



School of Engineering
Brown University

Dynamics and Vibrations MATLAB tutorial

To prepare for HW1, do sections 1-11.6 – you can do the rest later as needed

1. [What is MATLAB](#)
2. [Starting MATLAB](#)
3. [Basic MATLAB windows](#)
4. [Using the MATLAB command window](#)
5. [MATLAB help](#)
6. [MATLAB 'Live Scripts'](#) (for algebra, plotting, calculus, and solving differential equations exactly)
 - 6.1 [Solving Equations](#)
 - 6.2 [Functions and Plotting](#)
 - 6.3 [Calculus](#)
 - 6.4 [Vectors](#)
 - 6.5 [Matrices](#)
 - 6.6 [Solving differential equations](#) (with symbols)
7. [Working with M-files](#)
8. [MATLAB Functions in M files](#)
9. [Basic Programming Concepts](#)
 - 9.1 [Loops](#)
 - 9.2 [Creating a vector with a loop](#)
 - 9.3 [Dot notation for operations on vectors](#)
 - 9.4 [Operations on vectors using a loop](#)
 - 9.5 [Creating matrices with nested loops](#)
 - 9.6 [How to find the number of rows and columns in a matrix](#)
 - 9.7 [Using plots to display curves and surfaces stored in vectors and matrices](#)
 - 9.8 [Conditional statements](#)
10. [Organizing complex calculations as functions in an M-file](#)
11. [Solving ordinary differential equations \(ODEs\) using MATLAB](#)
 - 11.1 [Solving a basic differential equation](#)
 - 11.2 [Solving a basic differential equation in an M-file](#)
 - 11.3 [Solving a differential equation with adjustable parameters](#)
 - 11.4 [Common errors](#)
 - 11.5 [Solving simultaneous differential equations](#)
 - 11.6 [Controlling the accuracy of solutions to differential equations](#)
 - 11.7 [Looking for special events in a solution](#)
 - 11.8 [How the ODE solver works](#)
 - 11.9 [Other MATLAB differential equation solvers](#)
12. [Using MATLAB solvers and optimizers to make design decisions](#)
 - 12.1 [Using fzero to solve equations](#)
 - 12.2 [Simple unconstrained optimization problem](#)
 - 12.3 [Optimizing with constraints](#)
13. [Reading and writing data to/from files](#)
14. [Movies and animation](#)

1. What is MATLAB?

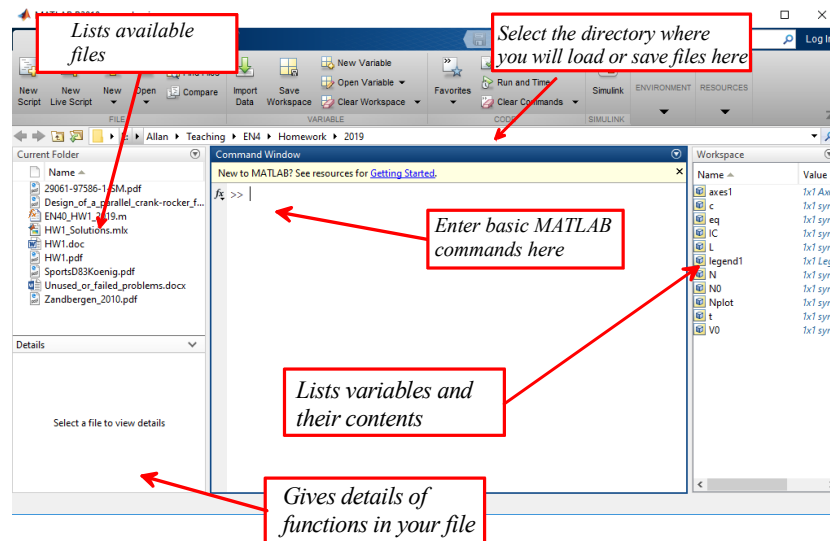
You can think of MATLAB as a sort of graphing calculator on steroids – it is designed to help you manipulate very large sets of numbers quickly and with minimal programming. MATLAB is particularly good at doing matrix operations (this is the origin of its name). It is also capable of doing symbolic computations, and has huge numbers of built-in packages that will do things like image processing, design control systems, machine learning, bioinformatics, and so on. It is also capable of communicating with a lot of hardware –for example you can use it to program or communicate with Arduinos.

