

Table of Contents

Section I: How MATLAB Works

- » Basic MATLAB: The Language
- » Mfiles (Function files)
- » Dealing with Functions
 - The 'plot' function
 - The 'fzero' function
 - Solving linear equations in MATLAB
 - The 'fsolve' function

Section II: Numerical and Symbolic Integration

- » Numerical Integration; Quadrature
 - The Simpson's Rule and Lobatto Quadrature
- » Symbolic Integration and Differentiation

Section III: Numerically Solving Differential Equations and Systems of Differential Equations

- » First order ode's
- » Higher order ode's

Appendix

- » Glossary of Commands, Parts I, II and III
- » Line Markers

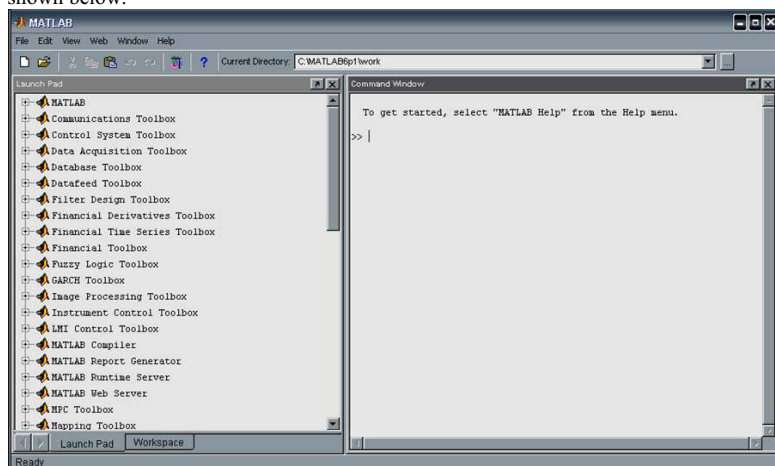
MATLAB® Applications in Chemical Engineering

James A. Carnell
North Carolina State University

MATLAB is a powerful code-based mathematical and engineering calculation program. It performs all calculations using matrices and vectors in a logical programming environment. This guide is a brief introduction to MATLAB in chemical engineering, and in no way attempts to be a comprehensive MATLAB learning resource. This guide is a starting point for the new MATLAB user, and as such no prior MATLAB experience is necessary. Further help can be found in the MATLAB help files or at Mathworks website at www.mathworks.com and in the help section at <http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml>

Section I: How MATLAB works

Most MATLAB commands can be run from the command window (shown below, on the right hand side of the interface). MATLAB commands can also be entered into a text file labeled with the '.m' extension. These files are known as 'm-files'. These commands can be broken down into scripts and programming. Scripts can be thought of as commands that instruct MATLAB to execute a particular function or pre-made program, and programming can be thought of as the raw code required to construct functions and programs within MATLAB. Generally, all programming must be contained within a file used by MATLAB (called an m-file), but script can be entered either in an m-file or directly into the command window. An image of the MATLAB interface is shown below.



MATLAB contains many ready-made programs or functions that are conveniently arranged into different toolboxes. When using MATLAB, these toolboxes and their functions can be called upon and executed in any MATLAB script. In the above image, the toolbox selection or launch pad is shown (at the left hand side of the interface).

Basic MATLAB: The language

MATLAB uses a language that is somewhat similar to that of Maple¹. The scripts or calling functions have a particular name and argument that must be entered into the function execution call. For example, to plot the sine function in MATLAB between 0 and 6 using the *fplot* command, the following code can be entered directly into the command window, or into an m-file:

```
fplot('sin(x)', [0,6])
```

¹ Maple is a registered trademark of Waterloo Maple, Inc.

(One can define the function $\sin(x)$ in an m-file and replace the *fplot* command to be *fplot('filename', [0,6])*)

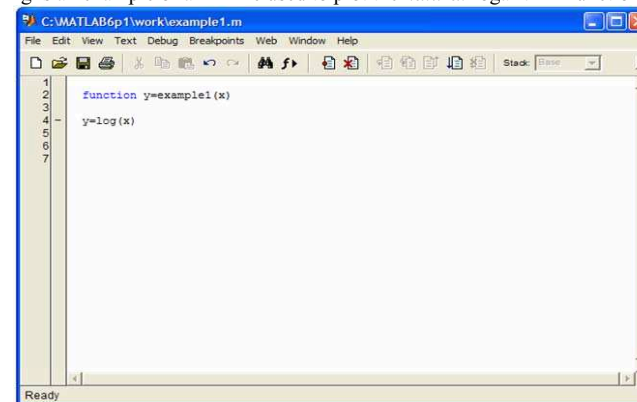
Before going much further, an understanding of the structure of a MATLAB simulation or execution must be developed.

M-files

M-files contain programming, scripts, equations or data that are called upon during an execution. If the m-file is a function definition, then the most important part of this type of m-file is the first line. The first line **must** contain the function definition so that MATLAB can find those m-files that are called upon. These types of m-files are called function m-files or function files. The code used to define the function file is as follows:

```
function z=file_name(x,y)
```

'file_name' is simply the name of the m-file (the filename *must* be the same in the definition and the file-name), z is the dependant variable, and x and y are the independent variables. (Of course, one can have less or more independent variables depending upon the complexity of the problem and the equations involved.) The next few lines of script in the m-file can define the function or functions and label any required variables. The following is an example of an m-file used to plot the *natural* logarithm² function.

