School of Engineering
Brown University

Dynamics and Vibrations
MATLAB tutorial

This tutorial is intended to provide a crash-course on using a small subset of the features of MATLAB. If you complete the whole of this tutorial, you will be able to use MATLAB to integrate equations of motion for dynamical systems, plot the results, and use MATLAB optimizers and solvers to make design decisions.

You can work step-by-step through this tutorial, or if you prefer, you can brush up on topics from the list below.

If you are working through the tutorial for the first time, you should complete sections 1-13. You can do the other sections later, when they are needed.

If you have taken EN30, you should be familiar with the material in Sections 1-10: you can review these if you like, or skip directly to Section 11.

1. What is MATLAB?

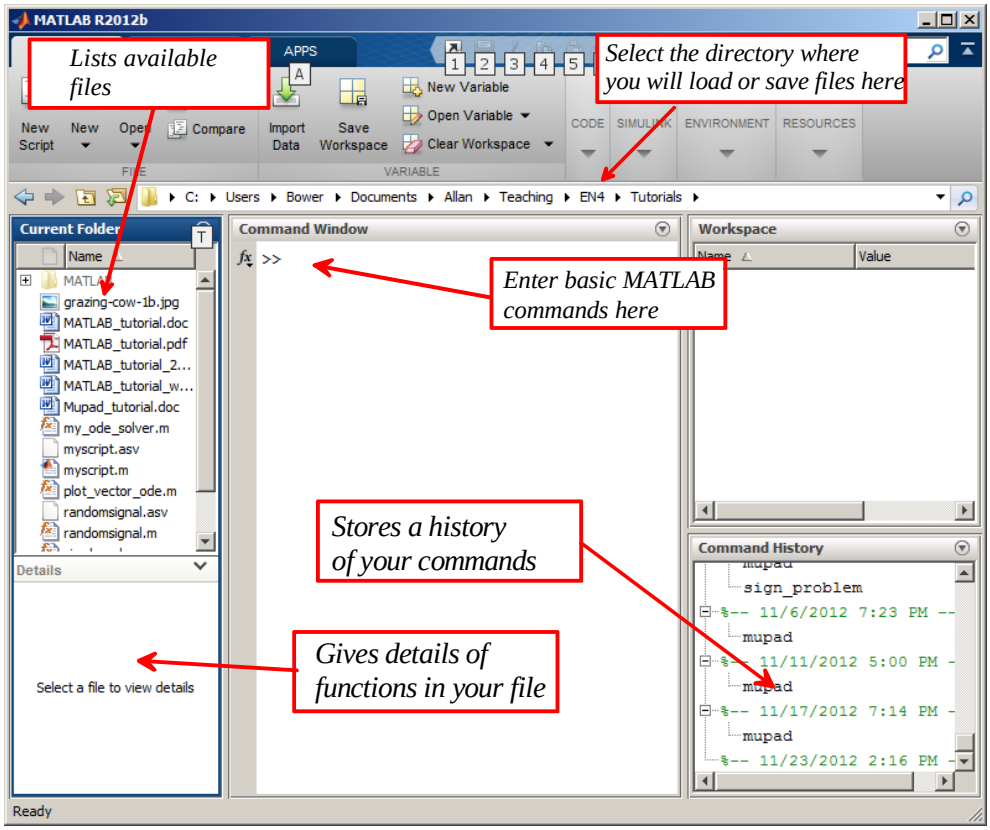
You can think of MATLAB as a sort of graphing calculator on steroids – it is designed to help you manipulate very large sets of numbers quickly and with minimal programming. MATLAB is particularly good at doing matrix operations (this is the origin of its name). It is also capable of doing symbolic computations (see the mupad tutorial for details).

2. Starting MATLAB

MATLAB is installed on the engineering instructional facility. You can find it in the Start>Programs menu. You can also install MATLAB on your own computer. This is a somewhat involved process – you need to first register your name at mathworks, then wait until they create an account for you there, then download MATLAB and activate it. Detailed instructions can be found at https://wiki.brown.edu/confluence/display/CISDOC/Matlab+Designated+Computer+Installation
The instructions tell you to wait for an email from mathworks, but they don't always send one. Check your account a day or two after you register – if the download button for MATLAB appears you are all set. If you have previously registered, you can download upgraded versions of MATLAB whenever you like. The latest release is 2012b, and it is worth downloading if you are using an older version.

3. Basic MATLAB windows

Install and start MATLAB. You should see the GUI shown below. The various windows may be positioned differently on your version of MATLAB – they are 'drag and drop' windows. You may also see a slightly different looking GUI if you are using an older version of MATLAB.



Select a convenient directory where you will be able to save your files.

4. Simple calculations using MATLAB

You can use MATLAB as a calculator. Try this for yourself, by typing the following into the command window. Press 'enter' at the end of each line.

```
>>x=4
>>y=x^2
>>z=factorial(y)
>>w=log(z)*1.e-05
>> format long
>>w
>> format long eng
>>w
>> format short
>>w
>>sin(pi)
```

MATLAB will display the solution to each step of the calculation just below the command. Do you notice anything strange about the solution given to the last step?

Once you have assigned a value to a variable, MATLAB remembers it forever. To remove a value from a variable you can use the 'clear' statement - try

```
>>clear a
>>a
```

If you type 'clear' and omit the variable, then *everything* gets cleared. Don't do that now – but it is useful when you want to start a fresh calculation.

MATLAB can handle complex numbers. Try the following

```
>>z = x + i*y
>>real(z)
>>imag(z)
>>conj(z)
>>angle(z)
>>abs(z)
```

You can even do things like

```
>> log(z)
>> sqrt(-1)
>> log(-1)
```

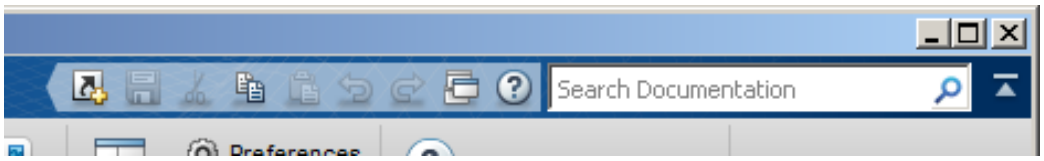
Notice that:

- Unlike MAPLE, Java, or C, you don't need to type a semicolon at the end of the line (To properly express your feelings about this, type >>load handel and then >> sound(y,Fs) in the command window).
- If you *do* put a semicolon, the operation will be completed but MATLAB will not print the result. This can be useful when you want to do a sequence of calculations.

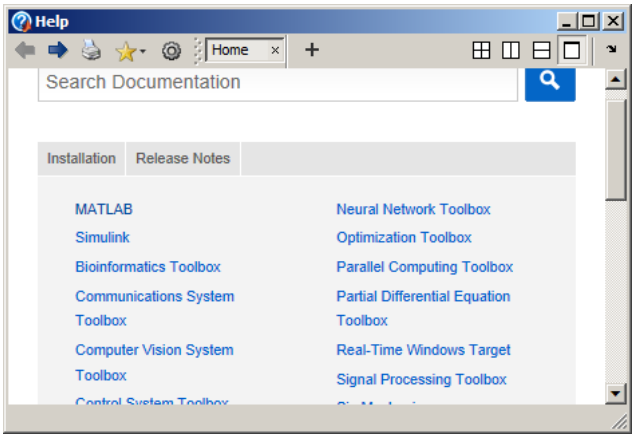
- Special numbers, like `pi' and 'i' don't need to be capitalized (Gloria in Excelsis Deo!). But beware – you often use i as a counter in loops – and then the complex number *i* gets re-assigned as a number. You can also do dumb things like pi=3.2 (You may know that in 1897 a bill was submitted to the Indiana legislature to declare pi=3.2 but fortunately the bill did not pass). You can reset these special variables to their proper definitions by using clear i or clear pi
- The Command History window keeps track of everything you have typed. You can double left click on a line in the Command history window to repeat it, or right click it to see a list of other options.
- Compared with MAPLE, the output in the command window looks like crap. MATLAB is not really supposed to be used like this. We will discuss a better approach later.
- If you screw up early on in a sequence of calculations, there is no quick way to fix your error, other than to type in the sequence of commands again. You can use the 'up arrow' key to scroll back through a sequence of commands. Again, there is a better way to use MATLAB that gets around this problem.
- If you are really embarrassed by what you typed, you can right click the command window and delete everything (but this will *not* reset variables). You can also delete lines from the Command history, by right clicking the line and selecting Delete Selection. Or you can delete the entire Command History.
- You can get help on MATLAB functions by highlighting the function, then right clicking the line and selecting Help on Selection. Try this for the sqrt(-1) line.

5. MATLAB help

Help is available through the online manual – Click on the question-mark in the strip near the top right of the window).



By default the help window opens inside the MATLAB GUI, but you can drag it out so it occupies a new window on your desktop.



If you already know the name of the MATLAB function you want to use the help manual is quite good – you can just enter the name of the function in the search, and a page with a good number of examples usually comes up. It is more challenging to find out how to do something, but most of the functions you need can be found by clicking on the MATLAB link on the main page and then following the menus that come up on subsequent pages.

6. Errors associated with floating point arithmetic (and a basic loop)

If you have not already done so, use MATLAB to calculate

```
>>sin(pi)
```

The answer, of course, should be zero, but MATLAB returns a small, but finite, number. This is because MATLAB (and any other program) stores floating point numbers as sequences of binary digits with a finite length. Obviously, it is impossible to store the exact value of π in this way.

More surprisingly, perhaps, it is not possible even to store a simple decimal number like 0.1 as a finite number of binary digits. Try typing the following simple MATLAB program into the command window

```
>> a = 0;
>> for n =1:10   a = a + 0.1;  end
>> a
>> a – 1
```

Here, the line that reads "for n=1:10 a= a + 0.1; end" is called a "loop." This is a very common operation in most computer programs. It can be interpreted as the command: "for each of the discrete values of the integer variable n between 1 and 10 (inclusive), calculate the variable "a" by adding +0.1 to the previous value of "a" The loop starts with the value n=1 and ends with n=10. Loops can be used to repeat calculations many times – we will see lots more examples in later parts of the tutorial.

Thus, the for… end loop therefore adds 0.1 to the variable a ten times. It gives an answer that is approximately 1. But when you compute a-1, you don't end up with zero.

Of course -1.1102e-016 is not a big error compared to 1, and this kind of accuracy is good enough for government work. But if someone subtracted 1.1102e-016 from your bank account every time a financial transaction occurred around the world, you would burn up your bank account pretty fast. Perhaps even faster than you do by paying your tuition bills.

You can minimize errors caused by floating point arithmetic by careful programming, but you can't eliminate them altogether. As a user of MATLAB they are mostly out of your control, but you need to know that they exist, and try to check the accuracy of your computations as carefully as possible.

7. Vectors in MATLAB

MATLAB can do all vector operations completely painlessly. Try the following commands

```
>> a = [6,3,4]
>> a(1)
>> a(2)
>> a(3)
>> b = [3,1,-6]
>> c = a + b
>> c = dot(a,b)
>> c = cross(a,b)
```

Calculate the magnitude of c (you should be able to do this with a dot product. MATLAB also has a built-in function called `norm' that calculates the magnitude of a vector)

A vector in MATLAB need not be three dimensional. For example, try

```
>>a = [9,8,7,6,5,4,3,2,1]
>>b = [1,2,3,4,5,6,7,8,9]
```

You can add, subtract, and evaluate the dot product of vectors that are not 3D, but you can't take a cross product. Try the following

```
>> a + b
>> dot(a,b)
>>cross(a,b)
```

In MATLAB, vectors can be stored as either a *row* of numbers, or a *column* of numbers. So you could also enter the vector a as

```
>>a = [9;8;7;6;5;4;3;2;1]
```

to produce a *column* vector.

You can turn a row vector into a column vector, and vice-versa by

```
>> b = transpose(b)
```

A few more very useful vector tricks are:

- You can create a vector containing regularly spaced data points very quickly with a loop. Try

  ```
  >> for   i=1:11   v(i)=0.1*(i-1);   end
  >> v
  ```

  The for…end loop repeats the calculation with each value of i from 1 to 11. Here, the "counter" variable i now is used to refer to the i^{th} entry in the vector v, and also is used in the formula itself.
- As another example, suppose you want to create a vector v of 101 equally spaced points, starting at 3 and ending at 2*pi, you would use

  ```
  >> for   i=1:101   v(i)= 3 + (2*pi-3)*(i-1)/100;   end
  >> v
  ```
- If you type

  ```
  >> sin(v)
  ```

  MATLAB will compute the sin of every number stored in the vector v and return the result as another vector. This is useful for plots – see section 11.
- You have to be careful to distinguish between operations on a vector (or matrix, see later) and operations on the *components* of the vector. For example, try typing

  ```
  >> v^2
  ```

  This will cause MATLAB to bomb, because the square of a row vector is not defined. But you can type

  ```
  >> v.^2
  ```

  (there is a period after the v, and no space). This squares every element within v. You can also do things like

  ```
  >> v. /(1+v)
  ```

  (see if you can work out what this does).
- I personally avoid using the dot notation – mostly because it makes code hard to read. Instead, I generally do operations on vector elements using loops. For example, instead of writing w = v.^2, I would use

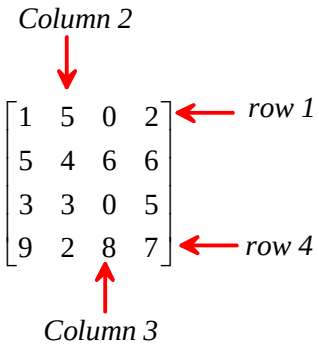
  ```
  >> for i=1:length(v) w(i) = v(i)^2; end
  ```

  Here, 'for i=1:length(v)' repeats the calculation for every element in the vector v. The function length(*vector*) determines how many components the vector v has (101 in this case). Using loops is not elegant programming, and slows down MATLAB. Purists (like CS40 TAs) object to it. But I don't care. For any seriously computationally intensive calculations I would use a serious

programming language like C or Fortran95, not MATLAB. Programming languages like Java, C++, or Python are better if you need to work with complicated data structures.

8. Matrices in MATLAB

Hopefully you know what a matrix is… If not, it doesn't matter - for now, it is enough to know that a matrix is a set of numbers, arranged in rows and columns, as shown below



A matrix need not necessarily have the same numbers of rows as columns, but most of the matrices we will encounter in this course do. A matrix of this kind is called *square*. (My generation used to call our professors and parents square too, but with hindsight it is hard to see why. 'Lumpy' would have been more accurate).