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://www.brown.edu/information-technology/software/catalog/matlab> The instructions tell you to wait for an email from mathworks, but they don't always send one. Just check your account – 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 2019b, but if you are using 2019a that should work fine.

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. Using the MATLAB command window

You can use the MATLAB command window as a simple 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
>>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?

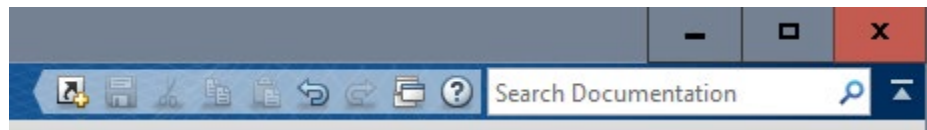
We almost never use MATLAB like this, however – the format of the output is difficult to read, and it's hard to keep track of complicated calculations. Instead, we write a script to do the calculations, and then have MATLAB execute the script. There are two general types of MATLAB script:

- 'Live Scripts,' which display the results of your calculations, and are good for doing algebra, calculus, plotting functions, and so on;
- 'Scripts' or 'm files' which store MATLAB functions – these are usually used for programming, and can also be easier to use than a 'Live Script' when we want to do calculations with numbers (data processing, simple programming, and in ENGN40, finding numerical solutions to differential equations).

We will show how to use both type of script in this tutorial.

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). The matlab manuals are also online, and will come up in google search.



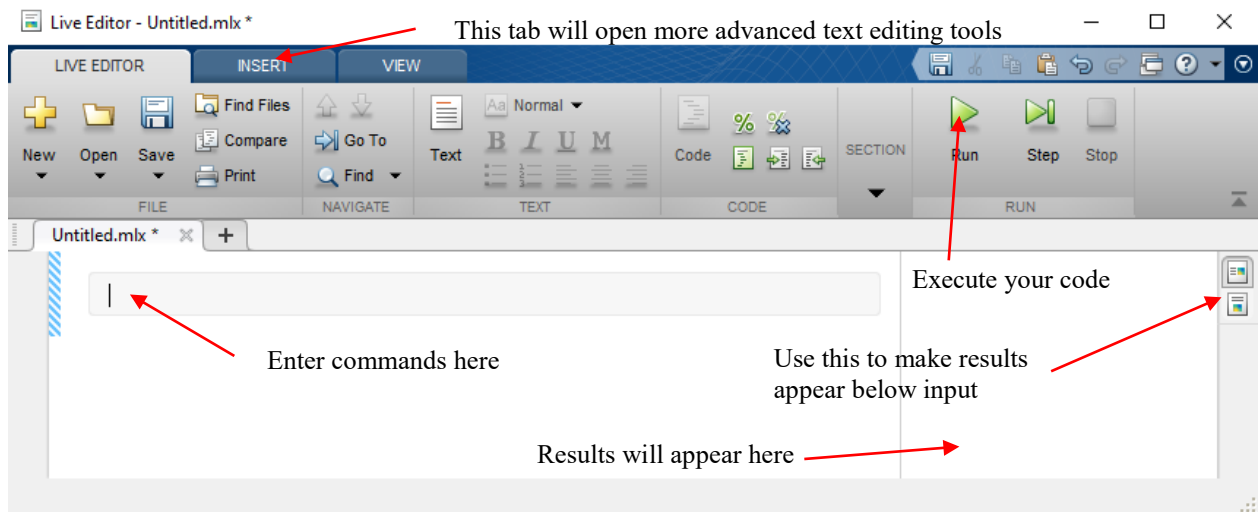
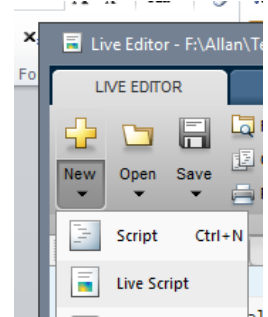
If you already know the name of the MATLAB function you want to use the help manual is quite good – you can just enter the name of the function in the search, and a page with a good number of examples usually comes up. It is more challenging to find out how to do something, but most of the functions 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. MATLAB 'Live Scripts'

If you would like to use MATLAB to do math calculations, algebra, or to plot graphs, you can write a MATLAB 'Live Script.' This will organize your calculations for you, and will also display the results in a nice clean format.