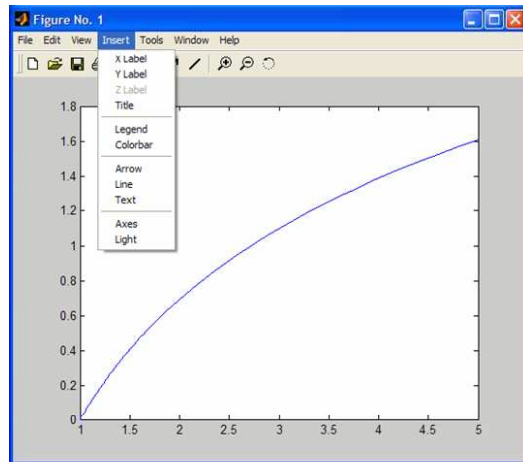


To produce a plot of this function, the following code is entered into the command window:

```
fplot('example1', [1,5])
```

This yields a plot of $\ln(x)$ between $x=1$ and $x=5$.

² MATLAB uses 'log' as the natural logarithm function, and 'log10' as logarithm base ten.



Using the 'insert' menu one can add a title, x and y axis titles, and if necessary a legend. One can also use commands nested within the *fplot* command to title the chart, add axis titles, or decide upon curve characteristics such as line color or marker.

Dealing with functions

Standard functions such as the sine, cosine, logarithmic, exponential, and user-defined functions within MATLAB will now be covered. *fplot* has already been introduced; now *plot*, *fzero* and *fsolve* will be introduced

The 'plot' function

The plot function produces a 2-D plot of a y-vector versus either its real index or a specified x-vector. The plot function can be used for data, standard functions, or user-defined functions. Let's look at a simple example of using *plot* to model data.

Example 1.1

The following reaction data has been obtained from a simple decay reaction:

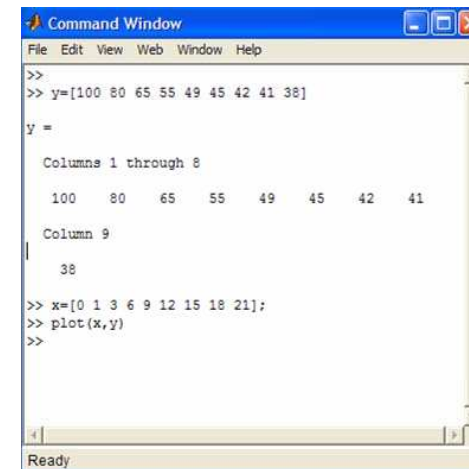


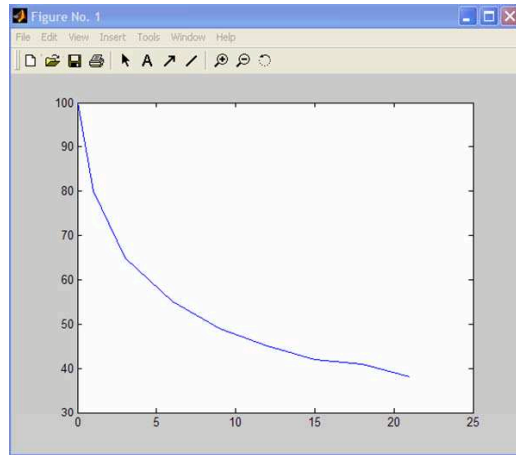
Use MATLAB to plot the concentration of component A in mol/L against the reaction time, t, in minutes. Title the plot, label the axes, and obtain elementary statistics for the data.

Time (Minutes)	Concentration (Moles/Liter)
0	100
1	80
3	65
6	55
9	49
12	45
15	42
18	41
21	38

Solution

First, the data must be entered into MATLAB as two vectors. The vectors x and y are defined in the command window, followed by the command to plot the data. The following graph is displayed:



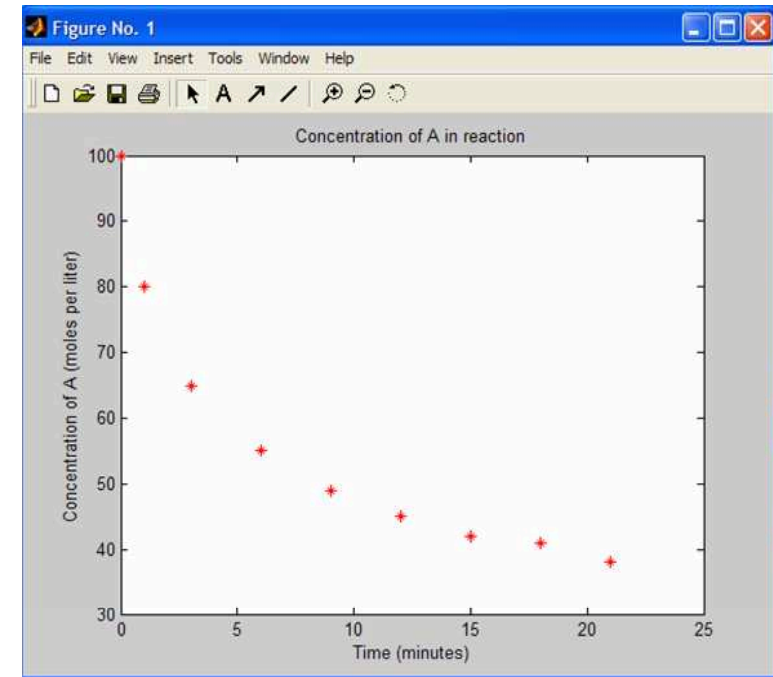


The 'x' row matrix (or vector*) has its display output suppressed using the ';' at the end of the line.

Syntax can be used to specify the title and labels, but an easier GUI (Graphical User Interface) based approach is to simply edit the figure.

- Select the 'Edit Plot' command found in the 'Tools' menu, or click the north-west facing arrow icon on the figure.
- Double click on the white-space in the graph. This enables the property editor. Now the title and axes can be inserted under the 'labels' command.
- Now click directly on the line, and the line property editor will come up. Now the lines color, pattern, or the markers can be edited. The final curve is presented below:

*A single row matrix or column matrix can represent vectors within MATLAB.



Solution curve to example 1.1

NOTE: These figures can be exported in .bmp or .jpg format so they can be made part of a document. Find these under the 'File→Export' menu.

To display simple statistics of the data, follow the path, 'Tools→Data Statistics' and the minimum, maximum, mean, median, standard deviation, and range of x and y will be displayed. Within this box, each statistic can be added to the curve as a data point/line.

To plot functions using *plot*, an m-file can be created defining the function, or the definition can be specified in the command window. This is done in a similar fashion as seen in '*fplot*'. The following code entered into the command window yields a plot of the exponential function between 0 and 10, at intervals of 1.

```
>> x = [0:10]
x =
Columns 1 through 8
    0    1    2    3    4    5    6    7
Columns 9 through 11
    8    9   10
>> y = exp(x);
>> plot(x,y)
>>
```

There are many other 2-D plotting functions in Matlab, but *fplot*, *plot* and *ezplot* are the most useful. 'ezplot' is another way to quickly plot functions between specified or non-specified ($-2\pi < x < 2\pi$) intervals. The syntax key for *ezplot* is shown below.

```
ezplot(f)
ezplot(f, [min,max])
ezplot(f, [xmin,xmax,ymin,ymax])
ezplot(x,y)
ezplot(x,y, [tmin,tmax])
```

This can be used to plot $f(x)$ or $f(x(t),y(t))$.

The 'fzero' function

The *fzero* function is a convenient function used to find local zeros of a one variable function. The general syntax that can be used for *fzero* (entered into the command window) is

```
x=fzero('function',initial guess)
```

Matlab will then return the solution or a NaN result. A NaN result is 'not a number' and represents a 'no solution' for the function. Below are a few examples relating to *fzero*.

Example 1.2

Find the zero of the function $y(x) = x^2 - 1$ using an m-file and *fzero*.

Solution

Create the m-file, 'myfunction.m'

```
function y = myfunction(x)
y = x^2-1
```

In the command window, use *fzero* to find the zero of the function 'myfunction'

```
x = fzero('myfunction',3)
```

Matlab returns a few iterations and then produces the result 'x = 1'

Example 1.3

Find the zero of the function $y(x) = \sin(x)$ closest to $x=3$ using the *fzero* command directly in the command window. Do not use an m-file for this example.

Solution

Enter into the command window `x = fzero('sin(x)',3)` Matlab returns a value of 3.1416. This is the zero of $\sin(x)$ closest to our initial guess of 3.

Solving linear equations in Matlab

Matlab has the ability to easily solve linear equations directly on the command window. The following example will demonstrate how to do this in Matlab.

Example 1.4

Solve the system of linear equations given below using Matlab.

$$\begin{aligned} 3.5u + 2v &= 5 \\ -1.5u + 2.8v + 1.9w &= -1 \\ -2.5v + 3w &= 2 \end{aligned}$$

Solution

Enter the following code directly into the clear Matlab command window. (If the window is not clear, enter the command 'clear' to clear the memory of any variables). The matrix 'a' contains the coefficients of u, v and w respectively. Each row in the matrix (there are three) corresponds to the coefficients of u, v and w in that order. Matrix 'b' contains the solution to each row or equation, 5, -1 and 2. The matrix 'x' that is divided out can be thought of as containing the values for u, v and w.

```
>> a = [3.5 2 0; -1.5 2.8 1.9; 0 -2.5 3];
>>
>> b = [5 -1 2]

b =

    5    -1    2

>> a\b'

ans =

    1.4421
   -0.0236
    0.6470
```

The solutions to the equations are displayed as u, v and w as they are listed in the matrix. The command `a\b'` performs reverse matrix division. The ' command transposes matrix b, this is required to perform matrix division. It is a property of matrices. Essentially, the

matrix is in the form $Ax=B$, the command `a\b` can be thought of as 'dividing out' the values of x . (Alternatively, the command `b/a` will also yield the correct solution set.)

