MATLAB FOR 2.672

This is a short introduction to the MATLAB software. The idea is to enable you to use MATLAB quickly. Refer to the MATLAB REFERENCE MANUAL for more sophisticated operations.

TO USE MATLAB - click the MATLAB icon. It will set up your directory as default for you so that you can access your files.

CONCEPTS

o MATLAB is a giant calculator which remembers the values of all the variables you have calculated.
- The command you type in is executed every time you hit a carriage return (CR). Use the **up-and-down arrow keys to recall previous commands**.
- Sometimes you **do not want the intermediate results** to be printed (e.g., when you are calculating the values of a thousand numbers). To do this, **end the command with semi-colon,** before the CR.
- For a very long command that you cannot finish in one line, the line can be continued to the next by using the ellipsis consisting of **three dots...**, before the CR.

o One can store a set of commands in a script file and recall it to be used. More of this later in the section **SCRIPT FILES**. (Using a script file is preferred if you have to do any serious calculations because then you have a record of all the commands you put in, and you can edit them and redo the command sequence easily.)

o If you want to clear your "calculator," use the command **CLEAR**. You can clear a specific variable by **CLEAR variable_name**.

o **Variables in MATLAB are case sensitive!!!** If you prefer, you could adapt the convention of using lower-case characters for scalars and upper-case characters for vectors and matrices.

o MATLAB stands for MATRIX LABORATORY; therefore, the basic representation of information is in terms of matrices. A number, or scalar, is a 1x1 matrix. In 2.672, we mostly deal with vectors (e.g., an array of data). Because the software is matrix-based, there can be quite a bit of confusion if you are not careful. For example, if x is a 5x1 matrix (i.e., a column vector of 5 elements), **X*X** is illegal. On the other hand **X*X'** (**X'** stands for the transpose of X) will give you a scalar.

o You can always click the **HELP** icon to search for information within MATLAB.

BASIC OPERATIONS

o The **operators +,-,*,/,^** have the usual meaning. (The last one is to raise power.)
  - operation between a scalar (s) and a matrix (A) is straight forward; e.g. s*A means that every element of A is multiply by s.
  - For operations between matrices (vectors are special case of matrices: a row vector of n elements is a nx1 matrix; a column vector is a 1xn matrix), you have to be careful that the dimensions match, or you will get error messages.
  - Division means multiplication by the inverse matrix. As such, there are two different symbols: the right division, e.g. $A/B = A*B^{-1}$, and the left division, e.g. $C\backslash D = C^{-1}*D$.
  - In most of the applications, we want to do element-by-element operation. To do this, **precede the operator by a dot**. e.g. If X=[1,2,3] and Y=[2,4,6]. Then X*Y is illegal, but X **.***Y=[2,8,18]. Use X **.** ^2 to obtain the square of the elements of X in the above example.

o **Complex numbers** - Numbers can be complex. When you first start MATLAB, (or after a CLEAR command,) the symbols i and j are given the value of sqrt(-1). Thus you can enter complex numbers such as x=4+5*i. (In this case, because you are entering a pure number instead of an expression, you do not have to use the multiplication sign, thus x=4+5i or x= 4+5j are valid.) (Think about applications when you have to do complex transfer functions.) If you have used i and j for some other things, you can recreate them by setting i=sqrt(-1) etc.

o **To access the elements of a vector -** For vector X (does not matter whether it is a column or row vector), X(n) is the nthelement of X.

o **To access the elements of a matrix -** For matrix A, A(m,n) refers to the element in the mthrow and the nth column.

o The **range operator (:) -** It is convenient to be able to refer to a range of elements. The colon symbol does that. e.g. for a vector X, V=X(3:5) is a vector (row or column vector, corresponding to that of X) of length 3, with V(1)=X(3), V(2)=X(4), V(3)=X(5). Similarly for a matrix A, A(2,7:9) refers to the second row, and elements A(2,7), A(2,8),A(2,9).

o The extremely useful expression is the r**ange operator without range specification**. Then, it refers to the **whole range**. For example A(:,3) refers to the third column of the matrix A, and B(2,:) refers to the second row of A.

o There are a lot of other useful operations such as **looping, logical test**, etc. which are not described in here. Refer to the manual or the on-line HELP if you need to use them. To learn about looping, type **HELP FOR**; to learn about logic test, type **HELP IF**.

BUILT-IN FUNCTIONS

There are many built-in functions, such as all the standard ones (sin, cos, tan, exp, sqrt etc.) and many special ones, such as quad (to integrate a function by quadrature). In addition, we have prepared several functions for the 2.672 lab, such as **steam_p** (to evaluate the saturated thermodynamic properties of water for an input pressure) and **fitplot** (to fit a polynomial of nth degree to data and plot the data as points and the fit as line). See the MATLAB REFERENCE MANUAL for listing of built-in functions, or click the HELP icon for on-line listing. A short listing of the 2.672 built-in functions can be found at the end of this manuscript.

