
    0.2311    0.4860

% The elements of this matrix are random numbers between 0 - 1.

rand(2,2)*10

ans =

    8.9130    4.5647
    7.6210    0.1850

% We now have a matrix that has random elements between 0 and 10.
% This also demonstrates scaler multiplication of matrices.

% Matlab is an exellent environment in which to learn linear
% algebra. Consider some simple matrix algebra.

% A=[2 3;0 6], B=[1 8;3 2]
A=[2 3;0 6], B=[1 8;3 2]

A =

     2     3
     0     6
```

```
B =

     1     8
     3     2

C=A+B

C =

     3    11
     3     8

A*B,B*A

ans =

    11    22
    18    12


ans =

     2    51
     6    21

% Thus we can see that matrix multiplication is not
% commutative. You should verify the other matrix
% algebra rules for yourself.

% Finding the determinant of a matrix:

det(A)

ans =

    12

% Notice the following:

det(A)*det(B)

ans =

  -264

det(A*B)

ans =

  -264

% We can also find the inverse of a matrix:

inv(A)

ans =

    0.5000   -0.2500
         0    0.1667

% And again verify results in linear algebra:

inv(A*B)

ans =

   -0.0455    0.0833
```

```
    0.0682   -0.0417

inv(B)*inv(A)

ans =

   -0.0455    0.0833
    0.0682   -0.0417

% Notice the order in which we did this.

% Recently you learned (or will do soon !) how to find the
% eigenvalues and eigenvectors of a matrix. These can be
% easily found in Matlab. You should realize that this
% topic of Math is one of the most important Mathematical
% areas in engineering, cropping up in a whole host of
% practical problems.

eig(A)

ans =

     2
     6

% Try verifying the Cayley Hamilton theorem for this case
% ("the matrix satisfies its own charateristic equation").

% To find the eigenvectors and eigenvalues we need a slightly
% more complicated syntax. We input an expression looking like
% [V,D]=eig(A), where V is a matrix with the columns composed
% of the eigenvectors, and D is a diagonal matrix of eigenvalues.

[V,D]=eig(A)

V =

    1.0000    0.6000
         0    0.8000


D =

     2     0
     0     6

% Matlab can easily be used to solve a linear system of equations,
% by first writing the system in matrix form. Consider :

%              x + y    = 1
%              x/2 + 2y = 1

% We re-write this system in the form:  Ax = b, and solve via
% x = (A^-1)b. Thus for the above problem:

A=[1 1;0.5 2], b=[1 1]'

A =

    1.0000    1.0000
    0.5000    2.0000


b =

     1
     1
```

```
x=inv(A)*b

x =

    0.6667
    0.3333

format rational

x

x =

     2/3
     1/3

% You should realize that solving large systems of linear equations
% via the inverse of the coeficient matrix (an EXTREMELY common
% requirement in Engineering Math) is very inefficient and prone
% to error, resulting from the accumulation of roundoff error. Part
% of the problem results from the large number of 'floating point
% operations' needed for Matlab to calculate inverses. Also, there
% is the issue of what is called 'conditioning'. If the conditioning
% of the coeficient matrix is poor this means that small changes in
% elements of this matrix can result in widely different solutions -
% thus we are not likely to have confidence in the answers we get.
% Small changes in the elements of the coeficient matrix can occur
% due to errors in experimental observation. In Matlab we can minimize
% these problems by doing A\b (divide b on the left by A) instead of
% doing inv(A)*b. This is equivalent to performing Gaussian Elimination
% in order to solve the system of equations.

x=A\b

x =

     2/3
     1/3

% Notice that 'dividing on the right' is not defined here:

A/b

Error using ==> /
Matrix dimensions must agree.

% In general to do (A^-1)xB do A\B, and in order to do
% Bx(A^-1) do B/A.