The 'fsolve' function

The 'fsolve' function will probably be the most useful function for simple chemical engineering problems. It is essentially a numeric solver, capable of solving systems of non-linear *continuous* equations.

For the system of n continuous equations (f_1 -- f_n) with n unknown variables (x_1 -- x_n) given by

$$f(1) \equiv f_1(x_1, x_2, x_3, \dots, x_n) = 0$$

$$f(2) \equiv f_2(x_1, x_2, x_3, \dots, x_n) = 0$$

$$f(3) \equiv f_3(x_1, x_2, x_3, \dots, x_n) = 0$$

$$\dots\dots\dots$$

$$f(n) \equiv f_n(x_1, x_2, x_3, \dots, x_n) = 0$$

Matlab's solver can be used to determine the unknown variables, x_1 through x_n . The following example will illustrate the use of the *fsolve* function.

Example 1.5

Solve the system of equations below using *fsolve*.

$$2a - b - e^{-a} = 0$$

$$2b - a - e^{-b} = 0$$

Solution

The most efficient way to solve these types of systems is to create a function m-file that contains the equations.

```
function f=example15(x)
```

```
f = [2*x(1)-x(2)-exp(-x(1)); -x(1)+2*x(2)-exp(-x(2))];
```

The equations are separated within the matrix 'f' using the semi-colon operator. Now the 'fsolve' command must be used to solve the system.

```
>>
>> x0 = [-5 -5]; %Initial guess, arbitrary
>>
>> fsolve('example15',x0,optimset('fsolve')) %Command calls m-file to solve
Optimization terminated successfully:
  Relative function value changing by less than OPTIONS.TolFun
```

```
ans =
    0.5671    0.5671
```

The solutions to the system are $a = 0.5671$ and $b = 0.5671$. Using the 'fsolve' command, much more complicated systems can be easily solved in Matlab. NOTE: 'optimset' is used to change the default solver in Matlab to the optimization toolbox solver (fsolve, 2.0 onwards)

Section II: Numerical and Symbolic integration

Numerical Integration-Quadrature

MATLAB can perform in-depth numerical integrations or quadrature effortlessly and accurately. The first part of this section will look at some of the simple methods of numerical integration within MATLAB.

The Simpson's Rule and Lobatto Quadrature

In MATLAB, the functions 'quad' or in more recent versions, 'quadl' perform numerical integration based on the Simpson's rule and the adaptive Lobatto quadrature respectively. The syntax for the Simpson's based approximation and the Lobatto quadrature are the same. The difference is in each function's name. The syntax below is for the Lobatto quadrature (quadl), and it is the same for the Simpson's quadrature (quad).

```
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
q = quadl(fun,a,b,tol,trace)
q = quadl(fun,a,b,tol,trace,p1,p2,...)
[q,fcnt] = quadl(fun,a,b,...)
```

The functions can be defined in function m-files or as inline functions (those functions entered directly into the command). MATLAB also has a function called 'trapz' which enables numerical integration using the trapezoid rule. More information about 'trapz' can be found in the MATLAB help files.

Symbolic Integration and differentiation

Matlab has the ability to perform symbolic integration and differentiation thanks to the Maple® engine located in the symbolic toolbox. Matlab can solve differential equations symbolically, providing general or unique solutions. First, let's look at symbolic integration and differentiation.

Using the 'diff' command, symbolic differentiation of a function can be achieved, and analogously, using the 'int' command symbolic integration of a function can be achieved. Example 2.1 demonstrates use of the 'int' and 'diff' commands.

Example 2.1

Integrate ax^2+bx , with respect to x , (where a and b are constant) and then differentiate the solution obtained with respect to x to regain the initial function.

Solution

To begin with, the symbolic variables within the expression must be defined within MATLAB as symbolic variables. This is done using the `syms` command. In the

expression, a, b, and x are the symbolic variables that must be defined. The following code is used directly at the command prompt to obtain the solution:

```
>> syms a b x
>> diff(a*x^2+b*x)

ans =

2*a*x+b

>> int(2*a*x+b)

ans =

a*x^2+b*x
```

The syms command is used to define a, b and x as variables for symbolic purposes. The expression is differentiated to obtain the solution $2ax+b$, which upon integration with respect to x yields ax^2+bx as expected.

MATLAB can solve ordinary differential equations symbolically with or without boundary conditions or initial value parameters. The 'dsolve' command is used for this purpose. Within dsolve, the letter 'Dij' is used to indicate a derivative, where i is the order of differentiation, and j is the dependent variable. 'D' implicitly specifies first order derivative, 'D2' signifies a second order derivative and so on. The letter t is the default independent variable for dsolve. So D2y is analogous to $\frac{d^2y}{dt^2}$. The following example illustrates the use of dsolve.

Example 2.2

Solve the ode $\frac{d^2y}{dx} = 6y - \frac{dy}{dx}$, where y(x) using dsolve

Solution

```
>> dsolve('D2y=6*y-Dy','x')

ans =

C1*exp(-3*x)+C2*exp(2*x)
```

Where C1 and C2 are constants of integration.