You can create a matrix in MATLAB by entering the numbers one row at a time, separated by semicolons, as follows

```
>> A = [1,5,0,2; 5,4,6,6;3,3,0,5;9,2,8,7]
```

You can extract the numbers from the matrix using the convention A(*row #, col #*). Try

```
>>A(1,3)
>>A(3,1)
```

You can also assign values of individual array elements

```
>>A(1,3)=1000
```

There are some short-cuts for creating special matrices. Try the following

```
>>B = ones(1,4)
>>C = pascal(6)
>>D = eye(4,4)
>>E = zeros(3,3)
```

The 'eye' command creates the 'identity matrix' – this is the matrix version of the number 1. You can use

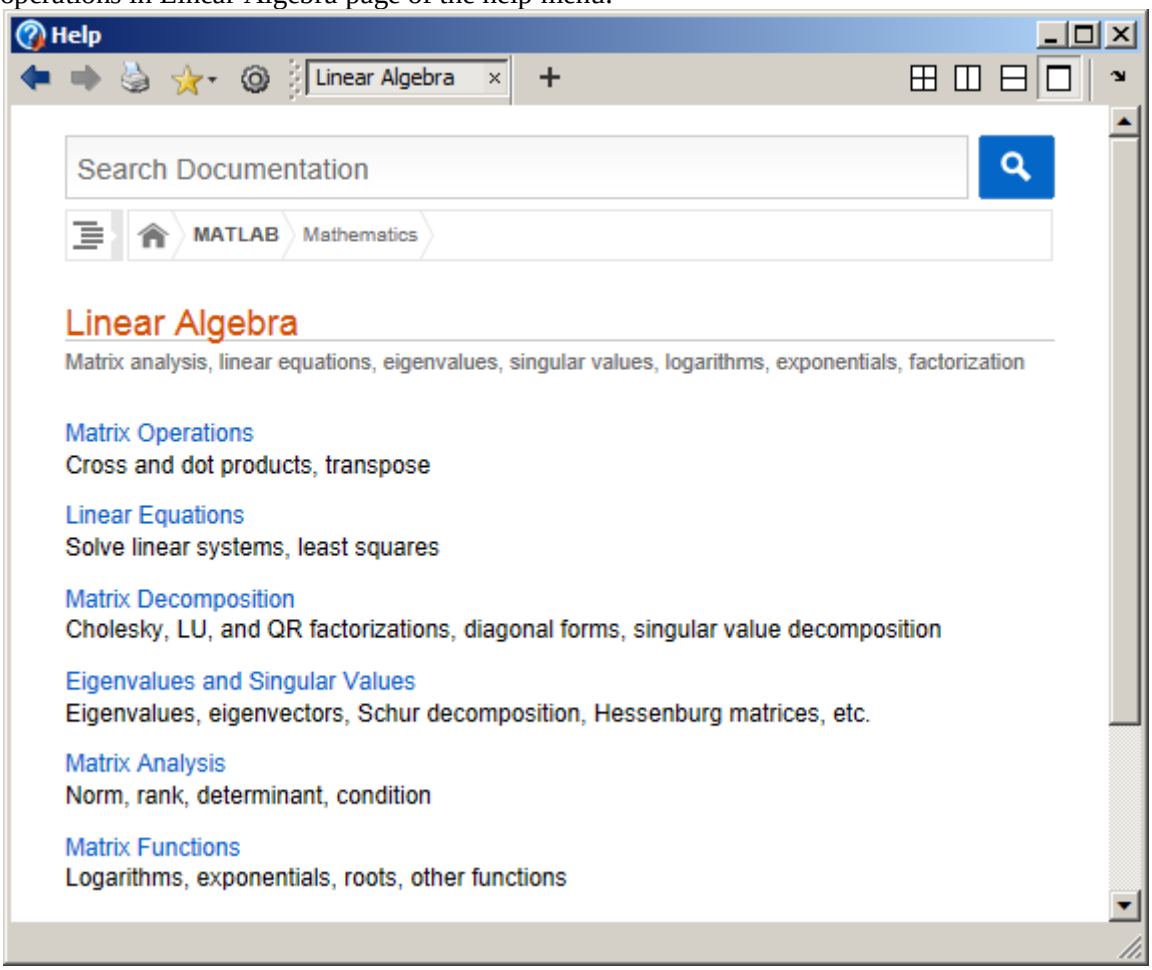
```
>> help pascal
```

to find out what pascal does.

MATLAB can help you do all sorts of things to matrices, if you are the sort of person that enjoys doing things to matrices. For example

1. You can flip rows and columns with >> B = transpose(A)

2. You can add matrices (provided they have the same number of rows and columns >> C=A+B
   Try also >> C – transpose(C)
   A matrix that is equal to its transpose is called *symmetric*
3. You can multiply matrices – this is a rather complicated operation, which will be discussed in more detail elsewhere. But in MATLAB you need only to type >>D=A*B to find the product of A and B. Also try the following
   >> E=A*B-B*A
   >> F = eye(4,4)*A - A
4. You can do titillating things like calculate the determinant of a matrix; the inverse of a matrix, the eigenvalues and eigenvectors of a matrix. If you want to try these things
   >> det(A)
   >> inv(A)
   >> eig(A)
   >> [W,D] = eig(A)
   You can find out more about these functions, and also get a full list of MATLAB matrix operations in Linear Algebra page of the help menu.



MATLAB can also calculate the product of a matrix and a vector. This operation is used very frequently in engineering calculations. For example, you can multiply a 3D *column* vector by a matrix with 3 rows and 3 columns, as follows

>>v = [4;3;6]
>>A = [3,1,9;2,10,4;6,8,2]

```
>>w=A*v
```

The result is a 3D column vector. Notice that you can't multiply a 3D row vector by a 3x3 matrix. Try this

```
>>v = [4,3,6]
>>w=A*v
```

If you accidentally enter a vector as a row, you can convert it into a column vector by using
>>v = transpose(v)

MATLAB is also very good at solving systems of linear equations. For example, consider the equations

$$3x_1 + 4x_2 + 7x_3 = 6$$
$$5x_1 + 2x_2 - 9x_3 = 1$$
$$-x_1 + 13x_2 + 3x_3 = 8$$

This system of equations can be expressed in matrix form as

$$\mathbf{Ax} = \mathbf{b}$$

$$\mathbf{A} = \begin{bmatrix} 3 & 4 & 7 \\ 5 & 2 & -9 \\ -1 & 13 & 3 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 6 \\ 1 \\ 8 \end{bmatrix}$$

To solve these in MATLAB, you would simply type

```
>> A = [3,4,7;5,2,-9;-1,13,3]
>> b = [6;1;8]
>> x = A\b
```

(note the forward-slash, not the back-slash or divide sign) You can check your answer by calculating

```
>> A*x
```

The notation here is supposed to tell you that x is b 'divided' by A – although `division' by a matrix has to be interpreted rather carefully. Try also

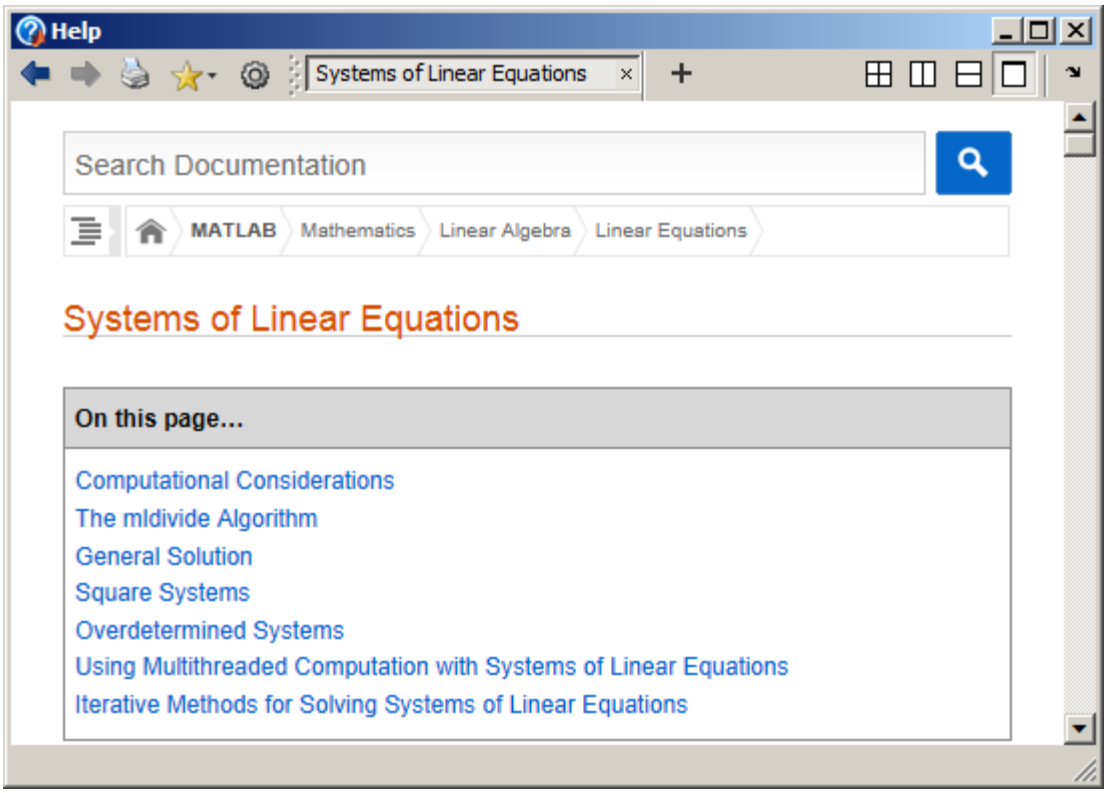
```
>>x=transpose(b)/A
```

The notation transpose(b)/A solves the equations $\mathbf{xA} = \mathbf{b}$, where **x** and **b** are row vectors. Again, you can check this with

```
>>x*A
```

(The answer should equal **b**, (as a row vector) of course)

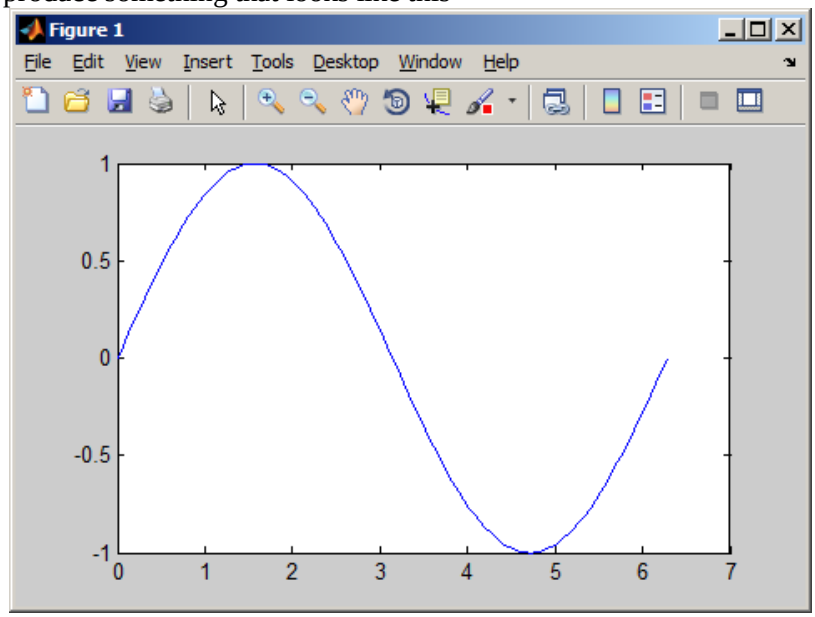
MATLAB can quickly solve huge systems of equations, which makes it useful for many engineering calculations. The feature has to be used carefully because systems of equations may not have a solution, or may have many solutions – MATLAB has procedures for dealing with both these situations but if you don't know what it's doing you can get yourself into trouble. For more info on linear equations check the section of the manual below



9. Plotting and graphics in MATLAB

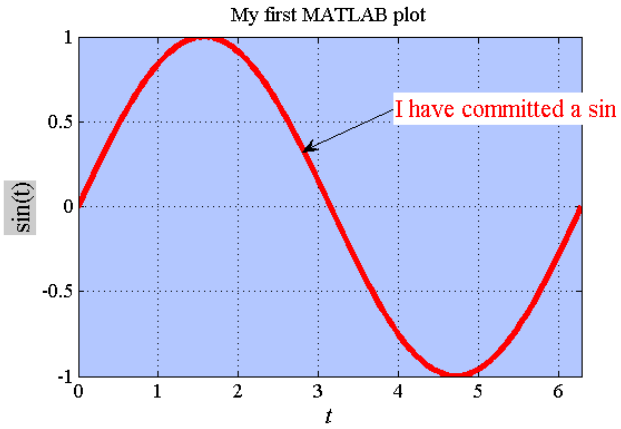
Plotting data in MATLAB is very simple. Try the following
>> for i=1:101 x(i)=2*pi*(i-1)/100; end
>> y = sin(x)
>> plot(x,y)

MATLAB should produce something that looks like this



MATLAB lets you edit and annotate a graph directly from the window. For example, you can go to Tools> Edit Plot, then double-click the plot. A menu should open up that will allow you to add x and y axis labels, change the range of the x-y axes; add a title to the plot, and so on.

You can change axis label fonts, the line thickness and color, the background, and so on – usually by double-clicking what you want to change and then using the pop-up editor. You can export figures in many different formats from the File> Save As menu – and you can also print your plot directly. Play with the figure for yourself to see what you can do.



Helpful Hint: If you annotate a plot by hand, you can make MATLAB generate the code to re-create your annotated plot automatically by going to File> Generate Code. This is useful when you write your own scripts and want to label plots automatically.