% Matlab has a facility for telling us whether a matrix is
% ill-conditioned. We use the command 'cond', which returns
% a number. The larger this number, the more ill-conditioned is
% the matrix. A matrix that is perfectly well-conditioned has
% a condition number of unity.

cond(A)

ans =

  1318/337

format

cond(A)

ans =

    3.9110
```

```
ILL=[0 1;-1000 -1001]

ILL =

           0           1
       -1000       -1001

cond(ILL)

ans =

  2.0020e+003

% Finally, a useful command in Matlab gives the row-reduced-
% echelon form of a matrix.

rref(A)

ans =

     1     0
     0     1

% As expected, because there was a unique solution to the problem.

diary off
```

```
% PLOTTING GRAPHS IN MATLAB
% -------------------------


% Matlab has many powerful & easy to use graphics facilities
% They are based on the use of vectors and matrices.

% plotting a point:

plot(2,3)

% This is difficult to see. We can plot various symbols
% and specify color & line characteristics (see handout):

plot(2,3,'k*')

% You should try 'help plot'

% We can plot several points at once, using the same
% command, but with options separated by commas:

plot(2,3,'k*',4,5,'k*',6,1,'k*')

% The 'hold on' command allows us to keep adding new points,
% lines etc. to a graph. 'hold off' tells Matlab that any
% subsequent graphing commands are to be used on a new
% graph (i.e., a new graph window produced).

hold on

% We minimize this so that we can type additional commands.

% We now learn how to connect points with lines.

% Suppose we wish to connect point (2,3) to (4,5), and point
% (4,5) to (6,1). Create a vector containing the x-values, and
% another vector containing the y-values:

x=[2 4 6]; y=[3 5 1];

% Now simply type the 'line' command:

line(x,y)       % x and y are vectors

% Check that additional graphing commands simply add on to
% the current plot:

plot(4,3,'rs')

hold off    % we are finished with the current plot

% Titles and labels for the x and y variables are easy:

xlabel('The X variable'); ylabel('The Y variable'); title('graph');

% There are many axis commands:

axis equal      % x and y scaling the same
axis square     % square plot

% Apart from graphing there are several commands that allow
% us to display messages on the screen (usually used in
% a program).

display('hello, I am your computer')

ans =
```

```
hello, I am your computer

% We can plot graphs of functions in Matlab, but we need to
% specify exactly where all the points are to be plotted. We
% use the method of creating vector function, covered in a
% previous session.

% For example, to plot the graph of y = x^2 from x = -2 to
% x = 4, with a step size of 0.1:

x = -2:0.1:4; y=x.^2; plot(x,y)

% This is a very efficient way of graphing. Conventional
% programming structures would require the use of a
% 'double loop' - see next tutorial.

% Another example:

x=linspace(-6,6,100); y=sin(x); plot(x,y)

% To print graphs use the menu on the current figure.
% Try also 'help orient'.

% We consider now the exciting world of 3-D graphing. The basic
% idea is that we create a grid of values in the x-y plane that
% represent the input values for a function of two variables
% z = f(x,y). Think of the x and y axes as the edges of a table,
% and the z values as heights above (or below) this table.

% We first create vectors to hold the possible x and y values:

x=-7.5:0.5:5; y=x;

% Now store a rectangular grid of all possible (x,y) points in
% a matrix called 'meshgrid':

[X,Y]=meshgrid(x,y);

% We now define a complicated function z = f(x,y), but instead
% of using x and y, use X and Y - the matrix equivalent of the
% scaler equation. Use the same method as for vectors to do
% element by element operations (dot . before each operation):

% Define the following quantity (eps is added to avoid the
% possibility of dividing by zero):

Z=X.^3-6*X.*Y+Y.^3;

% Z is a matrix of all 'heights' above the x-y plane, corresponding
% to the grid of (x,y) points.