The next example shows how to solve an initial value problem for a second order ode.

Example 2.3

Solve the ode $\frac{d^2m}{dx^2} + m = 0$, $m(0)=2$ and $\frac{dm}{dx}(0) = 3$, where m(x).

Solution

```
>> dsolve('D2m+m=0','m(0)=2','Dm(0)=3','x')

ans =

3*sin(x)+2*cos(x)
```

Within dsolve, m(0)=2 and Dm(0)=3 define the initial conditions of the ode.

In the assignment of the initial conditions, labeled variables could have been inserted rather than numeric values. For example, if the initial conditions for example 2.3 were $m(0)=a$ and $\frac{dm}{dx}(0) = b$, then the command chain

```
dsolve('D2m+m=0','m(0)=a','Dm(0)=b','x')
```

could have been used to yield the solution $b*\sin(x)+a*\cos(x)$.

Section III: Numerically Solving Differential Equations and Systems of Differential Equations

First order ode's

One of the best engineering uses of MATLAB is its application to the numeric solution of ordinary differential equations (ode's). MATLAB has multiple different ode solvers that allow ode's to be solved accurately and efficiently depending on the stiffness of the ode. Stiffness is the relative change in the solution of a differential equation. A stiff differential equation is one that the solution changes greatly when close to the point of integration, but need not change significantly over the duration of the integration. For this type of solution, a numerical method that takes small integration intervals rather than large intervals would be required. Stiffness and solver selection are mainly a matter of efficiency. In solving ode's, selecting a solver that takes the largest step, yet still maintains an accurate solution is the key to increased efficiency.

There are different ways to set up and execute the ode solvers, but for this guide, a system that uses multiple m-files per each ode solution will be employed. The main two m-files that are needed are a run file and a function file. For the solution of ode's in MATLAB *all* ode's *must* be defined in a function m-file. When entered into the function file, the ode's must have the first order form $\frac{dy}{dx} = f(y,x)$. The function file must contain

- i) The function definition e.g. `function dmdt=file_name(t,m)`, where t is the independent variable and m is the dependent variable of first order

- ii) If global variables are used, the global command must be inserted after the function definition
- iii) The ode or ode's in the form described above e.g. `dmdt=f(m,t)`

The filename, variables (m and t) and dmdt are arbitrary. 'dmdt' is also arbitrary and can equivalently be called 'A' or any other non-reserved name. One stipulation that exists is that the definition within the 'function `dmdt=file_name(t,m)`' MUST be the same as that defined when the ode's are listed. For example, the following function m-file is incorrect

```
function dmdt=myfile(t,m)
A=4*m
```

But, the following function file is correct:

```
function dmdt=myfile(t,m)
dmdt=4*m
```

(It may appear that from this MATLAB is unable to solve any more than a first order ode, but all ode's of second order or higher can be written in the form of multiple ode's, each of first order. This will be covered once an introduction using first order ode's has been accomplished).

Once a suitable function file has been created, a run-file or executable file is created that is used to solve the ode or ode's that are within the function file. The run file must contain the following items:

- i) If global variables are used, the global command must be inserted at this point.
- ii) Depending on how the tspan, the integration interval, is defined, the lower and upper limits can be defined here. E.g. `tspan=[t0 tf];`, where t0 and tf are predefined vectors for the integration limits. Forward or reverse integration can be used; t0 does not have to be less than tf in the case of reverse integration
- iii) The initial conditions must be defined as vectors or single direction matrices.
- iv) The ode solver must then be called. The following is the syntax for the solver, and ode45 is the solver that will be used. It is a good place to start as a general solver:

```
[independent, dependent]=ode45('file_name',tspan,initial_condition_vector)
```

- v) The solution can now be plotted using the 'plot' command and then formatted either using the GUI interface or by the commands 'legend', 'xlabel', 'ylabel', 'title' and 'axis'. E.g.

```
plot(independent, dependent(:,n),independent,dependent(:,n+1)),
```

where n denotes the 1st dependent variable, then second and so on.

The following example demonstrates the setup and execution for the solution of a set of differential equations.

Example 3.1

A fluid of constant density starts to flow into an empty and infinitely large tank at 8 L/s. A valve regulates the outlet flow to a constant 4L/s. Derive and solve the differential equation describing this process over a 100 second interval.

Solution

The accumulation is described as input – output, so the ode describing the process becomes $\frac{d(\rho V)}{dt} = (8 - 4)\rho$. Since density is constant, then $\frac{dv}{dt} = 8 - 4 = 4$ in liters per second. The initial condition is that at time t=0, the volume inside the tank =0. The following function file 'ex31' is used to set up the ode solver.

```
function dvdt=ex31(t,v)
dvdt=4
```

The file ex31run is used to execute the solver. The code for this file is overleaf.