Create a new 'Live Script' using the New > Live Script... menu on the top left hand corner of the MATLAB window.

You should see a menu like the one below.



In the next few sections we will work through some useful things you can do with a 'Live Script.'

6.1 Solving Equations

Try typing this into the rectangular box:

```
syms x y
eq1 = x + y == 5
eq2 = x - y == -5
[x,y] = solve([eq1,eq2],[x,y])
```

Then press the 'Run All' arrow at the top of the window to execute the script. Note that:

- When you use MATLAB to do algebra (with symbols instead of numbers) you have to specify that any variables that appear in your equations are 'syms.' This is a bit annoying, but you will get used to it.
- Notice the two different uses of '='. The 'eq1=' creates a MATLAB object called eq1 that stores the first equation. The object can be used in future calculations. If you want to create an equality inside an equation you have to use ==

Now try (just edit your script; you don't have to type it all in again)

```
syms x y a b
eq1 = x + y == a;
eq2 = x - y == -b;
solution = solve([eq1,eq2],[x,y])
```

then run the script again with the green arrow. Note that:

- Putting a semicolon at the end of a line suppresses the output. We usually hide intermediate steps in calculations to make the script more readable.
- The result of this calculation looks a bit funny. The 'solution' variable is a MATLAB object, and you can access its contents with

```
solution.x
solution.y
```

(just add these lines to the end of your script and press Run All again).

Now do a more complicated equation:

```
syms x y a b z
eq1 = x^2 + y^2 == a;
eq2 = x - y == b;
[x,y] = solve([eq1,eq2],[x,y])
x(1)
x(2)
z = x(1)-y(1)
```

Notes:

- There are now two solutions, which are returned as vectors. The rows of the vectors match up – so the first entry in x and the first entry in y is one solution; and the second also go together
- You can extract the first solution in the vectors using x(1), y(1), and the second with x(2),y(2).
- You can use the results of one calculation in a subsequent one.

Now try

```
syms x y a b z
eq1 = x^2 + y^2 == a;
eq2 = x - y == b;
[x,y] = solve([eq1,eq2],[x,y]);
z = dot(x,y)
simplify(z)
```

This has calculated the dot product of the two vectors x and y. In some versions of MATLAB, the results may look funny – if you see bars on top of the variables, they denote complex conjugates, and the || denote taking the magnitude of a complex number. If you are using MATLAB 2017b this behavior won't show up.

If you get complex numbers you can tell MATLAB that some of your variables are real

```
syms x y a b z
assume(a,'real')
assume(b,'real')
eq1 = x^2 + y^2 == a;
eq2 = x - y == b;

[x,y] = solve([eq1,eq2],[x,y])
z = dot(x,y)
simplify(z)
```

Notice that the bars on the a and b variables are now gone (because the complex conjugate of a real number is equal to the number). But there are still bars over the square roots, because of course they might still be complex.

You can try asking for real valued x and y as well

```
syms x y a b z
assume(a,'real')
assume(b,'real')
assume(x,'real')
assume(y,'real')
eq1 = x^2 + y^2 == a;
eq2 = x - y == b;
[x,y] = solve([eq1,eq2],[x,y])
z = dot(x,y)
simplify(z)
```

but now you get a warning that the solution is valid only if a and b have certain values (that make sure your square-roots are positive). You can make the warning go away with

```
syms x y a b z
assume(a,'real')
assume(b,'real')
assumeAlso(b^2<2*a)
assume(x,'real')
assume(y,'real')
eq1 = x^2 + y^2 == a;
eq2 = x - y == b;
[x,y] = solve([eq1,eq2],[x,y])
z = dot(x,y)
simplify(z)
```

Not all equations can be solved algebraically, of course. Create a new section in your live script (use the Section Break button) then try

```
eq1 = x*cos(x)^2 == 5
solve(eq1,x)
```

This will give you an error message. That is because the 'clear all' clears all variables, and x is no longer defined as a symbol. The error message is not very helpful, however. Fix the problem with

```
clear all
syms x
eq1 = x*cos(x)^2 == 5
solve(eq1,x)
```