To plot multiple lines on the same plot you can use
>> y = sin(x)
>> plot(x,y)
>> hold on
>> y = sin(2*x)
>> plot(x,y)
Alternatively, you can use
>> y(1,:) = sin(x);
>> y(2,:) = sin(2*x);
>> y(3,:) = sin(3*x);
>> plot(x,y)
Here, **y** is a matrix. The notation y(1,:) fills the first row of **y**, y(2,:) fills the second, and so on. The colon : ensures that the number of columns is equal to the number of terms in the vector **x**. If you prefer, you could accomplish the same calculation in a loop:
>> for i=1:length(x) y(1,i) = sin(x(i)); y(2,i) = sin(2*x(i)); y(3,i) = sin(3*x(i)); end
>> plot(x,y)

Notice that when you make a new plot, it always wipes out the old one. If you want to create a new plot without over-writing old ones, you can use
>> figure
>> plot(x,y)

The 'figure' command will open a new window and will assign a new number to it (in this case, figure 2). If you want to go back and re-plot an earlier figure you can use

```
>> figure(1)
>> plot(x,y)
```

If you like, you can display multiple plots in the same figure, by typing

```
>> newaxes = axes;
>> plot(x,y)
```

The new plot appears over the top of the old one, but you can drag it away by clicking on the arrow tool and then clicking on any axis or border of new plot. You can also re-size the plots in the figure window to display them side by side. The statement 'newaxes = axes' gives a name (or 'handle') to the new axes, so you can select them again later. For example, if you were to create a third set of axes

```
>> yetmoreaxes = axes;
>> plot(x,y)
```

you can then go back and re-plot `newaxes' by typing

```
>> axes(newaxes);
>> plot(x,y)
```

Doing parametric plots is easy. For example, try

```
>> for i=1:101 t(i) = 2*pi*(i-1)/100; end
>> x = sin(t);
>> y = cos(t);
>> plot(x,y)
```

MATLAB has vast numbers of different 2D and 3D plots. For example, to draw a filled contour plot of the function $z = \sin(2\pi x)\sin(2\pi y)$ for $0 \le x \le 1, \quad 0 \le y \le 1$, you can use

```
>> for i =1:51 x(i) = (i-1)/50; y(i)=x(i); end
>> z = transpose(sin(2*pi*y))*sin(2*pi*x);
>> figure
>> contourf(x,y,z)
```

The first two lines of this sequence should be familiar: they create row vectors of equally spaced points. The third needs some explanation – this operation constructs a matrix z, whose rows and columns satisfy z(i,j) = sin(2*pi*y(i)) * sin(2*pi*x(j)) for each value of x(i) and y(j).
If you like, you can change the number of contour levels

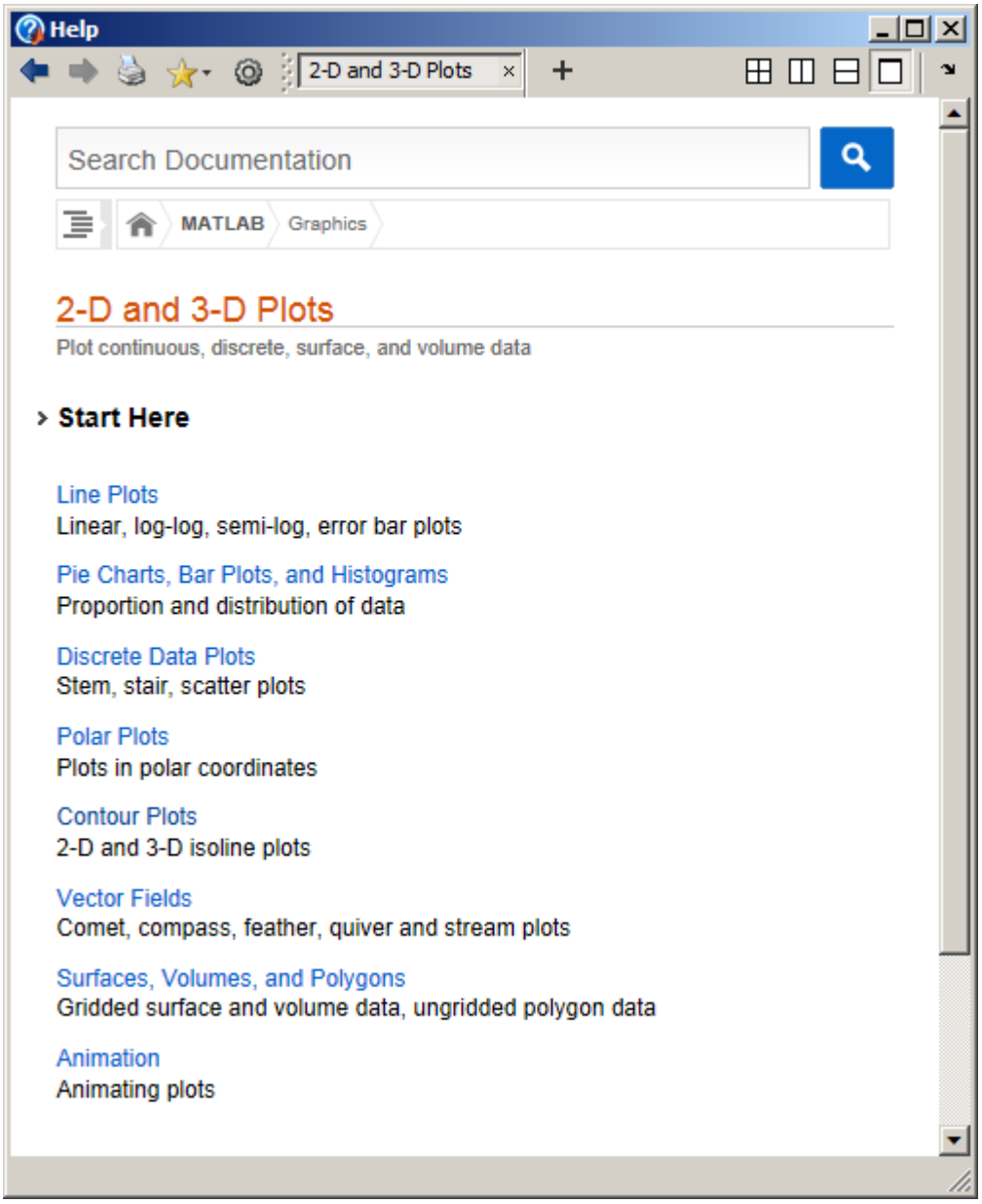
```
>>contourf(x,y,z,15)
```

You can also plot this data as a 3D surface using

```
>> surface(x,y,z)
```

The result will look a bit strange, but you can click on the 'rotation 3D' button (the little box with a circular arrow around it) near the top of the figure window, and then rotate the view in the figure with your mouse to make it look more sensible.

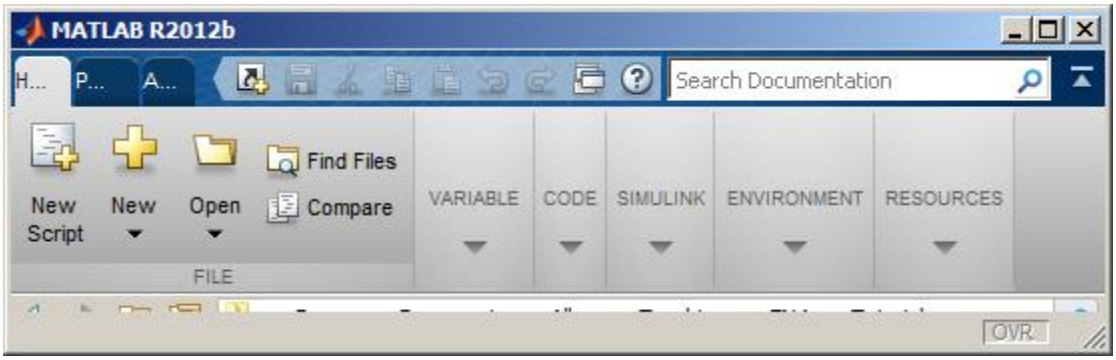
You can find out more about the different kinds of MATLAB plots in the section of the manual shown below



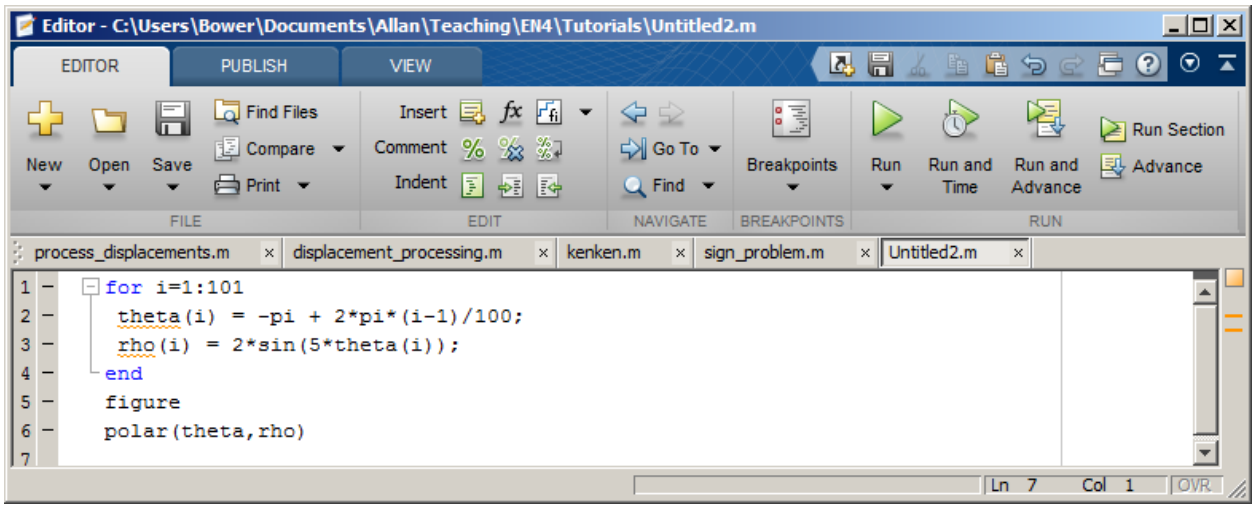
10. Working with M files

So far, we've run MATLAB by typing into the command window. This is OK for simple calculations, but it is usually better to type the list of commands you plan to execute into a file (called an M-file), and then tell MATLAB to run all the commands together. One benefit of doing this is that you can fix errors easily by editing and re-evaluating the file. But more importantly, you can use M-files to write simple programs and functions using MATLAB.

To create an M-file, simply press the 'New Script' button on the main MATLAB window.



This will open a new window for the matlab script editor, as shown below



Now, type the following lines into the editor (see the screenshot):

```
for i=1:101
 theta(i) = -pi + 2*pi*(i-1)/100;
 rho(i) = 2*sin(5*theta(i));
end
figure
polar(theta,rho)
```

You can make MATLAB execute these statements by:

1. Pressing the green arrow labeled 'Run' near the top of the editor window – this will first save the file (MATLAB will prompt you for a file name – save the script in a file called myscript.m), and will then execute the file.
2. You can save the file yourself (e.g. in a file called myscript.m). You can then run the script from the command window, by typing

>> myscript

You don't have to use the MATLAB editor to create M-files – any text editor will work. As long as you save the file in plain text (ascii) format in a file with a .m extension, MATLAB will be able to find and execute the file. To do this, you must open the directory containing the file with MATLAB (e.g. by entering the path in the field at the top of the window). Then, typing

>> *filename*

in the command window will execute the file. Usually it's more convenient to use the MATLAB editor - but if you happen to be stuck somewhere without access to MATLAB this is a useful alternative. (Then again, this means you can't use lack of access to MATLAB to avoid doing homework, so maybe there is a down-side)

11. MATLAB functions

You can also use M-files to define new MATLAB *functions* – these are programs that can accept user-defined data and use it to produce a new result. For example, to create a function that computes the magnitude of a vector:

1. Open a new M-file with the editor (press the 'New' button on the editor)
2. Type the following into the M-file

```
function y = magnitude(v)
% Function to compute the magnitude of a vector
y = sqrt(dot(v,v));
end
```

   Note that MATLAB ignores any lines that start with a % - this is to allow you to type comments into your programs that will help you, or users of your programs, to understand them.
3. Save the file (accept the default file name, which is magnitude.m)
4. Type the following into the command window

```
>> x =  [1,2,3];
>> magnitude(x)
```

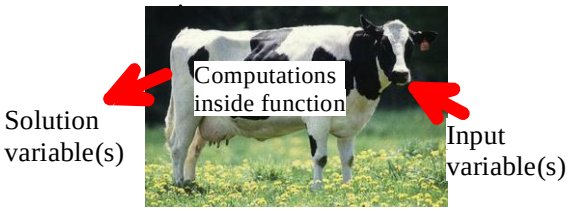
Note the syntax for defining a function – it must always have the form

function *solution_variable = function_name*(*input_variables…)*

…script that computes the function, ending with the command:

solution_variable = ….
end

You can visualize the way a function works using the picture. Think of the 'input variables' as grass (the stuff cows eat, that is, not the other kind of grass); the body of the script as the cow's many stomachs, and the solution variable as what comes out of the other end of the cow.



A cow can eat many different vegetables, and you can put what comes out of the cow in many different containers. A function is the same: you can pass different variables or numbers into a function, and you can assign the output to any variable you like – try >> result1 = magnitude([3,4]) or >> result2 = magnitude([2,4,6,8]) for example.

If you are writing functions for other people to use, it is a good idea to provide some help for the function, and some error checking to stop users doing stupid things. MATLAB treats the comment lines that appear just under the function name in a function as help, so if you type

>> help magnitude

into the command window, MATLAB will tell you what `magnitude' does. You could also edit the program a bit to make sure that v is a vector, as follows

```
function y = magnitude(v)
% Function to compute the magnitude of a vector

% Check that v is a vector
[m,n] = size(v);   % [m,n] are the number of rows and columns of v
% The next line checks if both m and n are >1, in which case v is
% a matrix, or if both m=1 and n=1, in which case v scalar; if either,
% an error is returned
if ( ( (m > 1) & (n > 1)  ) | (m == 1 & n == 1) )
    error('Input must be a vector')
end
y = sqrt(dot(v,v));
end
```

This program shows another important programming concept. The "if……end" set of commands is called a "conditional" statement. You use it like this:

```
if (a test)
        calculation to do if the test is true
end
```

In this case the calculation is only done if *test* is true – otherwise the calculation is skipped altogether. You can also do

```
if (a test)
        calculation to do if the test is true
else
        calculation to do if the test is false
end
```

In the example `(m > 1) & (n > 1)` means "m>1 and n>1" while `| (m == 1 & n == 1)` means "or m=1 and n=1". The symbols & and | are shorthand for `and' and 'or'.

Functions can accept or return data as numbers, arrays, strings, or even more abstract data types. For example, the program below is used by many engineers as a substitute for social skills

```
function s = pickup(person)
% function to say hello to someone cute
s = ['Hello ' person ' you are cute']

beep;
end
```

(If you try to cut and paste this function into MATLAB the quotation marks will probably come out wrong in the Matlab file and the function won't work. The quotes are all 'close single quote' on your keyboard – you will need to correct them by hand.)

You can try this out with

>> pickup('Janet')

(Janet happens to be my wife's name. If you are also called Janet please don't attach any significance to this example)

You can also pass *functions* into other functions. For example, create an M-file with this program

```
function plotit(func_to_plot,xmin,xmax,npoints)
% plot a graph of a function of a single variable f(x)
% from xmin to xmax, with npoints data points
for n=1:npoints
    x(n) = xmin + (xmax-xmin)*(n-1)/(npoints-1);
```

```
    v(n) = func_to_plot(x(n));
end
figure;
plot(x,v);
end
```

Then save the M-file, and try plotting a cosine, by using

```
>> plotit(@cos,0,4*pi,100)
```

Several new concepts have been introduced here – firstly, notice that variables (like func_to_plot in this example) don't necessarily have to contain numbers, or even strings. They can be more complicated things called "objects." We won't discuss objects and object oriented programming here, but the example shows that a variable can contain a function. To see this, try the following

```
>> v = @exp
>> v(1)
```

This assigns the exponential function to a variable called v – which then behaves just like the exponential function. The '@' before the exp is called a 'function handle' – it tells MATLAB that the function exp(x), should be assigned to v, instead of just a variable called exp.