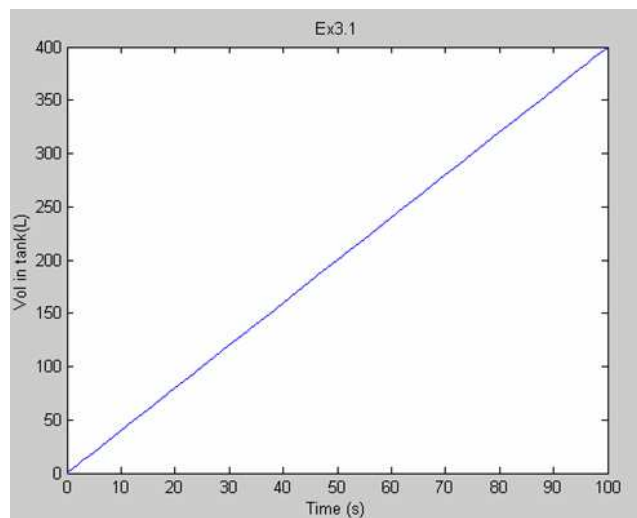
```
t0=0;
tf=100;
tspan=[t0 tf]; %Integration interval
v0=0 %Initial condition

[t,v]=ode45('ex31',tspan,v0)

plot(t,v(:,1))
xlabel('Time (s)')
ylabel('Vol in tank(L)')
title('Ex3.1')
```

The plot produced is





As expected, a constant volume increase of 4L/s is described by the curve.

Systems of ode's can also be easily solved in MATLAB using the same setup as described for a single ode. Example 3.2 demonstrated how to solve a system of simultaneous ode's. In example 3.2 the 'global' command is used to define certain variables as 'shared' or 'in common' between the run file and the function file. Global variables are not necessary, but they are convenient. Using the 'global' command variables can be defined once in the run file and the global command will link them to the function file. If used, the global command *must not* contain the independent or dependent variables.

Example 3.2

The following set of differential equations describes the change in concentration three species in a tank. The reactions $A \rightarrow B \rightarrow C$ occur within the tank. The constants k_1 , and k_2 describe the reaction rate for $A \rightarrow B$ and $B \rightarrow C$ respectively. The following ode's are obtained:

$$\frac{dCa}{dt} = -k_1Ca$$

$$\frac{dCb}{dt} = k_1Ca - k_2Cb$$

$$\frac{dCc}{dt} = k_2Cb$$

Where $k_1=1 \text{ hr}^{-1}$ and $k_2=2 \text{ hr}^{-1}$ and at time $t=0$, $Ca=5\text{mol}$ and $Cb=Cc=0\text{mol}$. Solve the system of equations and plot the change in concentration of each species over time. Select an appropriate time interval for the integration.

Solution

The following function file and run file are created to obtain the solution:

```
function dcdt=Ex32(t,c)
%c(1)=ca, c(2)=cb, c(3)=cc
global k1 k2
dcdt=[-k1*c(1); k1*c(1)-k2*c(2); k2*c(2)];
```

Ca, Cb and Cc must be defined within the same matrix, and so by calling Ca c(1), Cb c(2) and Cc as c(3), they are listed as common to matrix c.

```
clc
clf
clear
global k1 k2
k1=1;
k2=2;

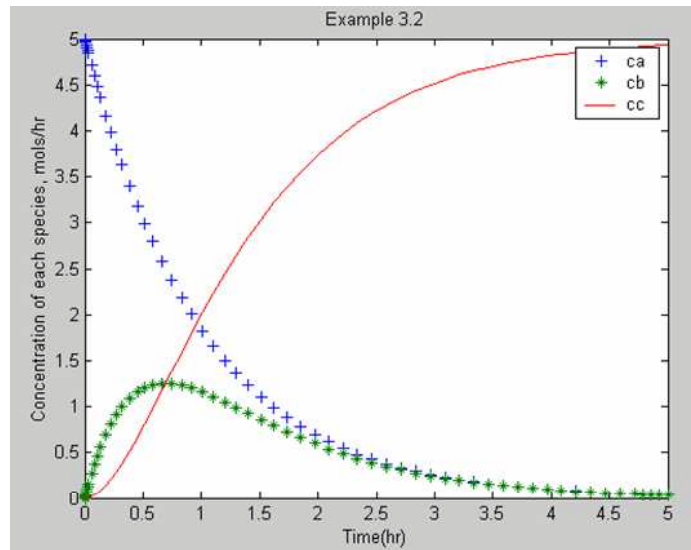
tspan=[0 5];

c0=[5 0 0];

[t,c]=ode45('Ex32',tspan,c0)

plot(t,c(:,1),'+',t,c(:,2),'*',t,c(:,3))
legend('ca','cb','cc')
xlabel('Time(hr)')
ylabel('Concentration of each species, mols/hr')
title('Example 3.2')
```

Notice that the constants k_1 and k_2 are defined (only once) in the run file, and using the 'global' command they are linked to the function file. The following curve is produced upon execution. In the 'plot' command, the + and * change the line markers so that they can be easily distinguished using a non-color printer. The appendix contains a list of available markers for plotting.



Higher order ode's

To be able to solve higher order ode's in MATLAB, they must be written in terms of a system of first order ordinary differential equations. An ordinary differential equation can be written in the form

$$\frac{d^n y}{dx^n} = f(x, y, y', y'', y''', \dots, y^{n-1})$$

and can also be written as a system of first order differential equations such that

$$y_1 = y, y_2 = y', y_3 = y'', \dots, y_n = y^{n-1}.$$

From here, the system can then be represented as an arrangement such that

$$y_1' = y_2, y_2' = y_3, \dots, y_{n-1}' = y_n, \text{ where } y_n' = f(x, y_1, y_2, \dots, y_n).$$

Example 3.3 demonstrates this technique.