ENTERING NUMBERS BY HAND

o **Scalar.** Entering a **scalar** is obvious, e.g., s=5.

o **Row Vector.** To enter a **row vector**, start off with the left square bracket and end with the right square bracket. For example, X=[1,2,3,4,5] is a 1x5 vector. If it is a very long vector, use the ... line continuation method.

o **Column Vector.** To enter a **column vector**, the method is similar to the row vector, except that the elements are separated by semi-colons instead of commas. For example, Y=[2;4;6;8] is a 4x1 vector. Another way to do this is to use the transpose operator; i.e., enter as Y=[2,4,6,8]’.

o **Range Operator.** The **range operator** (:) is very useful for entering numbers. For example: X=[1:5] is the vector [1,2,3,4,5]. Even more useful, is the range with increment specification, i.e., **start:increment:end**. For example, Y=[0:0.2:1] is the vector [0,0.2,0.4,0.6,0.8,1.0]. The latter construction is very useful if you want to specify a range to calculate a function (e.g., the range of frequency to calculate the transfer function).

o **Matrix.** Entering the values of the **elements of a matrix** can best be illustrated by an example.

```
X=[1,2,3
   4,5,6]
```

is the 2x3 matrix. Alternatively, you can enter it as `X=[1,2,3;4,5,6]`, with the rows separated by the semicolon. Note that the construction is consistent with that of the row and column vectors.

IMPORTING DATA FILES FROM THE DATA ACQUISITION SYSTEM

In the data acquisition package, choose to **save the data in the matlab format**. Give it a file name, for example xyz (**the *first character of the filename should be non-numeric***; do not use any extension, because the default extension is **.m**). Then go to the MATLAB window and type in the filename as a command; in this case, it is xyz. The data will be imported. The data will also be plotted. Use the command **whos** to find out what are the data variable names.

Often you want to **pick off values from the plot**. The following command reads off n points on the plot and puts the coordinates into arrays x and y.

```
[x,y]=ginput(n)
```

You have to enter a value for n. For example, if you enter [x,y]=ginput(4), a cross-hair will appear on the figure of the plot. Use the mouse to locate the cross-hair on your curve. Clicking the mouse will enter the coordinate of the cross-hair into x and y. Repeat the process four times to finish the command and get four pair of coordinates in the arrays x and y. Note that the process does not terminate until all n points are done. (To kill the process, use ctl-c.)

To access the variables in the acquired data, use the function ***getdata*** which can be downloaded from the 2.672 web page. After you download it to your working directory, type *help getdata* to learn how to use the function.

PLOTTING

The strategy is to create the plot (using auto-scaling) first, and to rescale the axes if necessary, add titles, axes labels etc. later. The plot appears in another window called "Figure 1" (and "Figure 2" etc. if you have more than one plot). **To print the graph**, go to this window and click the **file** icon. See later for option to import the graph to Microsoft Word document.

The **plot** command creates a 2D plot of vectors or matrices. Use **plot**(X,Y) to plot vector X versus vector Y. If X or Y is a matrix, the vector is plotted versus the rows or columns of the matrix, whichever line up.

plot(Y) plots the columns of Y versus their index. If Y is complex, **plot**(Y) is equivalent to **plot**(real(Y),imag(Y)). In all other uses of **plot**, the imaginary part is ignored.

Various line types and plot symbols may be obtained with **plot**(X,Y,S) where S is a character enclosed in ' ' selecting the plot style as follows:

.	point	-	solid line
o	circle	:	dotted line
x	x-mark	-.	dash-dot line
+	plus	--	dashed line
*	star		

For example, **plot** (X,Y,'o') plots a circle symbol at each data point**,** and **plot**(X,Y,'-') connects the data points with a solid line.

plot (X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by the (X,Y,S) triples, where the X's and Y's are vectors or matrices and the S's are strings. X1,Y1,S1 are for the first plot; X2,Y2,S2 are for the second; etc.

Now that you have the plot, you can make it look nice by the following commands:

To **re-scale the axis,** use the command **axis**([xmin, xmax, ymin,ymax]). Note that the [ ] is necessary because the argument to the function AXIS is actually a 1x4 vector. You may use the functions **xlim** and **ylim** to set the x and y axis limits respectively.
To add axes labels and title, use **xlabel**('text'), **ylabel**('text') and **title**('text').