The '@' in front of cos "plotit(@cos,0,4*pi,100)" assigns the cosine function to the variable called `func_to_plot` inside the function.

Although MATLAB is not really intended to be a programming language, you can use it to write some quite complicated code, and it is widely used by engineers for this purpose. CS4 will give you a good introduction to MATLAB programming. But if you want to write a real program you should use a genuine programming language, like C, C++, Java, lisp, etc.

IMPORTANT: You may have got used to writing scripts in EN30 without defining them as functions. ***For the calculations you will be doing in EN40, it will be essential to make all your scripts define a function***. This means you must start the script with 'function *name of function*' and terminate it with an 'end' statement. An example is shown below.

```
function  My_Design_Project
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here


end
```

If you like, you can automatically create a new function by using the New>Function menu, (You will usually be able to delete the input and output arguments for the function).

12. Organizing complicated calculations as functions in an M file

It can be very helpful to divide a long and complicated calculation into a sequence of smaller ones, which are done by separate functions in an M file. As a very simple example, the M file shown below creates plots a pseudo-random function of time, then computes the root-mean-square of the signal. Copy and paste the example into a new MATLAB M-file, and press the green arrow to see what it does.

```
function randomsignal
 % Function to plot a random signal and compute its RMS.
```

```
close all % This closes any open figure windows


npoints = 100; % No. points in the plot
dtime = 0.01;  % Time interval between points

% Compute vectors of time and the value of the function.
% This example shows how a function can calculate several
% things at the same time.
[time,function_value] = create_random_function(npoints,dtime);

% Compute the rms value of the function
rms = compute_rms(time,function_value);

% Plot the function
plot(time,function_value);
% Write the rms value as a label on the plot
label = strcat('rms signal = ',num2str(rms));
annotation('textbox','String',{label},'FontSize',16,...
    'BackgroundColor',[1 1 1],...
    'Position',[0.3499 0.6924 0.3944 0.1]);

end
%
function [t,y] = create_random_function(npoints,time_interval)
% Create a vector of equally spaced times t(i), and
% a vector y(i), of random values with i=1…npoints
  for i = 1:npoints
    t(i) = time_interval*(i-1);
% The rand function computes a random number between 0 and 1
    y(i) = rand-0.5;
  end

end

function r = compute_rms(t,y)
%  Function to compute the rms value of a function y of time t.
%  t and y must both be vectors of the same length.
   for i = 1:length(y) %You could avoid this loop with . notation
      ysquared(i) = y(i)*y(i);
   end
%  The trapz function uses a trapezoidal integration
%  method to compute the integral of a function.
   integral = trapz(t,ysquared);

   r = sqrt(integral/t(length(t))); %  This is the rms.

end
```

Some remarks on this example:

1. Note that the m-file is organized as

   main function – this does the whole calculation
   result 1 = function1(data)
   result2 = function2(data)
   end of main function
   Definition of function 1
   Definition of function2

   When you press the green arrow to execute the M file, MATLAB will execute all the statements that appear in the *main function*. ***The statements in function1 and function2 will only be executed if the function is used inside the main function; otherwise they just sit there waiting to be asked to do something***. I remember doing much the same thing at high-school dances.

2. When functions follow each other in sequence, as shown in this example, then variables defined in one function are invisible to all the other functions. For example, the variable 'integral' only has a value inside the function compute_rms. It does not have a value in create_random_function.

13. Solving differential equations with MATLAB

The main application of MATLAB in EN40 is to analyze motion of an engineering system. To do this, you always need to solve a differential equation. MATLAB has powerful numerical methods to solve differential equations. They are best illustrated by means of examples.

Before you work through this section it is very important that you are comfortable with the way a MATLAB function works. Remember, a MATLAB function takes some input variables, uses them to do some calculations, and then returns some output variables.

13.1 What is a differential equation?

Differential equations are mathematical models used by engineers, physicists, mathematical biologists, and even economists. For example, crude economic models say that the rate of growth of the GDP g of a country is proportional to its GDP, giving the differential equation

$$\frac{dg}{dt} = kg(t)$$

This is an equation for the unknown function *g(t)*. If you know the value of g at time *t=0* (eg $g(0) = g_0$), the equation can be solved to see that $g(t) = g_0 \exp(kt)$.

Thus

- A differential equation is an algebraic equation for an unknown function of a single variable (e.g. *g(t)*. The equation involves derivatives of the function with respect to its argument.
- To solve a differential equation, you need to know *initial conditions* or *boundary conditions* that specify the value of the function (and/or its time derivatives) at some known instant.

There are many different kinds of differential equation – your math courses will discuss them in more detail. In this course, we will usually want to solve differential equations that describe the motion of some mechanical system. We will often need to solve for more than one variable at the same time. For example, the *x,y,z* coordinates of a satellite orbiting a planet satisfy the equations

$$\frac{d^2x}{dt^2} = -\frac{kx}{\left(x^2+y^2+z^2\right)^{3/2}} \qquad \frac{d^2y}{dt^2} = -\frac{ky}{\left(x^2+y^2+z^2\right)^{3/2}} \qquad \frac{d^2z}{dt^2} = -\frac{kz}{\left(x^2+y^2+z^2\right)^{3/2}}$$

where *k* is a constant. We will see many other examples. All the differential equations we solve will come from Newtons law **F**=m**a** (or rotational equations for rigid bodies), and so are generally *second order* differential equations (they contain the second derivative of position with respect to time).

13.2 Solving a basic differential equation.

Suppose we want to solve the following equation for the unknown function *y(t)* (where *t* is time)

$$\frac{dy}{dt} = -10y + \sin(t)$$

with initial conditions $y = 0$ at time *t=0*. We are interested in calculating *y* as a function of time – say from time *t=0* to *t=20*. We won't actually compute a formula for *y* – instead, we calculate the *value* of *y* at a series of different times in this interval, and then plot the result.

We would do this as follows

1. Create a function (in an *m-file*) that calculates $\frac{dy}{dt}$, given values of *y* and *t*. Create an *m-file* as follows, and save it in a file called 'simple_ode.m'

```
function dydt = simple_ode(t,y)
% Example of a MATLAB differential equation function

   dydt = -10*y + sin(t);
end
```

   It's important to understand what this function is doing. Remember, (*t,y)* are the 'input variables' to the function. 'dydt' is the 'output variable'. So, if you type *value* = simple_ode(*some value of t, some value of y*) the function will compute the value of dydt for the corresponding (*t,y*), and put the answer in a variable called 'value'. To explore what the function does, try typing the following into the MATLAB command window.

```
>> simple_ode(0,0)
>>  simple_ode(0,1)
>>  simple_ode(pi/2,0)
```

   For these three examples, the function returns the value of *dy / dt* for *(t=0,y=0); (t=0,y=1)* and *(t=pi,y=0)*, respectively. Do the calculation in your head to check that the function is working.

2. Now, you can tell MATLAB to solve the differential equation for y and plot the solution. To do this, type the following commands into the MATLAB command window.

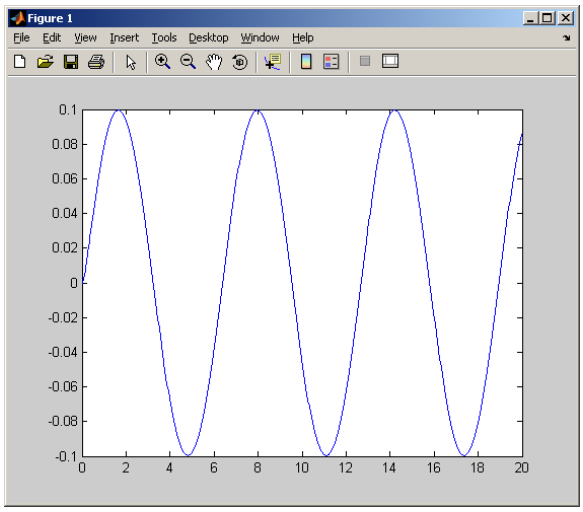
```
>> [t_values,y_values] = ode45(@simple_ode,[0,20],0);
>> plot(t_values,y_values)
```

Here,

ode45(*@function name,*[start time, end time], *initial value of variable y*)

is a special MATLAB function that will integrate the differential equation (numerically). Note that a `function handle' (the @) has to be used to tell MATLAB the name of the function to be integrated.

Your graph should look like the plot shown below



Easy? Absolutely! But there is a great deal of sophisticated math and programming hidden in the built-in numerical functions of MATLAB.

Let's look a bit more carefully at how the ODE solver works. Think about the command

```
>> [t_values,y_values] = ode45(@simple_ode,[0,20],0);
```

This is calling a function called 'ode45'. Like all functions, ode45 takes some input variables, digests them, and produces some output variables. Here, the input variables are

1. The name of the function that computes the time derivative (simple_ode). It must be preceded by an @ to tell MATLAB that this variable is a function. You must always write this function yourself, and it must always have the form

```
function dydt = name_of_function(t,y)
% MATLAB differential equation function
%   Do some calculations here to compute the value of dydt
   dydt = …
end
```

2. The time interval for which you would like to calculate *y*, specified as [*start time*, *end time*].
3. The value of *y* at *start time*.

Now let's look at the 'output variables' from ode45. The output variables are two *vectors* called t_values, y_values. (Of course you can give these different names if you like). To see what's in the vectors, type

```
>> t_values
>> y_values
```

The vector 't_values' contains a set of time values. The vector 'y_values' contains the value of 'y' at each of these times. So when we typed plot(t_values,y_values) we got a graph of *y* as a function of *t* (look back at how MATLAB plots work if you need to).

It's important to understand that MATLAB has not actually given you a formula for y as a function of time. Instead, it has given you *numbers* for the value of the solution *y* at a set of different times.

HEALTH WARNING: People find the way MATLAB is using the 'simple_ode' function hard to understand – spend some time going through this section and the next to make sure you are comfortable with it.

Let's work through a second example, but this time write a MATLAB 'm' file to plot the solution to a differential equation. This is the way we normally use MATLAB – you can solve most problems by just modifying the code below. As an example, let's solve the Bernoulli differential equation

$$\frac{dy}{dt} = \frac{1}{6}\left(ty^4 + 2y\right)$$

with initial condition $y = -2$ at time $t=0$. Here's an 'm' file to do this

```
function solve_bernoulli
% Function to solve dy/dt = (t*y^4+2y)/6
% from t=tstart to t=tstop, with y(0) = initial_y
% Define parameter values below.
    tstart = 0;
    tstop = 20;
    initial_y = -2;

   [t_values,sol_values] = ode45(@diff_eq,[tstart,tstop],initial_y);

   plot(t_values,sol_values);

   function dydt = diff_eq(t,y) % Function defining the ODE
      dydt = (t*y^4+2*y)/6;
   end

end
```

Notice how the code's been written. We created a function to compute the value of *dy/dt* given a value for *t* and *y*. This is the function called 'diff_eq'. Notice that although the function comes *after* the line that computes the solution to the ODE (the line with 'ode45' in it), that doesn't matter. When MATLAB executes your 'm' file, it first scans through the file and looks for all the functions – these are then stored internally and can be evaluated at any time in the code. It's common to put functions at or near the end of an 'm' file to make it more readable.

The solution to the ODE is computed on the line

```
[t_values,sol_values] = ode45(@diff_eq,[tstart,tstop],initial_y);
```

Notice that the function handle `@diff_eq` is used to tell MATLAB which function to use to compute the value of *dy/dt*. As before, MATLAB computes a series of time values, and a series of *y* values, which are stored in the vectors t_values, sol_values. These are then plotted.

The main reason that people find this procedure hard to follow is that it's difficult to see how MATLAB is using your differential equation function – this is buried inside MATLAB code that you can't see. The next section tries to show how this works.

13.3 How the ODE solver works

It is helpful to have a rough idea of how MATLAB solves a differential equation. To do this, we'll first come up with a simple (but not very good) way to integrate an arbitrary differential equation. Then we'll code our own version of MATLAB's ode45() function.

Let's take another look at the first differential equation discussed in the preceding section. We would like to find a function *y(t)* that satisfies

$$\frac{dy}{dt} = -10y + \sin(t)$$

with $y = 0$ at time *t=0*. We won't attempt to find an algebraic formula for *y* – instead, we will just compute the value of *y* at a series of different times between $t = 0$ and $t = 20$ - say $t = 0, t = 0.1, t = 0.2...$ To see how to do this, remember the definition of a derivative

$$\frac{dy}{dt} = \lim_{\Delta t \to 0} \frac{y(t + \Delta t) - y(t)}{\Delta t}$$

So if we know the values of *y* and dy / dt at some time *t*, we can use this formula backwards to calculate the value of *y* at a slightly later time $t + \Delta t$

$$y(t + \Delta t) = y(t) + \Delta t \frac{dy}{dt}$$

This procedure can be repeated to find all the values of *y* we need. We can do the calculation in a small loop. Try copying the function below in a new M-file, and run it to see what it does.

```
function simple_ode_solution
% The initial values of y and t
y(1) = 0;
t(1) = 0;
delta_t = 0.1;
n_steps = 20/delta_t;
for i = 1:n_steps
    dydt = -10*y(i) + sin(t(i)); % Compute the derivative
    y(i+1) = y(i) + delta_t*dydt; % Find y at time t+delta_t
    t(i+1) = t(i) + delta_t; % Store the time
end

plot(t,y);

end
```

This function is quite complicated, so let's look at how it works. Start by noticing that *t* and *y* are two vectors, which contain values of time *t* and the corresponding value of *y* at each time. The function just computes and plots these vectors – it accomplishes the same task as ode45 in the previous section.

Now let's take a look at how the vectors are calculated. We know that at time *t=0*, *y* is zero. So we put these in the first entry in the vector: y(1) = 0; t(1) = 0. Next, we want to fill in the rest of the vector. We first need to calculate how long the vector will be: the line `delta_t = 0.1` chooses the time interval Δt, and then `n_steps = 20/delta_t` calculates how many time intervals are required to reach *t=20*sec. Finally, look at the 'for i=1:nsteps … end' loop. The calculation starts with i=1. For this value of i, the program first calculates the value of dydt, using the values of *y(1)* and *t(1)*. Then it uses dydt to calculate the value of *y(2)*, and calculates the corresponding value of *t(2)*. The calculation is repeated with i=2 – this defines the value of *y(3)* and *t(3)*. This procedure is repeated until all the values in the vectors t and y have been assembled.