Example 3.3

Solve the following differential equation by converting it to a system of first order differential equations, then using a numeric solver to solve the system. Plot the results.

$$Y'' + Y' + Y = 0, Y(0)=1 \text{ and } Y'(0)=0$$

To convert this 2nd order ode to a system of 1st order ode's, the following assignment is made:

$$Y = y_1 = y(1) \text{ and } Y' = y_2 = y(2),$$

then the ode can be written as the first-order system:

$$Y = y(1)$$

$$\frac{dY}{dt} = Y' = y(2)$$

$$\frac{d^2 Y}{dt^2} = Y'' = -(y(2) + y(1))$$

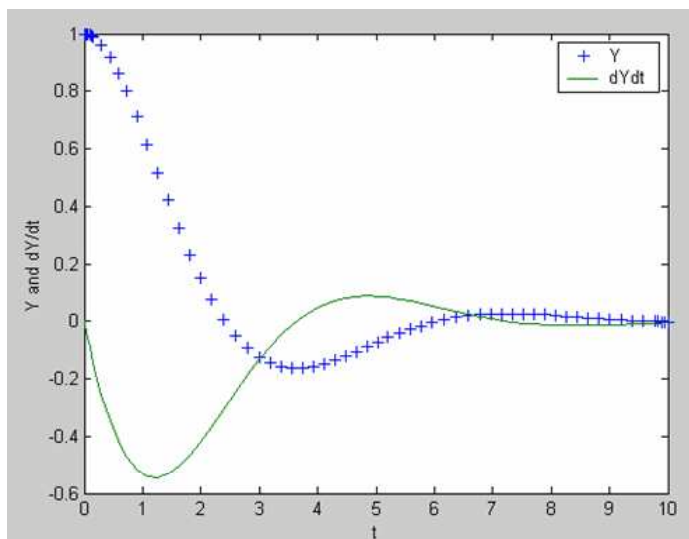
The function file containing this system can now be created and solved. The function file and run file are shown below:

```
function dmdt=Ex33(t,y)
%Solving Y'' + Y' + Y = 0
%Assign Y=y(1), Y'=y(2), Y''=y(3)
dYdt=y(2);
dY2dt=-(y(2)+y(1));
dmdt=[dYdt; dY2dt];
```

The variable dmdt is a 'dummy' variable that is used to describe the system as a whole,

```
clc
clf
clear
tspan=[0 10];
y0=[1 0];
[t,Y]=ode45('Ex33',tspan,y0)
plot(t,Y(:,1),'+',t,Y(:,2))
legend('Y','dY/dt')
xlabel('t')
ylabel('Y and dY/dt')
```

The following curve is produced



The appendices of this guide are designed to be used as 'cheat-sheets' to aid in using MATLAB without simply manipulating the examples herein. Successful use of MATLAB comes from understanding how and why it works the way it does. MATLAB can be used for almost anything that requires calculation and so frequent use for simple tasks will aid in the competency of the package.

Appendix

MATLAB Glossary of commands (Part 1)

- Matrix commands (A is an arbitrary matrix)
 - » `sum(A)` → Summation of the rows of matrix A
 - » `A'` → Transposes matrix A
 - » `diag(A)` → Takes the diagonal values of square matrix A
 - » `fliplr(A)` → Flips matrix A from left to right
 - » `1:n:10` → Fills sequence of numbers, 1 to 10 with spacing n (if n is not stipulated, then MATLAB will assume n=1)
- Command Window
 - » `clear` → Clears memory of variables
 - » `clc` → Clears command window (clf will clear the figure window)
 - » `...` → Wraps text or code if required, (followed by return or enter)
 - » `up arrow` → Recalls last line
 - » `;` → Output suppression (eg. Does not SHOW output)
 - » `iskeyword` → Displays reserved MATLAB keywords
- Functions and other commands
 - » `plot(independent, dependent)` → A graphing command. Independent variables must be defined before using the plot command.
 - » `fplot('function',[xmin, xmax, ymin, ymax])` → Graphs a function either from an m-file or direct command, the function entered can be a file-name linking to an m-file or it can be the function itself e.g. 'sin(x)'
 - » `ezplot('function',[xmin, xmax, ymin, ymax])` → As above
 - » `fsolve('m-file-name',initial_guess,options)` → Solves non-linear continuous systems of equations. Initial guess usually in the form of a matrix, predefined. Usual option is 'optimset('fsolve')'
 - » `x=fzero('function',initial_guess)` → Numerically finds the zero of the function close to the initial guess. The function can be in an m-file or a direct command (similar to fplot). The initial guess can be entered directly, or in the form of a predefined vector. The in-line function must be in terms of x only. It cannot contain any predefined constants.
 - » `m=input('m =')` → Requests that an input variable be entered into MATLAB, which is then saved as 'm' or any other arbitrary name. Used in m-files
 - » `display('text to be displayed')` → Displays 'text to be displayed' in a matlab execution. Used in m-files

In all of the above function definitions, the ' notation must be included in the MATLAB command execution. For example, the fzero command to find the zero of x^3 , one would use the notation: `x=fzero('x^3',1)`, where 1 is the initial guess.