% We now have several choices of 3-D graphs:

mesh(X,Y,Z)       % 3-D colored mesh
surf(X,Y,Z)       % 3-D colored surface (grid lines shown)

% A useful command takes each z value and represents the height
% by color. This is equivalent to taking the surface plot and
% projecting it down onto the x-y plane:

pcolor(X,Y,Z)

% We can remove the grid lines. This becomes a requirement when
% the grid is very fine (because there are so many together that
% they produce a dark surface):

shading interp
```

```
% There are many other ways of changing lighting, color, 'aspect'
% etc. Try 'help contour' and 'help pcolor' for example. Don't
% forget that if you are doing several things to a graph that
% you will need to use the 'hold on' and 'hold off' commands to
% apply all the options to the same matrix of Z values.

diary off
```

PROGRAMMING IN MATLAB

We now come to the main reason for using Matlab - the ability to program ('code') our own algorithms and mathematical procedures. With just a few commands we can create structures of amazing complexity and versatility. The main programming facilities are outlines here briefly. Examples are given separately.

Logical & Relational Operators

Before we look at looping & conditional statements we need to mention logical and relational symbols.

The **relational operators** are as follows:

```
<
<=
>
>=
=
~= (not equal to)
```

also, = = used with the *if-else-end* statements (see below)

The **logical operators** are as follows:

```
&     AND
|     OR
~     NOT
```

We can combine logical & relational operators, for example:

(A>2) & (A<6) true for A=3, 4, 5.

(used in the *if-else-end* statements)

Note that Matlab interprets a logical/relational expression as either **true (1)** or **false (0)**. Some trivial example:

» 1= =3 ans = 0

» (1= =1)|(1= =2) ans = 1

» (1= =1)&(1< =2) ans = 1

Using Loops in Matlab

Loops are used when we want to repeat a sequence of commands many times. This is required when evaluating a function recursively. For example, generating fractals requires the evaluation of a simple recursive function many thousands of times (the output of the function becomes the input of the function, and so we 'loop' round repeatedly).

For Loops

This is the most frequently used looping command in Matlab. The syntax is as follows:

```
for x=array
        commands . . .
end
```

The *array* is simply some vector defined via, e.g., x = 1:10. In this case the *commands* are performed 10 times.

Loop Alternatives

For simple cases we can replace a *for* loop in Matlab with a vectorized equivalent. This is quicker to write and faster for Matlab to evaluate - a real bonus when running a program that involves a heavy amount of computation. It is not unusual for programs involving the generation of complicated graphics to require the program to run over many hours, even with a fast 486 pentium processor.

e.g.,

```
for n=1:10
   calc=2*n^3;
end
```

do instead:

```
n=1:10
calc=2*n.^3;
```

- this is shorter

While Loops

Unlike the *for* loop that evaluates the *commands* a fixed number of times, the *while* loop evaluates the *commands* until some *condition* after the *while* statement is false. The syntax is as follows:

```
while condition (true)
        commands. . . .
end
```

To maximize speed

To speed up the running of a program with large arrays (vectors or matrices) we can pre-allocate them with zeros. This essentially tells Matlab how much memory we will need beforehand.

e.g., x = zeros(1,10000)

Conditional Statements

We use the *if-else-end* statement that allows the program to 'choose' different courses of action, depending on the outcome of a calculation. There are three variations of this statement. Each *condition* is a relational/logical expression (see above). Conditional statements are usually used within a loop.

Simple case:

```
if condition
        commands . . .
end
```

Case with two possible courses of action:

```
if condition
        commands evaluated if true
else
        commands evaluated if false
end
```

Case with several conditions & several possible outcomes:

```
if condition 1
        commands
elseif condition 2
        commands
elseif condition 3
        commands
        .
        .
        .
else
        commands evaluated if no
        other condition true
end
```

Note that as soon as a true expression is found, the rest of the alternatives are skipped.

The Break Command

We can use as one of our *commands* the *break* command. This causes us to prematurely leave the current loop we are in. If there are nested loops (a loop within a loop) then a *break* command takes us out of the current loop.

User Inputs

Most programs require user inputs from the keyboard, usually at the beginning of the program. The syntax is as follows:

```
variable_name=input('Enter the value of the variable_name   ')
```

When Matlab encounters this command it will halt the running of the program, display the message *'Enter the value of the variable   '*, and wait for the value to be entered.

Final Note

This brief review of Matlab has NOT covered basic programming methodology, or how to use flow diagrams to efficiently plan the overall structure of a program before coding. You should consult a standard textbook on how to do this (highly recommended).