Now, we can use this idea to write a more general function that behaves like the MATLAB ode45() function for solving an arbitrary differential equation. Cut and paste the code below into a new M-file, and then save it in a file called my_ode45.

```
function [t,y] = my_ode45(ode_function,time_int,initial_y)

  y(1) = initial_y;
  t(1) = time_int(1);
  n_steps = 200;
  delta_t = (time_int(2)-time_int(1))/n_steps;
  for i = 1:n_steps
     dydt = ode_function(t(i),y(i));% Compute the derivative
     y(i+1) = y(i) + delta_t*dydt;% Find y at time t+delta_t
     t(i+1) = t(i) + delta_t;% Store the time
  end
end
```

You can now use this function just like the built-in MATLAB ODE solver. Try it by typing

```
>> [t_values,y_values] = my_ode45(@simple_ode,[0,20],0);
>> plot(t_values,y_values)
```

into the MATLAB command window (if you get an error, make sure your simple_ode function defined in the preceding section is stored in the same directory as the function my_ode45).

Of course, the MATLAB equation solver is actually much more sophisticated than this simple code, but it is based on the same idea.

13.4 Solving a differential equation with adjustable parameters.

For design purposes, we often want to solve equations which include design parameters. For example, we might want to solve

$$\frac{dy}{dt} = -ky + F\sin(\omega t)$$

with different values of k, F and ω, so we can explore how the system behaves. It would be nice to be able to pass these parameters to the function `simple_ode', for example by using

```
function dydt = simple_ode(t,y,k,F,omega)
% Example of a MATLAB differential equation function
   dydt = -k*y + F*sin(omega*t);
end
```

but this ***doesn't work*** – don't bother trying. Instead, you have to make the function simple_ode, a *nested* function within an M-file, as follows. Open a new M-file, and type (or cut and paste) the following example

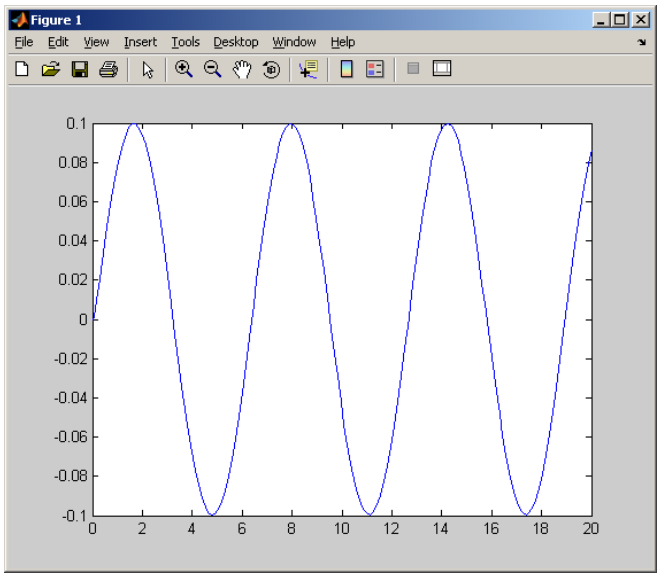
```
function solve_my_ode
% Function to solve dy/dt = -k*y + F*sin(omega*t)
% from t=tstart to t=tstop, with y(0) = initial_y

close all

% Define parameter values below.
    k = 10;
    F = 1;
    omega = 1;
    tstart = 0;
    tstop = 20;
    initial_y = 0;

   [t_values,sol_values] = ode45(@diff_eq,[tstart,tstop],initial_y);
```

```
    plot(t_values,sol_values);

    function dydt = diff_eq(t,y) % Function defining the ODE
       dydt = -k*y + F*sin(omega*t);
    end

end % Here is the end of the 'solve my ode' function
```

The function 'diff_eq' is said to be *nested* within the function 'solve_my_ode' because it appears before the 'end' statement that terminates the 'solve_my_ode' function. Because the function is nested, the variables (k,F,omega,tstart,tstop), which are given values in the _solve_my_ode function, have the same values inside the diff_equation function. Note that variable values are *only* shared by nested functions, not by functions that are defined in sequence.

Now you can solve the ODE by executing the M-file. You can get solutions with different parameter values quickly by editing the M-file.

Your graph should look like this



HEALTH WARNING: forgetting about the behavior of variables inside nested functions is (for me at least) one of the most common ways to screw up a MATLAB script, and one of the hardest bugs to find. For example, create your first MATLAB bug like this

```
function stupid
    for i=1:10
      vector(i) = i
    end
    for i=1:10
        result(i) = double_something(vector(i));
    end

    result

    function y = double_something(x)
```

```
            i=2*x;
            y = i;
        end
    end
```

Run this program to see what happens. The moron who wrote the program intended to multiply every element in the vector by two and store the answer in a vector of the same length called 'result.' This is not what happens, because the counter 'i' in the loop was modified inside the function. If you get an error message from MATLAB that says that a vector has the wrong length, it is probably caused by something like this.

13.5 Solving simultaneous differential equations

Most problems in dynamics involve more complicated differential equations than those described in the preceding section. Usually, we need to solve several differential equations at once – for example, if we are interested in 3D motion, we would need to solve for 3 (*x*,*y*,z) components of position, or velocity, at the same time. Fortunately MATLAB knows how to do this. As a simple example, let's write a function to solve and plot the solution for the following pair of coupled ODEs

$$\frac{dv_x}{dt} = -cv_x\sqrt{v_x^2+v_y^2} \qquad \frac{dv_y}{dt} = -9.81 - cv_y\sqrt{v_x^2+v_y^2}$$

with initial conditions $v_x = V_0\cos\theta$ $v_y = V_0\sin\theta$, at time t=0, where c, V_0 and θ are parameters. (These happen to be the differential equations that govern the x and y components of velocity of a projectile, accounting for the effects of air resistance). Before proceeding, make sure you are clear in your mind what is being calculated – we want to find two functions of time $v_x(t), v_y(t)$ that satisfy the two differential equations. As before, we won't compute the functions algebraically. Instead, we will calculate *values* for $v_x(t), v_y(t)$, at a sequence of successive times, and then plot the results.

To do this, we must first define a function that specifies the differential equations. This function must calculate values for $dv_x/dt, dv_y/dt$ given values of v_x, v_y and time. MATLAB can do this if we re-write the equations we are trying to solve as a ***vector***, which looks like this

$$\frac{d}{dt}\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} -cv_x\sqrt{v_x^2+v_y^2} \\ -9.81 - cv_y\sqrt{v_x^2+v_y^2} \end{bmatrix}$$

This is an equation that has the general form

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t,\mathbf{y})$$

This looks like the function we were working with in sections 15.1 and 15.2, except that now we must solve the equation for a vector **y** instead of a scalar.

The following function will calculate the vector dydt given a time value and the y vector value.

```
function dydt_vector = ode_function(t,y_vector)
    vx = y_vector(1);
    vy = y_vector(2);
    vmag = sqrt(vx^2+vy^2);
    dvxdt = -c*vx*vmag;
    dvydt = -9.81-c*vy*vmag;
    dydt_vector = [dvxdt;dvydt]; % Note the ; so dydt is a column vector
end
```

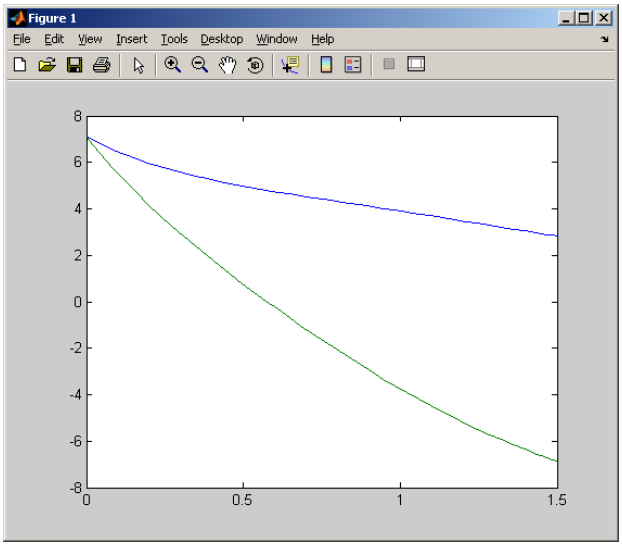
To understand how this works, note that when MATLAB is solving several ODEs, it stores the variables as components of a vector. In this example, the variables v_x, v_y are stored in a column vector y_vector, with components $y\_vector(1) = v_x$, $y\_vector(2) = v_y$. Similarly, the variables $dv_x/dt, dv_y/dt$ are stored in a vector dydt_vector, with components $dydt\_vector(1) = dv_x/dt$ and $dydt\_vector(2) = dv_y/dt$. So the function (i) extracts values for v_x, v_y out of the vector y_vector; then (ii) uses these to compute values for $dv_x/dt, dv_y/dt$; then (iii) sets up the column vector dydt_vector.

HEALTH WARNING: Note that the variable dydt_vector computed by ode_function ***must be a column vector*** (that's why semicolons appear between the dvxdt and the dvydt in the definition of dydt_vector). If you forget to do this, you will get a very confusing error message out of MATLAB.

Now, to solve the ODEs, create an M-file containing this function as shown below.

```
function solve_coupled_odes
% Function to solve a vector valued ODE
close all
V0 = 10;
theta = 45;
c=0.1;
tstart = 0;
tstop = 1.5;
theta = theta*pi/180; % convert theta to radians

%compute initial values for the two velocity components
initial_y_vector = [V0*cos(theta);V0*sin(theta)];
% IF you like you can also specify the initial condition as a row
% eg using [V0*cos(theta),V0*sin(theta)] - MATLAB will accept both.

%Here is the call for the ODE solver
[times,sols] = ode45(@ode_function,[tstart,tstop],initial_y_vector);

plot(times,sols);

      %here is the (vector) function to be integrated
      function dydt_vector = ode_function(t,y_vector)
         vx = y_vector(1);
         vy = y_vector(2);
         vmag = sqrt(vx^2+vy^2);
         dvxdt = -c*vx*vmag;
         dvydt = -9.81-c*vy*vmag;
         dydt_vector = [dvxdt;dvydt];
      end

end
```

Now execute the M file. The result should look like the plot shown below. The two lines correspond to the solution for v_x, v_y. They are both plotted as functions of time.



We also need to take a closer look at the way MATLAB gives you the solution. When we solved a *scalar* equation, the solution was a vector of time values, together with a vector of solution values. When you solve a vector equation, the variable 'times' still stores a bunch of time values where the solution has been computed. But now MATLAB has computed a bunch of *vectors* at every time. How is the result stored?

The answer is that the variable 'sols' is a *matrix*. To see what's in the matrix, suppose that solution values v_{x1}, v_{y1} v_{x2}, v_{y2} v_{x3}, v_{y3}..... were computed at a bunch of different times t_1, t_2, t_3.... The variables 'times' and 'sols' then contain a vector and a matrix that look like this

$$times = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ \vdots \end{bmatrix} \qquad sols = \begin{bmatrix} v_{x1} & v_{y1} \\ v_{x2} & v_{y2} \\ v_{x3} & v_{y3} \\ v_{x4} & v_{y4} \\ \vdots & \vdots \end{bmatrix}$$

Each row in the matrix 'sol' corresponds to the solution at a particular time; each column in the matrix corresponds to a different component for the vector valued solution. For example, the solution for v_x at time times(i) can be extracted as sols(i,1), while the solution for v_y is sols(i,2).

Understanding this lets us plot the solution in many different ways:

- To plot only v_x as a function of time use `plot(times,sols(:,1))`
- To plot only v_y as a function of time use `plot(times,sols(:,2))`
- To plot v_x on the *x* axis –v- v_y on the *y* axis use `plot(sols(:,1),sols(:,2))`

In each case, the ':' tells MATLAB to use *all* the rows in the matrix for the plot.

13.6 Solving higher order differential equations

The equations we solve in dynamics usually involve *accelerations*, and so don't have the form $d\mathbf{y}/dt = f(t,\mathbf{y})$ required by MATLAB. Fortunately, a simple trick can be used to convert an equation involving accelerations into this form. As an example, suppose that we would like to plot the trajectory of a projectile as it flies through the air, accounting for the effects of air resistance. The equations for the *(x,y)* coordinates of the projectile are

$$\frac{d^2x}{dt^2} = -c|v|\frac{dx}{dt} \qquad \frac{d^2y}{dt^2} = -g - c|v|\frac{dy}{dt}$$

where *c* is the specific drag coefficient for the projectile ,*g* is the gravitational acceleration, and

$$|v| = \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

is the magnitude of the velocity of the projectile. The initial conditions for the equation are

$$x = y = 0 \qquad \frac{dx}{dt} = V_0\cos\theta \qquad \frac{dy}{dt} = V_0\sin\theta$$

at time *t=0*. We would like to solve these equations for (*x,y*), but MATLAB can't handle the equations in this form. To convert this into the correct form for MATLAB, we simply introduce new variables to denote dx/dt and dy/dt

$$\frac{dx}{dt} = v_x \qquad \frac{dy}{dt} = v_y$$

and note that

$$\frac{d^2x}{dt^2} = \frac{dv_x}{dt} \qquad \frac{d^2y}{dt^2} = \frac{dv_y}{dt}$$

With these substitutions, our equations of motion become

$$\frac{dv_x}{dt} = -c|v|v_x \qquad \frac{dv_y}{dt} = -g - c|v|v_y$$

We now have four unknowns $\mathbf{y} = [x, y, v_x, v_y]$, and can write the equation of motion as

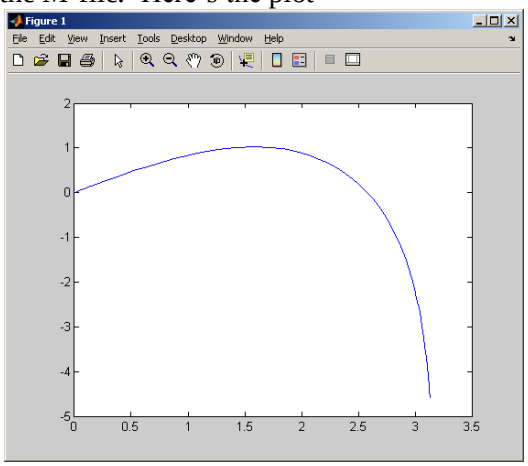
$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ -c|v|v_x \\ -g - c|v|v_y \end{bmatrix}$$

This is now in standard MATLAB form, and you can solve it by making a small change to the `plot_vector_ode` function, as follows

```
function plot_vector_ode
% Function to calculate and plot trajectory of a projectile
close all
V0 = 10;
theta = 45;
c=0.5;
tstart = 0;
tstop = 1.5;
theta = theta*pi/180; % convert theta to radians

% w0 contains initial values for all four variables
% stored as [x,y,vx,vy]
w0 = [0;0;V0*cos(theta);V0*sin(theta)];
```

```
% solve the ode for the output vector w as a function of time t:
[times,sols] = ode45(@odefunc,[tstart,tstop],w0);

% The solution is a matrix. Each row of the matrix contains
% [x,y,vx,vy] at a particular time.  The next line plots
% x as a function of y.
plot(sols(:,1),sols(:,2));

function dwdt = odefunc(t,w)
% The vector w contains [x,y,v_x,v_y].  We start by extracting these
% Note that the argument of the function has been named w, because
% we use y for another variable.
   x = w(1);
   y = w(2);
   vx = w(3);
   vy = w(4);
   vmag = sqrt(vx^2+vy^2);
   dwdt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
end

end
```

You can run this by executing the M-file. Here's the plot



Don't worry if the procedure to convert the higher order equation to MATLAB form is confusing – we will discuss this in more detail in class, and you'll have plenty of practice.