MATLAB glossary of commands (Part II)

Symbolic integration and differentiation

- » `syms v1 v2` → Declares variable v1 and v2 as symbolic variables, used for symbolic integration and differentiation
- » `int(exp1, independent)` → Performs symbolic integration of exp1 with respect to independent variable 'independent'
- » `diff(exp1, independent)` → Performs symbolic differentiation of exp1 with respect to independent variable 'independent'
- » `dsolve('ode1,ode2,oden', 'bc1,bc2,bcn', 'IV')`
→ Performs symbolic ode solving, where ode1, ode2 and oden are systems of ode's and bc1, bc2 and bcn are their respective boundary conditions. IV is the common independent variable. The default IV is t, so if no IV is specified, then t is assumed to be the independent variable. To indicate a derivative the capital letter 'D' is used, followed by the order then the dependent variable. So D2y represents $\frac{d^2y}{d(IV)^2}$. For non-boundary condition problems, the 'bc1,bc2,bc3' term can be negated and the solution will be returned containing integration constants.
- » `simplify(A)` → Simplifies expression A

MATLAB glossary of commands (Part III)

The sequence of this sheet hints towards the correct sequence for solving odes in matlab

1. The different ODE solvers:

Solver	Solves These Kinds of Problems	Method
ode45	Nonstiff differential equations	Runge-Kutta
ode23	Nonstiff differential equations	Runge-Kutta
ode113	Nonstiff differential equations	Adams
ode15s	Stiff differential equations and DAEs	NDFs (BDFs)
ode23s	Stiff differential equations	Rosenbrock
ode23t	Moderately stiff differential equations and DAEs	Trapezoidal rule
ode23tb	Stiff differential equations	TR-BDF2

2. `global v1 v2 vi`
Defines global variables v1, v2, and vi. When required, the `global` command is used in all m files to declare common variables.
3. `tspan [to tf]`
Sets integration interval. `to` and `tf` must be defined prior to `tspan`. `tspan` is an arbitrary name for the row vector containing `to` and `tf`, which are also arbitrarily named.
4. `yo=[a0, a1, a2 ...]`
Initial conditions for each dependent variable in a 1st order differential equation. `yo` is an arbitrary name. A single 1st order DEQ requires only one initial condition, and each independent DEQ thereafter requires another.
5. `[Independent, dependentbase] = ode#('filename', tspan, y0)`
Syntax to instruct MATLAB to use ODE solver called `ode#` to solve the function defined in file called `filename`, using the initial conditions described in `y0`. "`#`" is the number for a particular solver; generally `ode45` works as an initial try.
6. `plot(t, y(:,1), t, y(:,2), t, ...)`
Invokes plot command to plot t versus variables defined in function as `y(1)`, `y(2)`, etc.
7. Plot formatting commands

```
xlabel('X axis title')
ylabel('Y axis title')
axis([xmin xmax ymin ymax])
title('Plot title')
```

8. Flow commands

Flow commands are used to change functions between different sets of specified conditions. For example, equations describing the amount of heat required by a system used to boil cold water would change once the temperature reached T_b , the boiling temperature:

```
if t<100
    Q=m*cp*(T-Tref)
else
    Q=m*heatvap
end
```

The operator 'elseif' can also be used. It acts in a similar fashion to the 'else' command except that it requires a condition to be true. For example, the following set of differential equations apply to the specified intervals

$$\frac{dm}{dt} = 4, \text{ when } m \text{ is less than or equal to } 10$$

$$\frac{dm}{dt} = 2, \text{ when } m \text{ is between } 10 \text{ and } 20$$

$$\frac{dm}{dt} = 6, \text{ when } m \text{ is greater or equal to } 20$$

In MATLAB, the 'if', 'else' and 'elseif' commands can be used to specify this system.

```
if m<=10           %If this statement is true, then
    dmdt=4         %MATLAB executes the ode dmdt=4
elseif m<20       %If the first statement is not true, I.E. we
    %are at a point where m is greater than 10,
    %then the 'elseif m<20' dictates that if m is
    %less than 20, then dmdt=2, and this is the ode
    %MATLAB executes
    dmdt=2
elseif 20<=m      %If neither of the first statements are true,
dmdt=6             %but the third 'elseif 20<=m', or if 20 is less
end               %than or equal to m, then MATLAB executes the
%               %ode dmdt=6
```

Allowable operators for the "if" statement:

>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to
==	equals
~=	does not equal

Line Markers

The following table describes the available line markers that can be used for plotting in MATLAB.

Keyboard key or symbol	Marker
+	Plus sign +
*	Multiplication sign *
o	Circle o
.	Dot .
x	Cross x
s	Square
d	Diamond
v	Downward pointing triangle
^	Upward pointing triangle
>	Right pointing triangle
<	Left pointing triangle
p	Five point star
h	Six point star