Use **gtext**('text') to add text any where on the graph. A cross-hair will show up. Use the mouse to move it to position and click.
To create a log-log plot, substitute the command **plot** (...) by **logplot** (...). Similar substitutions are used for **semilogx**, and **semilogy**.

See the manual or use the HELP icon for more advance topics such as multiple plots on a page (SUBPLOT), overlaid plots (HOLD), and freezing the axes scales etc. You can also use the edit manual to make the lines thicker etc.

FITPLOT - Many times, one has to fit a curve to data points. This procedure is done by a 2.672 built-in function.

C=**fitplot**(X,Y,n) fits a nth degree polynomial to the data, and plot the data points as circles and the fit as a solid line. The fit coefficients are put in the vector C in order of descending powers. For example C=**fitplot**(X,Y,1) would do a linear fit, with y=C(1)*x+C(2).

Often you have to look at the exponential decay coefficient of y as a function of t. The command:

```
C = fitplot(T,LOG(Y),1)
```

will do the trick. The information is then in C(1).

TO IMPORT GRAPHS TO MICROSOFT WORD DOCUMENT

If you want your graph to appear as a picture in the Word document, first create the graph in Matlab, and copy it using the edit menu. Then the graph is copied onto the clipboard as a Window metafile ready to be pasted onto a Word document.

SCRIPT FILES

These are files containing commands that you can recall to reuse again and again. These files have to have the file extension **.m.**

To create them, click the **file** icon, follow by the **new** icon, and then follow by the **M_file icon**. Then the **Matlab editor** will come up, and you can type in your commands. Afterwards hit the **file** icon; follow by the **save as** icon. You'll be asked for the filename etc. Save the file with extension **.m**. After successfully saving what you have typed in the file, the program will return to the editor. Then click the **file** icon again, and click **exit**.

To edit a previous file, click the file icon; follow by the open m file icon. Then the procedure is as in the above.

There are two kinds of script files:

The **command script files**--This is a collection of commands. To invoke them in a MATLAB session, type in the filename of the script file (without extension, because the default extension is **.m**). Then all the commands in the file will be inserted and executed.

Hints- (a) It is useful to document your script file with comments. Anything starting with a % symbol is a comment. The comment could either start at the beginning or anywhere in a line.
(b) Usually, you don't want the intermediate results to be displayed. End the commands with ; in the script file.

The **function subroutine script files -** These are files that you can call within a MATLAB command as a function. The filename in which the function subroutine is stored must be the same as the function name (i.e., **name** in the following), with the extension **.m**. The first line of these files must be:

`FUNCTION output = name(argument1, argument2, ...)`

where **output** is the name of the scalar, vector or matrix containing the output of the function evaluation, **name** is the name you give to the function, and **argument1**,... are the arguments passed down from the calling program.

Note that the names given to **output** and all the **arguments** are "dummy variable names"; i.e., you can think of them as if they take on the names of the variables used in the calling program during execution. Note also that all the internal variables are not saved and are released after the functional call.

As an example, the following is the fitplot program in **fitplot.m**.

```
function c=fitplot(x,y,n)
% fit n th degree polynomial to input vectors x,y
% fit coefficients output in c,  fit is c(1)*x^n+...+c(n)*x^n+c(n+1)
% then plot the input vector pairs as points and the fit as line
c=[];                                %Zero out output vector c
c=polyfit(x,y,n);                    %Call the built-in function polyfit to do the least   square fit
x1=min(x);                           %Get the minimum and maximum of the input array x
x2=max(x);
xout=[x1:(x2-x1)/99:x2];             %Generate the x and y values for the plot, note that these values
yout=polyval(c,xout);                %are not passed back to the calling program, but are released
                                     %after exit
plot(x,y,'o',xout,yout,'-');         %Plot the input data as points and fit values as line
```

INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS (ODE)

The steps in integrating ordinary differential equations are: (These steps will be made clearer with an example later.)

1. Rewrite the higher order ordinary differential equation as a set of first-order differential equations. This can be easily done by renaming the derivatives as other new variables. Say the variables are y1,y2,..., and the independent variable is t.
2. Write a function subroutine m file to evaluate the derivatives, i.e., dy1/dt, dy2/dt,...
3. Call the built-in ode solver **ode45** which stands for solving ode by 4th and 5th order Runge-Kutta method. The calling format is **`: [t,y] = ode45(@ydot, [t0,tfinal], y0).`**

t	is the output vector containing the independent variable (t) in the course of the integration
y	is the solution matrix. Each column corresponds to the components Y1, Y2,... The rows are the values evaluated at time points contained in **T**.
ydot	is the name of the function m file (i.e. the filename is **ydot.m**) containing the formula for evaluation of the derivatives. The ydot.m file may look like this:

```
dydt = function ydot(t,Y);
dydt(1) = expression;         %which can contain t and components of Y
dydt(2)=..... etc.            %In these expressions, dydt are the derivatives of
                              %y(1),y(2)...Note that dydt must be a column vector
```

t0	is the initial time
tfinal	is the final time (the end of the integration)
y0	is the initial condition column vector