13.7 Controlling the accuracy of solutions to differential equations.

It is important to remember that MATLAB is not calculating the *exact* solution to your differential equation – it is giving you an *approximate* solution, which has been obtained using some sophisticated (and buried) computational machinery. MATLAB allows you to control the accuracy of the solution using three parameters, defined as follows. Suppose that $\mathbf{y} = [y_1, y_2 ... y_n]$ is the solution to your ODE. MATLAB computes the *relative error* for each variable, defined as:

$$e_i = \left| (y_i^{Matlab} - y_i^{correct}) / y_i^{correct} \right|$$

("How does MATLAB know the correct solution? And if it knows, why doesn't it just give the correct solution?" I hear you cry… Good questions. But I am not going to answer them). You can control the accuracy of the solution by telling MATLAB what relative tolerance you would like, as shown in the code sample below

```
function plot_vector_ode
% Function to plot the trajectory of a projectile
close all
V0 = 10;
theta = 45;
c=0.1;
tstart = 0;
tstop = 1.5;

theta = theta*pi/180; % convert theta to radians
w0 = [0;0;V0*cos(theta);V0*sin(theta)];
options = odeset('RelTol',0.00001);
[times,sols] = ode45(@odefunc,[tstart,tstop],w0,options);

figure
plot(sols(:,1),sols(:,2));

function dwdt = odefunc(t,w)
% The vector w contains [x,y,v_x,v_y].  We start by extracting these
   x = w(1);
   y = w(2);
   vx = w(3);
   vy = w(4);
   vmag = sqrt(vx^2+vy^2);
   dwdt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
end

end
```

Here, the 'odeset('*variable name',value,…*) function is used to set values for special control variables in the MATLAB differential equation solver. This example sets the relative tolerance to 10^{-5}.

13.8 Looking for special events in a solution

In many calculations, you aren't really interested in the time-history of the solution – instead, you are interested in learning about something special that happens to the system. For example, in the trajectory problem discussed in the preceding section, you might be interested in computing the maximum height reached by the projectile, or the distance it travels. You can so this by adding an `event' function to your m-file. This procedure is best illustrated by examples.

HEALTH WARNING: People find the 'events' function and its use very difficult to understand! If you are one of the few people who understand it, please find some of the confused people in the class and explain it to them… You will make many friends.

Example1: stopping the trajectory calculation when the projectile returns to the ground. If you add the function below to your `plot_vector_ode' script, and modify the call to ode45 as shown, it will continue

the calculation until the projectile reaches *y=0*, and then stop. Try it and see what happens. Note that you have to modify the `options=' line as well as adding the `event' function.

```
function plot_vector_ode
      % Function to plot the trajectory of a projectile
      close all
      V0 = 10;
      theta = 45;
      c=0.1;
      tstart = 0;
      tstop = 1.5;
      theta = theta*pi/180; % convert theta to radians
      w0 = [0;0;V0*cos(theta);V0*sin(theta)];

      options = odeset('Events',@events);
      [times,sols] = ode45(@odefunc,[tstart,tstop],w0,options);

      figure
      plot(sols(:,1),sols(:,2));

      % Print the time of impact and the distance traveled
      % The 'end' in the vectors tells MATLAB that you want
      % the last entry in the vector

      times(end)
      sols(end,1)

      function dwdt = odefunc(t,w)
      % The vector w contains [x,y,v_x,v_y].
         x = w(1);
         y = w(2);
         vx =w(3);
         vy = w(4);
         vmag = sqrt(vx^2+vy^2);
         dwdt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
      end

      function [eventvalue,stopthecalc,eventdirection] = events(t,w)
        % Function to check for a special event in the trajectory
        % In this example, we look for the point when y=0 and
        % y is decreasing, and then stop the calculation.
        % The vector w contains [x,y,v_x,v_y].
         x = w(1);
         y = w(2);
         vx = w(3);
         vy = w(4);
        eventvalue = y; % 'Events' are detected when eventvalue=0
        stopthecalc = 1; % stop if event occurs
        eventdirection = -1; %  Detect only events with dydt<0
      end
 end
```

Here is an explanation of what's happening in this example. First, notice the new lines

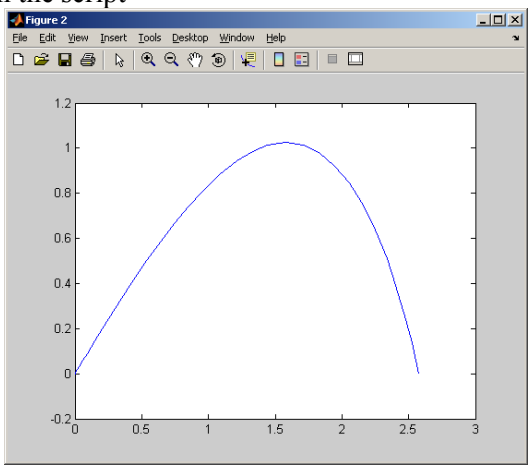
```
      options = odeset('Events',@events);
      [times,sols] = ode45(@odefunc,[tstart,tstop],w0,options);
```

The first line tells MATLAB that you want to look for a special event, and that a function called 'events' will be used to specify what you are interested in. The 'options' variable in the call to ode45 passes this information to the ODE solver, so MATLAB knows where to look for your function.
It is important to understand how MATLAB will use the function called 'events.' There is a line inside the ode45 function in MATLAB that repeatedly calls your function with different values of time t and the solution vector **w**. For each value of t and **w**, your function computes values for the three variables [eventvalue, stopthecalc,eventdirection]. ***You must write the function so that the value of the 'eventvalue' goes to zero whenever the values of t and w reach values you are interested in.***

The MATLAB ODE solver then uses your function as though it were playing the 'hot and cold' game with you. MATLAB tries lots of different values of the solution (*t*,*w*) (you never see this happening – the 'event' function is called internally from the ODE solver function). Your function must then tell MATLAB whether the values of (*t*,*w*) are getting close to the 'event' you are interested in. It does this by computing values for [eventvalue, stopthecalc,eventdirection]. Here's what the variables do:

1. The value of 'eventvalue' tells MATLAB when the event occurs. You should write your function so that 'eventvalue' goes smoothly to zero as the event you wish to detect is approached. 'eventvalue' will always be some function of time *t* and the variables stored in the vector w. Here, we are simply interested in finding when *y*=0, so we set 'eventvalue=y'. If, instead, you wanted to detect when the projectile reaches a critical distance *d* from the point where it is launched, you would set eventvalue=sqrt(y^2+x^2)-d'
2. 'stopthecalc' tells MATLAB whether or not to continue the calculation if it detects the event. If you set 'stopthecalc=1' it will stop; otherwise if you set 'stopthecalc=0' it will continue.
3. 'eventdirection' gives you some additional control over event. In general, the variable 'eventvalue' could start positive, and then decrease until it reaches zero, or could start negative, and then increase until it reaches zero. Sometimes you may only be interested in one of these two cases. For example, in our projectile problem, it is not enough to check for y=0 – the height of the projectile *starts* at y=0, so if we only check for y=0 the calculation will never get anywhere. We are only interested in finding the point where y=0 *and y is decreasing*. You can tell MATLAB this by assigning the correct value to the variable 'eventdirection,' as follows
   - If `eventdirection=0' MATLAB will detect an event each time 'eventvalue=0'
   - If 'eventdirection=1' MATLAB will respond only when eventvalue=0 and is increasing
   - If 'eventdirection= -1' MATLAB will respond only when eventvalue=0 and is decreasing.

Here's what happens if you run the script



Notice that the calculation ended when the projectile hit the ground, as expected.

***Example 2: Calculating the maximum height of the projectile*.** Sometimes we want to find features of the solution at some particular time, instead of stopping MATLAB. To do this, you can ask MATLAB to tell you the value of the solution and the time at which an event occurs. For example, suppose we want to find the maximum height of the projectile in the previous example. We can do this as follows

```
function plot_vector_ode
      % Function to plot the trajectory of a projectile
      close all
      V0 = 10;
      theta = 45;
      c=0.1;
      tstart = 0;
      tstop = 1.5;
      theta = theta*pi/180; % convert theta to radians
      w0 = [0;0;V0*cos(theta);V0*sin(theta)];

      options = odeset('Events',@events);
      [times,sols,t_event,sol_event,index_event]=...
      ode45(@odefunc,[tstart,tstop],w0,options);

      figure
      plot(sols(:,1),sols(:,2));

      % If an event has been detected, print the time at max height and
      % the solution at max height.

      if (isempty(t_event))
           'No event was detected'
      else
        t_event
        sol_event
      end

      function dwdt = odefunc(t,w)
      % The vector w contains [x,y,v_x,v_y].
         x = w(1);
         y = w(2);
         vx =w(3);
         vy = w(4);
         vmag = sqrt(vx^2+vy^2);
         dwdt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
      end
      function [eventvalue,stopthecalc,eventdirection] = events(t,w)
        % Function to check for a special event in the trajectory
        % In this example, we look for the point when y=0 and
        % y is decreasing, and then stop the calculation.
        % The vector w contains [x,y,v_x,v_y].
         x = w(1);
         y = w(2);
         vx = w(3);
         vy = w(4);
        eventvalue = vy; % 'Events' are detected when eventvalue=0
        stopthecalc = 0; % Continue even if event occurs
        eventdirection = -1; %  Detect only events with dvydt<0
      end
end
```

Notice we have changed three things in the script:

1. The eventvalue variable is now set to vy, (we know the vertical velocity is zero at the peak)
2. We asked for three new output variables from the ode45 solver:

   ```
   t_event,sol_event,index_event
   ```

   The variable t_event contains a list of the times that events were detected; sol_event is a matrix, in which each row contains the value of the solution vector at an event. Forget index_event for now - it will be discussed in Example 4.
3. We made MATLAB print the time of each event and the solution at the instant that the event occurs

   ```
   if (isempty(t_event))
       'No event was detected'
   else
     t_event
     sol_event
   end
   ```

   The isempty(*vector*) checks to see whether the vector t_event has any data in it – if not, then the max height was not reached, and no event was detected. If you run the code as is, you will find the time and solution at the event printed in the MATLAB command window as

   ```
   t_event =
     0.5759
   sol_event =
     3.2937    1.8543    4.8001   -0.0000
   ```

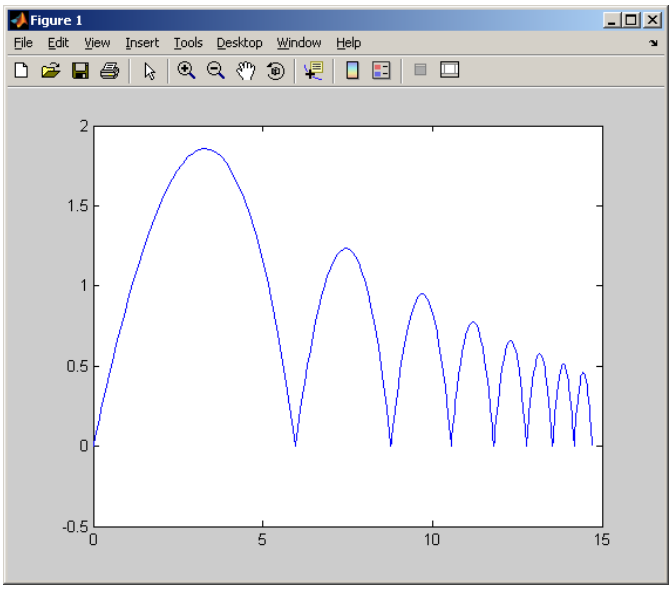
   The sol_event variable contains the value of *x,y,vx,vy* at the instant of the event. Notice that *vy=0*, as expected.

Example 3: Making the projectile bounce when it hits the ground. Sometimes, we use `events' to change the way the solution behaves. For example, we might want to make our projectile bounce when it hits the ground. You can't code this behavior into the equations of motion – you have to stop the calculation, change the velocity of the projectile to make it bounce, and start the calculation again. Here's how to modify your M-file to do this.

```
function plot_vector_ode
    % Function to plot the trajectory of a projectile
    close all
    V0 = 10;
    theta = 45;
    c=0.1;
    tstart = 0;
    tstop = 15;
    theta = theta*pi/180; % convert theta to radians
    w0 = [0;0;V0*cos(theta);V0*sin(theta)];

    options = odeset('Events',@events);

    max_number_bounces = 8;
    % The full solution will be stored in matrices times,sols
    times = 0;
    sols = transpose(w0);

    for n = 1:max_number_bounces

      [tbounce,wbounce] = ode45(@odefunc,[tstart,tstop],w0,options);
```

```
    % This appends the solution for the nth bounce to the end of the full
  solution
    n_steps = length(tbounce); % Extracts # of steps in new solution
    times = [times;tbounce(1:n_steps)];  % This appends the new times to
  end of the existing ones
    sols = [sols;wbounce(1:n_steps,:)]; % This appends new solution to
  end of y

    % This sets the initial conditions for the next bounce
    w0 = wbounce(n_steps,:); % Initial condition is sol at end of bounce
    w0(4) = -w0(4); % But we change the direction of vertical velocity
    tstart = tbounce(n_steps);

  end

  figure;
  plot(sols(:,1),sols(:,2));

  function dwdt = odefunc(t,w)
  % The vector w contains [x,y,v_x,v_y].
     x = w(1);
     y = w(2);
     vx =w(3);
     vy = w(4);
     vmag = sqrt(vx^2+vy^2);
     dwdt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
  end

  function [eventvalue,stopthecalc,eventdirection] = events(t,w)
    % Function to check for a special event in the trajectory
    % In this example, we look for the point when y=0 and
    % y is decreasing, and then stop the calculation.
    % The vector w contains [x,y,v_x,v_y].
     x = w(1);
     y = w(2);
     vx = w(3);
     vy = w(4);
    eventvalue = y; % 'Events' are detected when eventvalue=0
    stopthecalc = 1; % stop if event occurs
    eventdirection = -1; %  Detect only events with dydt<0
  end
end
```

The functions 'odefunc' and 'events' are not shown – they are the same as for the preceding example. Here's the solution for this case

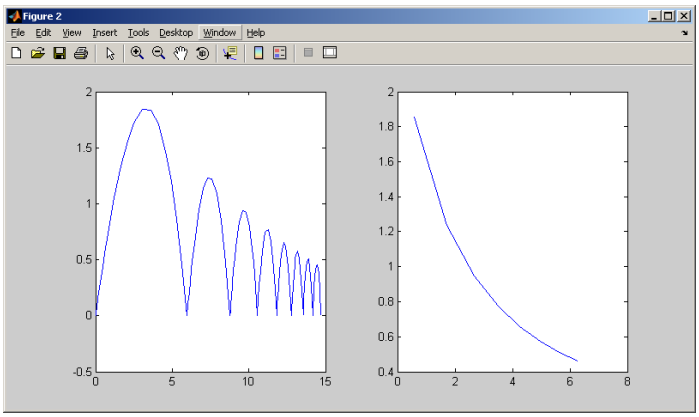


Example 4: Calculating the height of the projectile during each bounce. The `event' function can also be used to look for more than one event in a single solution. An example is shown in the script below. This script finds the time and height of the projectile at the top of each successive bounce, and plots a graph of the max bounce height as a function of time.

```
function plot_vector_ode
% Function to plot the trajectory of a projectile
close all
V0 = 10;
theta = 45;
c=0.1;
tstart = 0;
tstop = 15;

theta = theta*pi/180; % convert theta to radians
w0 = [0;0;V0*cos(theta);V0*sin(theta)];

options = odeset('Events',@events);

max_number_bounces = 8;
% The full solution will be in times,sols
times = 0;
sols = transpose(w0);
% These store the max height of each bounce, and the corresponding time
maxheight = [];
maxtime = [];
for n = 1:max_number_bounces

  [tbounce,wbounce,tevent,wevent,indexevent] =
ode45(@odefunc,[tstart,tstop],w0,options);

  % This appends the solution for the nth bounce to the end of the full
solution
  n_steps = length(tbounce);
  times = [times;tbounce(1:n_steps)];
```

```
    sols = [sols;wbounce(1:n_steps,:)];

    % This finds the event corresponding to the max height of the bounce
    % The index for this event is (2) (because its the second one listed
    % in the `event' function
    n_events = length(tevent);
    for n=1:n_events
      if (indexevent(n)==2)
          maxheight = [maxheight,wevent(n,2)];
          maxtime = [maxtime,tevent(n)];
      end
    end

    % This sets the initial conditions for the next bounce
    w0 = wbounce(n_steps,:);
    w0(4) = -w0(4);
    tstart = tbounce(n_steps);

  end

  figure('Units','pixels','Position',[250,250,800,400]);
  axes1 = subplot(1,2,1);
  plot(sols(:,1),sols(:,2));

  axes2 = subplot(1,2,2);
  plot(maxtime,maxheight);

  function dwdt = odefunc(t,w)
  % The vector w contains [x,y,v_x,v_y].  We start by extracting these
     x = w(1);
     y = w(2);
     vx =w(3);
     vy = w(4);
     vmag = sqrt(vx^2+vy^2);
     dwdt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
  end

  function [eventvalue,stopthecalc,eventdirection] = events(t,w)
    % Function to check for a special event in the trajectory
    % In this example, we look
    % (1) for the point when y=0 and y is decreasing.
    % (2) for the point where vy=0 (the top of the bounce)

    y = w(2);
    vy = w(4);
    eventvalue = [y,vy];
    stopthecalc = [1,0];
    eventdirection = [-1,0];
  end

end
```

You should see a result like this

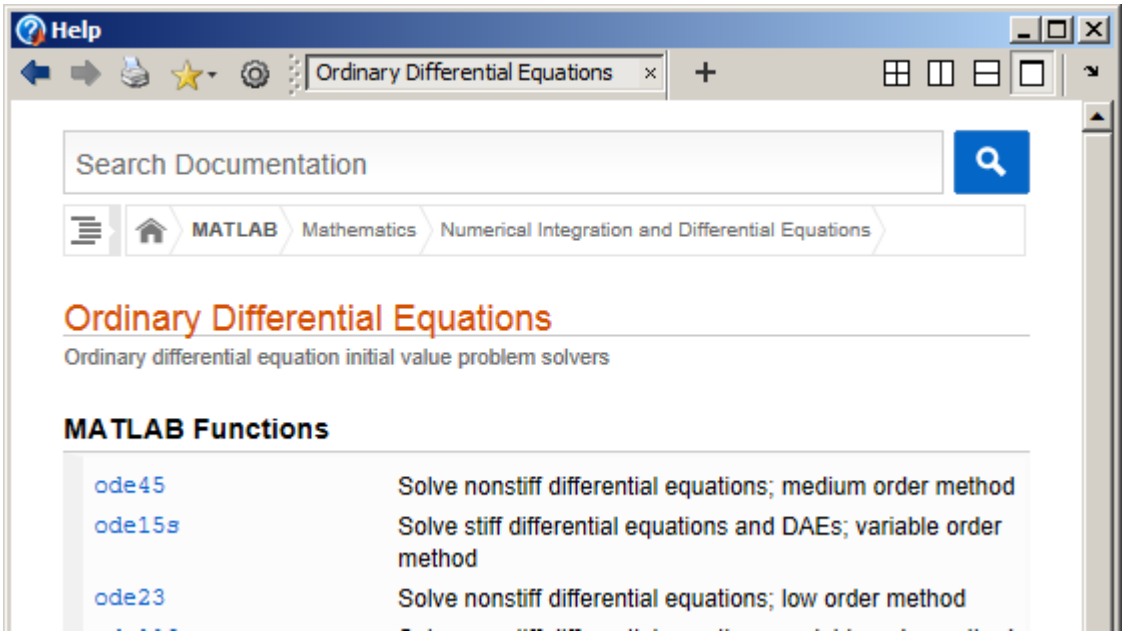


Here's what this script is doing. First, note that `eventvalue', `stopthecalc' and `eventdirection' are now vectors, because we are looking for more than one event. The first vector component tells MATLAB to look for the point where the projectile hits the ground; the second looks for the instant where y(4)=0 – i.e. the top of each bounce, where the vertical component of velocity is zero. We have also asked ode45() function to return [tevent,wevent,indexevent], as in example 2. Remember that:

1. tevent(n) tells you the time when the nth event occurred. Note that *all* events are included, in sequence, including both collisions with the ground, and the top of each bounce.
2. wevent(n,:) tells you the solution at the instant of the nth event.
3. indexevent(n) tells you what happened at the nth event – if indexevent(n)=1, it is a bounce, if indexevent(n)=2, it is the instant of max height.

13.9 Other MATLAB ODE solvers

The solver called 'ode45' is the basic MATLAB work-horse for solving differential equations. It works for most ODEs, but not all. MATLAB has many other choices if 'ode45' doesn't work. Using them is very similar to using 'ode45' but they use different numerical algorithms to solve the equations. You can learn a lot more about MATLAB ODE solvers in the section of the manual shown below



14. Using MATLAB optimizers and solvers to make design decisions

In engineering, the reason we solve equations of motion is usually so that we can select design parameters to make a system behave in a particular way. MATLAB has powerful equation solvers and optimizers that can be helpful to do this kind of calculation.

14.1 Using fzero to solve equations

The simplest design decisions involve selecting a single design parameter to accomplish some objective. If this is the case, you can always express your design problem in the form of a *nonlinear equation*, which looks something like

$$f(x)=0$$

where f is some very complicated function – perhaps involving the solution to an equation of motion. MATLAB can solve this kind of equation using the `fzero' function.

As a very simple example, here's an 'm' file that will solve the equation $x+\sin(x)=0$,

```
function solveequation
% Simple example using fsolve to solve x + sin(x)=0

sol_for_x = fzero(@fofx,[-100,100])

  function f = fofx(x)
%     Function to calculate f(x) = x + sin(x)
  f = x+sin(x);

  end
end
```

Notice that 'fsolve' works much like the ODE solver - The function 'fzero(*@function*,[*initial guess 1*,*initial guess 2*])'. The two initial guesses should bracket the solution – in other words, f(x) should change sign somewhere between *x=initial guess1* and *x=initial guess2*. The solution to the equation is printed to the main MATLAB window.

14.2 Simple unconstrained optimization example

MATLAB has a very powerful suite of algorithms for numerical optimization. The simplest optimizer is a function called 'fminsearch,' which will look for the minimum of a function of several (unconstrained) variables.

As a very simple example, let's try to find the value of *x* that minimizes the function

$$f(x)=(x-5)^2+4$$

```
function minimizef
% Function to compute projectile velocity required to hit a target
% target_position is the distance from the launch point to the target
```

```
optimal_x = fminsearch(@fofx,[3])

   function f = fofx(x)
% Function to calculate f(x)=(x-5)^2+4
      f = (x-5)^2+4;
  end
end
```

The script should be fairly self-explanatory. The function `fminsearch(@function,guess)` tries to minimize 'function' starting with an initial guess '*guess*' for the solution. If the function has many minima, it will always return the first solution that it finds, by marching downhill from the 'guess.'

For example, modify your script to minimize $f(x) = \sin(x) + x$. Try to find three or four minima by starting with different initial guesses.

Note that MATLAB has no explicit function to *maximize* anything. That's because there's no need for one. You can always turn the problem: 'maximize *f(x)*' into 'minimize *–f(x)*'.

MATLAB can also optimize a function of many variables. For example, suppose we want to find values of *x,y* that minimize

$$z(x, y) = \sin(x)\cos(y)$$

We can do this as follows

```
function minimizez
% Function to find values of x and y that minimize sin(x)*cos(y)

optimal_xy = fminsearch(@zofw,[1,2])

   function z = zofw(w)
% w is now a vector.  We choose to make w(1)=x, w(2)=y
      x = w(1); y=w(2);
      z = sin(x)*cos(y);
   end
end
```

The only thing that has changed here is that our function *zofw* is now a function of a vector w=[*x,y*] that contains values of both *x* and *y*. We also need to give MATLAB a vector valued initial guess for the optimal point (we used *x=1,y=2*).

You can include as many variables as you like in the vector *w*.

14.3 Optimizing with constraints

You are probably aware by now that most engineering optimization problems involve many *constraints* – i.e. limits on the admissible values of design parameters. MATLAB can handle many different kinds of constraint.

We'll start by summarizing all the different sorts of constraints that MATLAB can handle. As a specific example, let's suppose that we want to minimize

$$z(x, y) = \sin(x)\cos(y)$$

We could:

1. Find a solution that lies in a particular range of x and y. This could be a square region
$$x_{\min} \le x \le x_{\max} \qquad y_{\min} \le y \le y_{\max}$$
or, for something a bit more complicated, we might want to search over a triangular region
$$0 \le x \le x_{\max} \qquad x + 2y \le 4$$
2. Search along a straight line in x,y space – for example, find the values of x and y that minimize the function, subject to
$$3x - 10y = 14$$
3. Search in some funny region of space – maybe in a circular region
$$(x-3)^2 + (y-8)^2 \le 10$$
4. Search along a curve in x,y space – e.g. find the values of x and y that minimize the function, with x and y on a circle with radius $\sqrt{3}$
$$x^2 + y^2 = 3$$

In really complicated problems, you could combine all four types of constraint. MATLAB can handle all of these. Here, we'll just illustrate cases 1 and 2 (this is all you need if you want to use the optimizer in your design projects).

As a specific example, let's suppose we want to minimize
$$z(x, y) = \sin(x)\cos(y)$$
subject to the constraints
$$0 \le x \le 10 \qquad x + 2y \le 4 \qquad 3x - 10y = 14$$
These are constraints of the form 1 and 2. To put the constraints into MATLAB, we need to re-write them all in *matrix* form. Suppose that we store $[x,y]$ in a vector called **w**. Then, we must write all our constraints in one of 3 forms:
$$\mathbf{w}_{\min} \le \mathbf{w}$$
$$\mathbf{w} \le \mathbf{w}_{\max}$$
$$\mathbf{A}\mathbf{w} \le \mathbf{b}$$
$$\mathbf{C}\mathbf{w} = \mathbf{d}$$
Here, $\mathbf{w}_{\max}, \mathbf{w}_{\min}$ are vectors specifying the maximum and minimum allowable values of **w**. **A** and **C** are constant matrices, and **b** and **d** are vectors. For our example, we would write
$$\mathbf{w}_{\min} = \begin{bmatrix} 0 \\ -\infty \end{bmatrix} \qquad \mathbf{w}_{\max} = \begin{bmatrix} 10 \\ \infty \end{bmatrix}$$
$$\mathbf{A} = \begin{bmatrix} 1 & 2 \end{bmatrix} \qquad \mathbf{b} = [4]$$
$$\mathbf{C} = \begin{bmatrix} 3 & -10 \end{bmatrix} \qquad \mathbf{d} = [14]$$
In our example, **A**, **b**, **C**, **d**, each had just one row – if you have many constraints, just add more rows to all the matrices and vectors.

Now we have the information needed to write an 'm' file to do the minimization. Here it is:

```
function constrained_minimization_example
% Function to find values of x and y that minimize sin(x)*cos(y)

A=[1,2]; b=[4]; C=[3, -10]; d=[14];wmin=[0,-Inf];wmax=[10,Inf];
optimal_xy = fmincon(@zofw,[1,2],A,b,C,d,wmin,wmax)
```

```
    function z = zofw(w)
% w is now a vector.  We choose to make w(1)=x, w(2)=y
      x = w(1); y=w(2);
      z = sin(x)*cos(y);
    end
end
```

If you run this script, you'll get the following message in the MATLAB window

```
Warning: Large-scale (trust region) method does not currently solve
this type of problem,
 using medium-scale (line search) instead.
> In fmincon at 317
  In constrained_minimization_example at 5
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                           1


optimal_xy =

    4.2500   -0.1250
```

You can ignore the warning – all this guff means that MATLAB found a minimum. The values for *x* and *y* are 4.25 and -0.125.

This is just one possible minimum – there are several other local minima in this solution. You'll find that the solution you get is very sensitive to the initial guess for the optimal point – when you solve a problem like this it's well worth trying lots of initial guesses. And if you have any physical insight into what the solution might look like, use this to come up with the best initial guess you can.

Here's an explanation of how this script works. The constrained nonlinear optimizer is the function

`fmincon(@objective_function,initialguess,A,b,C,d,lowerbound,upperbound)`

1. The '*objective function*' specifies the function to be minimized. The solution calculated by the function must always be a real number. It can be a function of as many (real valued) variables as you like – the variables are specified as a vector that we will call **w**. In our example, the vector is $[x, y]$.
2. The '*initial guess*' is a vector, which must provide an initial estimate for the solution **w**. If possible, the initial guess should satisfy any constraints.
3. The arguments '*A*' and '*b*' are used to specify any constraints that have the form $\mathbf{Ax} \le \mathbf{b}$, where **A** is a matrix, and **b** is a vector. If there are no constraints like this you can put in blank vectors, i.e. just type [] in place of both **A** and **b**.
4. The arguments **C** and **d** enforce constraints of the form $\mathbf{Cw} = \mathbf{d}$. Again, if these don't exist in your problem you can enter [] in place of **C** and **d**.
5. The *lowerbound* and *upperbound* variables enforce constraints of the form $\mathbf{w}_{\min} \le \mathbf{w} \le \mathbf{w}_{\max}$, where *lowerbound* is the lowest admissible value for the solution, and *upperbound* is the highest admissible value for the solution.