The above procedure can best be illustrated with an example. Consider the solution to the mass/spring/damper system released from an initial position x0 from the equilibrium position. The ode is:

$$m\frac{d^2x}{dt^2}+b\frac{dx}{dt}+kx=0$$

with initial condition x(0)=x0 and dx/dt(0)=0. (Note that in solving the problem numerically, we have to put in numerical values for the parameters. Let's use m = 1, b = 2, k= 3, x0 = 4, t0 = 0, tfinal = 5 as an example. These values are purely arbitrary here. Note that in real problems, **you have to use a consistent set of units for numerical values**.) The equation can be reduced to two first order ode by defining v=dx/dt, then

```
dx/dt = v, and
dv/dt = -(bv+kx)/m
```

To use the ode function, we have to put the equations in vector form. We'll use y=[x,v]' and dy/dt=[dx/dt, dv/dt]'. (*Note that Matlab requires them to be both column vectors*.) Then we write the m file containing the derivative formulae. Let's call it **ydot.m**. (You could call it anything you like, as long as everything is consistent). **ydot.m** would look like the following:

```
function dydt=ydot(t,y);  %Note that dydt is a dummy variable to pass back the derivatives values
x=y(1);                   %It is convenient to copy the components of the vector to scalars with
v=y(2);                   %meaningful names to make it easier for typing in the formula and
                          %for debugging
m=1;
b=2;
k=3;
                          %The equation for the derivatives
dxdt=v;
dvdt=-(b*v+k*x)/m;        %In this example, t is not used
dydt=[dxdt; dvdt];        %Putting back the derivatives into the vector array of derivatives
```

Then in the MATLAB main window, use the command:

```
    [t,y] = ode45(@ydot, [0 5], [4 0]’); %[0 5] is [t0 tfinal]; [4 0]’ is [x(0) v(0)]’
```

will give you at the time points in the array t, the values of x and v in the first and second column of y.

Note that a ; is used at the end of the command to prevent displaying all the values of the variables in the course of integration.

You could plot the results of each by **plot**(t,y(:,1)) for x and **plot**(t,y(:,2)) for v. (**plot**(t,y) would plot them both in the same graph, but it is better to plot them individually using the subplot command because in general, they may have very different scales. To get drastically different values on the same graph, rescale them to new variables first)

Very often, you have to pass values to and from the ode routine. This process can be facilitated by using an anonymous function to access your derivative formulae. For instance, if you want to pass the value in the variable "mass" to your function ydot, you could do the following:

```
mass = 12;
[t,y] = ode45(@(t,y) ydot(t,y,mass), [0 5], [4 0]');

...

function dydt=ydot(t,y,m)      % ydot has been defined with an extra input parameter m
x=y(1);                   %It is convenient to copy the components of the vector to scalars with
v=y(2);                   %meaningful names to make it easier for typing in the formula and
                          %for debugging
b=2;
k=3;
                          %The equation for the derivatives
dxdt=v;
dvdt=-(b*v+k*x)/m;        %Here, the mass m was passed in as an input parameter
dydt=[dxdt; dvdt];        %Putting back the derivatives into the vector array of derivatives
```

Here, the notation **@(t,y)** indicates that the expression that follows is to be treated as a function of two variables, t and y. When MATLAB then looks at the expression **ydot(t,y,mass)**, it substitutes in the value for "mass", and passes the two-variable function **ydot(t,y,12)** to the ode routine. Consult the help entries on function handles and anonymous functions for more details.

Notes on ode45
The basis for the ‘matching’ kind of ordinary differential equation solvers such as the **ode45** is as follows. Using the above example, when the initial conditions of the above system are known, the state of the system at

every next time step, $x(t+\Delta t)$ and $v(t+\Delta t)$, can be predicted from its current state, $x(t)$, $v(t)$ and their derivatives. In this case:

$dx/dt = v(t)$
$dv/dt = a(t)$, where
$a(t)= -(b(t)*v(t)+k*x(t))/m$,

The time-marching solution process can be illustrated by a first-order-accuracy numerical scheme,

$x(t+\Delta t)=x(t)+v(t)*\Delta t$, and
$v(t+\Delta t)=v(t)+a(t)* \Delta t$.

In **ode45**, the above time-marching solution is obtained with a higher order of accuracy. It is based on the fourth- and fifth-order Runge-Kutta method. See a numerical analysis text book for detailed description of the algorithm.