15. Reading and writing data to/from files

MATLAB is often used to process experimental data, or to prepare data that will be used in another program. For this purpose, you will need to read and write data from files. Learning to read and write data is perhaps the most difficult part of learning MATLAB, and is usually a big headache in mastering any new programming language. Don't worry if this section looks scary – you won't need to do any complicated file I/O in this class. You could even skip this section altogether the first time you do this tutorial, and come back to it later.

MATLAB can read a large number of different types of file, including simple text files, excel worksheets, word documents, pdf files, and even audio and video files. We will only look at a small subset of these here.

Comma separated value files are the simplest way to get numerical data in and out of MATLAB. For example, try the following in the command window

```
>> A = pascal(5);
>> csvwrite('examplefile1.dat',A);
```

Then open the file examplefile1.dat with a text editor (Notepad will do). You should find that the file contains a bunch of numbers separated by commas. Each row of the matrix appears on a separate line. You can read the file back into MATLAB using

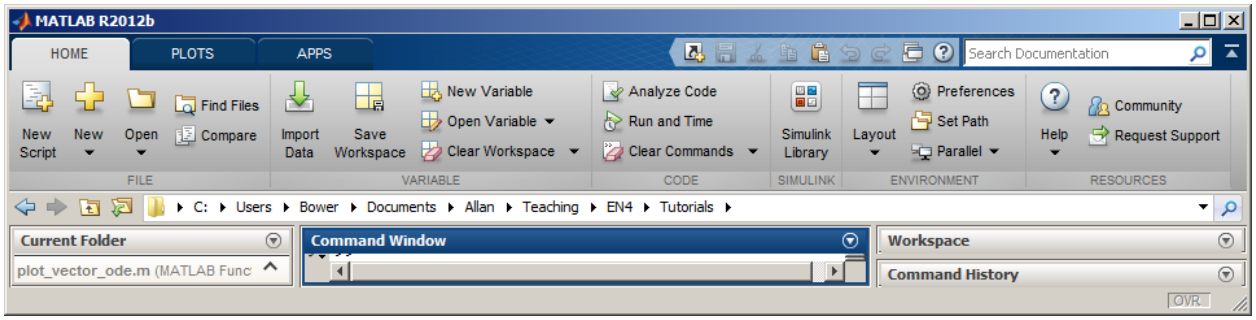
```
>> B = csvread('examplefile1.dat);
>> B
```

If you are using this method to read experimental data, and need to know how much data is in the file, the command

```
>>[nrows,ncols]=size(B)
```

will tell you how many rows and columns of data were read from the file.

Using the 'import' wizard Simple files, such as comma-separated-value files, simple spreadsheets, audio files, etc can often be read using the 'import' wizard. To try this, click the 'Import Data' button on the main matlab window (see below)



and then select 'import file' from the popup window. The resulting window should show you what's in the file, and give you some options for reading it. If you then select 'Import Selection>Import Data', the data in the file will be read into a matrix valued variable called 'examplefile1'

Formatted text files: With a bit of work, you can produce more readable text files. For example, write a MATLAB script in an M-file containing the following lines

```
s = ['Now is the winter of our discontent              ';
```

```
        'Made glorious summer by this son of York   ';
        'And all the clouds that lowrd upon our house';
        'In the deep bosom of the ocean buried          '];
% Note that you have to add blanks to make the lines all the same length, or MATLAB gives an error
   phone = [202,456,1414];
   pop = 7054168338;
   outfile = fopen('examplefile2.dat','wt');
   fprintf(outfile,' MY FILE OF USEFUL INFORMATION \n\n');
   fprintf(outfile,' First lines of Richard III \n');
   fprintf(outfile,'    %44s \n',s(1,:),s(2,:),s(3,:),s(4,:));
   fprintf(outfile,'\n Phone number for White House: ');
   fprintf(outfile,'%3d %3d %4d',phone);
   fprintf(outfile,'\n\n Estimated global population on Jan 1 2012 %d',pop);
   fclose(outfile);
```

Then run the script, open the file examplefile2.dat to see what you got. For full details of the fprintf command you will need to consult the MATLAB manual (look under 'Functions' for the entry entitled I/O). Here is a brief explanation of what these commands do.

1. The outfile=fopen('*filename*','wt') command opens the file – the argument 'wt' indicates that the file will be opened for writing. If you omit the second argument, or enter 'r' the file will be opened for reading.
2. The fprintf(outfile,' *some text* \n') command will write *some text* to the file. The \n tells MATLAB to start a new line after writing the text.
3. The fprintf(outfile,' %44s \n', *string1, string2, ...*) command writes a succession of character strings, each 44 characters long. Each character string will be on a new line.
4. The fprintf(outfile,'%3d %3d %4d',*vector*) writes a number 3 digits long, a space, another number 3 digits long, another space, and then a number 4 digits long.

It usually takes a lot of fiddling about to get fprintf to produce a file that looks the way you want it.

To read formatted files, you can use the 'fscanf' command – which is essentially the reverse of fprintf. For example, the following script would read the file you just printed back into MATLAB

```
infile = fopen('examplefile2.dat','r');
heading = fscanf(infile,' %30c',1);
play = fscanf(infile,' %27c',1);
s = fscanf(infile,'    %47c \n',4);
dummy = fscanf(infile,'%30c',1);
phone = fscanf(infile,'%4d',3);
pop = fscanf(infile,'%*46c %d',1);
fclose(infile);
```

Run this script, then type

```
>>s
>>phone
>>pop
```

into the command window to verify that the variables have been read correctly.

Again, you can read the manual to find full details of the fscanf function. Briefly

1. `heading = fscanf(infile,' %30c',1)` reads the first line of the file – the %30c indicates that 30 characters should be read, and the ',1' argument indicates that the read operation should be performed once.
2. `s = fscanf(infile,'    %47c \n',4)` reads the 4 lines of the play: each lines is 47 characters (including the white space at the start of each line), and the \n indicates that each set of 47 characters is on a separate line in the file.

3. `phone = fscanf(infile,'%4d',3)` reads the White-house phone number, as 3 separate entries into an array. The %4d indicates that each number is 4 digits long (the white space is counted as a digit and ignored).
4. The `pop = fscanf(infile,'%*46c %d',1)` reads the world population. The syntax %*46c indicates that 46 characters should be read and then ignored –i.e. not read into a variable.

It is usually a pain to use fscanf – you have to know *exactly* what your input file looks like, and working out how to read a complicated sequence of strings and numbers can be awful.

Reading files with textscan The problem with *fscanf* is that you need to know exactly what a file looks like in order to read it. An alternative approach is to read a large chunk of the file, or even the whole file, at once, into a single variable, and then write code to extract the information you need from the data. For example, you can read the whole of 'examplefile2.dat' into a single variable as follows

```
>> infile = fopen('examplefile2.dat','r');
>> filedata = textscan(infile,'%s');
>> filedata{:}
```

(Note the curly parentheses after filedata). This reads the entire file into a variable named '*filedata'*. This variable is an example of a data object called a 'cell aray' – cell arrays can contain anything: numbers, strings, etc. In this case, every separate word or number in the file appears as a separate string in the cell array. For example, try typing

```
>> filedata{1}{1}
>> filedata{1}{10}
>> filedata{1}{50}
```

This will display the first, 10^{th} and 50^{th} distinct word in the file. Another way to display the entire contents of a cell array is

```
>>celldisp(filedata)
```

You can now extract the bits of the file that you are interested in into separate variables. For example, try

```
>> pop = str2num(filedata{1}{58});
>> for n=1:3 phone(n) = str2num(filedata{1},{47+n}); end
>> pop
>> phone
```

Here, the 'str2num' function converts a string of characters into a number. (The distinction between a string of numerical characters and a number is apparent only to computers, not to humans. But the computers are in charge and we have to humor them).

Challenge: see if you can reconstruct the variable 's' containing the Shakespeare. This is quite tricky…

MATLAB can read many other types of file. If you are using a Windows PC, and enjoy bagpipe music, you could download this file and then type

```
>> [y,fs,n] = wavread(filename);
>> wavplay(y,fs);
```

16. Animations

In dynamics problems it is often fun to generate a movie or animation that illustrates the motion of the system you are trying to model. The animations don't always tell you anything useful, but they are useful to kill time in a boring presentation or to impress senior management.

Making movies in MATLAB is simple – you just need to generate a sequence of plots, and display them one frame at a time. But animations can be a bit of a chore – it can take some fiddling about to get

MATLAB to plot frames at equal time intervals, and it is tedious to draw complicated shapes – everything needs to be built from scratch by drawing 2D or 3D polygons.

As always, the procedure is best illustrated with examples.

Example 1: Movie of a flying projectile: The script below makes a movie showing the path of a projectile

```
function trajectorymovie
% Function to make a movie of the trajectory of a projectile

close all
V0 = 1; % Launch speed
theta = 75; %Launch angle
c = 0.2; % Drag coefficient
time_interval = 0.005;  % time interval between movie frames
max_time = 1; % Max time of simulation.
theta = theta*pi/180; % convert theta to radians
w0 = [0;0;V0*cos(theta);V0*sin(theta)];

options = odeset('Events',@events);
[t_vals,sol_vals] = ode45(@odefunc,[0:time_interval:max_time],w0,options);

for i=1:length(t_vals)
   clf % Clear the frame
   plot(sol_vals(:,1),sol_vals(:,2)) % Plot the trajectory as a line
   hold on
%  The projectile is simply a single point on an x-y plot

plot(sol_vals(i,1),sol_vals(i,2),'ro','MarkerSize',20,'MarkerFaceColor','r');
   axis square
   pause(0.05); % This waits for a short time between frames
end

function dydt = odefunc(t,w)
   x = w(1); y=w(2); vx = w(3); vy = w(4);
   vmag = sqrt(vx^2+vy^2);
   dydt = [vx;vy;-c*vx*vmag;-9.81-c*vy*vmag];
end

      function [eventvalue,stopthecalc,eventdirection] = events(t,w)
        % Function to check for a special event in the trajectory
        % In this example, we look
        % (1) for the point when y(2)=0 and y(2) is decreasing.
        % (2) for the point where y(4)=0 (the top of the bounce)
       y = w(2); vy = w(4);
       eventvalue = [y,vy];
       stopthecalc = [1,0];
       eventdirection = [-1,0];
      end

end
```

Example 2: Movie of a freely rotating rigid body The script below makes a movie showing the motion of a rigid block, which is set spinning with some initial angular velocity. The code also can print an animated .gif file that you can put on a webpage, use in a powerpoint presentation, or post on your facebook page.

```
function tumbling_block
% Function to make a movie of motion of a rigid block, with dimensions
% a = [a(1),a(2),a(3)], which has angular velocity vector omega0
% at time t=0
% The code includes lines that can save the movie to an animated gif
% file for use e.g. on a webpage or a presentation.
a = [1,2,4]; % Dimensions of the block
omega0 = [0,1,0.1]; % Initial angular velocity of the block
time = 30; % Time of the computation
n_frames = 240; % # frames in the movie

% This creates vertices of a block aligned with x,y,z, with one
% corner at the origin
initial_vertices =
[0,0,0;a(1),0,0;a(1),a(2),0;0,a(2),0;0,0,a(3);a(1),0,a(3);a(1),a(2),a(3);0,a(
2),a(3)];
% This loop moves the center of mass of the block to the origin
for j=1:3
  initial_vertices(:,j) = initial_vertices(:,j)-a(j)/2;
end
face_matrix = [1,2,6,5;2,3,7,6;3,4,8,7;4,1,5,8;1,2,3,4;5,6,7,8];

% This sets up the rotation matrix at time t=0 and the angular
% velocity at time t=0 as the variables in the problem
y0 = [1;0;0;0;1;0;0;0;1;transpose(omega0(1:3))];
% This creates the inertia matrix at time t=0 (the factor m/12 has
% been omitted as it makes no difference to the solution)
I0 = [a(2)^2+a(3)^2,0,0;0,a(1)^2+a(3)^2,0;0,0,a(1)^2+a(2)^2];

options = odeset('RelTol',0.00000001);
% Here we use ode45 in a new way.  Instead of computing times and
% solution values at discret points, we put everything about the solution
% in a variable called sol.  We can use this variable to reconstruct
% the solution later.
sol = ode45(@odefunc,[0,time],y0,options);

close all
count = 0;
ax_length = max(a)/2;
scrsz = get(0,'ScreenSize');
% The line below can be used to change the size of the image
% (useful if you want to reduce the size of the .gif file)
figure1 = figure('Position',[scrsz(3)/2-200 scrsz(4)/2-200 400
400],'Color',[1 1 1]);
for i=1:n_frames
   count = count + 1;
   t = time*(i-1)/(n_frames-1);
   y = deval(sol,t);
   R= [y(1:3),y(4:6),y(7:9)];
   new_vertices = transpose(R*transpose(initial_vertices));
   clf;
```

```
%   axes1 = axes('Parent',figure1,'Position',[0.3804 0.3804 0.44 0.46]);

patch('Vertices',new_vertices,'Faces',face_matrix,'FaceVertexCdata',hsv(6),'F
aceColor','flat');
   axis([-ax_length,ax_length,-ax_length,ax_length,-ax_length,ax_length])
   axis off
   pause(0.05)
%   Uncomment the lines below to create an animated .gif file
%   if (count==1)
%        set(gca,'nextplot','replacechildren','visible','off')
%        f = getframe;
%        [im,map] = rgb2ind(f.cdata,256,'nodither');
%    else
%      f = getframe;
%      im(:,:,1,count) = rgb2ind(f.cdata,map,'nodither');
%    end
end
% Uncomment the line below to save the animated .gif file
%imwrite(im,map,'animation.gif','DelayTime',0,'LoopCount',inf)

function  dwdt = odefunc(t,w)
% Function to calculate rate of change of rotation matrix
% and angular acceleration of a rigid body.
   omega = [w(10:12)];  % Angular velocity
   R= [w(1:3),w(4:6),w(7:9)]; %Current rotation matrix
   I = R*I0*transpose(R); %Current inertia matrix
   alpha = -I\(cross(omega,I*omega)); % Angular accel
% Next line computes the spin matrix
   S = [0,-omega(3),omega(2);omega(3),0,-omega(1);-omega(2),omega(1),0];

   Rdot = S*R; % Rate of change of rotation matrix dR/dt
   dwdt(1:9) = Rdot(1:9); %Columns of dR/dt arranged in sequence in dydt
   dwdt(10:12) = alpha;
   dwdt = transpose(dwdt); %make dydt a column vector
end

end
```


A.F. Bower
School of Engineering
Brown University
December 2012