MATLAB gives you a solution now, but can only give you a number (and warns you that the solution is approximate, although to most engineers 31 digits is beyond our wildest dreams). You have to be careful with numerical solutions because your equation could have many solutions but MATLAB can only give you one (you should be able to see that $x*\cos(x)^2 == 5$ has many solutions). There is no way to find all the roots of an equation like this, but you can ask for a solution in a particular range using

```
clear all
syms x
eq1 = x*cos(x)^2 == 5
vpasolve(eq1,x,[10,15])
```

The 'vpa' here stands for 'variable precision arithmetic – it will give you a numerical answer, but with a very large number of decimal places (far more than the 8 or 16 that is the usual precision for floating point arithmetic – see eg the example in Sect 9.1)

Matlab does not always give you all the solutions to an equation even if it can find them – for example try

```
clear all
syms x
eq1 = cos(x)^2 == 5
solve(eq1,x)
```

This gives you two roots, but if you want to see them all, you have to use

```
clear all
syms x
eq1 = cos(x)^2 == 5
sols = solve(eq1,x,'Returnconditions',true)
sols.x
sols.parameters
sols.conditions
```

The answer is again barely comprehensible to humans, but the 'parameters' and 'conditions' are trying to tell you that the solution is $\cos^{-1}(\sqrt{5}) + k\pi$ where k is an integer. k is the 'parameter,' and the 'Z' in the 'conditions' is some sort of code known only to mathematicians saying k is an integer.

If you want the solution for a particular value of the parameter k you can use

```
clear all
syms x
eq1 = cos(x)^2 == 5
sols = solve(eq1,x,'Returnconditions',true)
sols.x
sols.parameters
sols.conditions
subs(sols.x,sols.parameters,3)
```

This substitutes $k=3$ into the solution.

For more information about equation solutions type 'solve - equations and systems' into the search bar of the matlab search window

6.2 Functions and plotting


Create a new section in your Live Script, and try the following (use the 'run section' button to run only the new section):

```
clear all
syms f(t,k) t k
f(t,k) = sin(k*t)
fplot(f(t,2*pi),[0,1])
```

This has (1) Defined the function $f(t,k)$, and (2) plotted the result for $k = 2\pi$ for $0 < t < 1$.

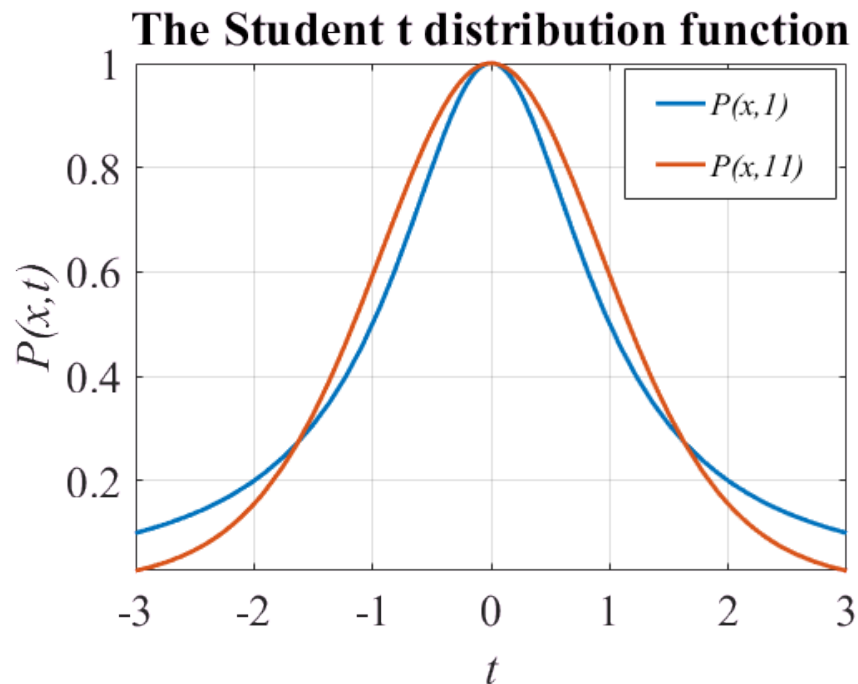
You can edit a plot to make it look nice: to do so click on the plot, and then click the arrow on the top right hand corner. This will open the plot in a MATLAB 'figure' window. You can then click the



arrow in the figure window , and double click the plot. This will open a menu that allows you to add axis labels, a grid, change the fonts, and the line styles for your plot. Once you have the plot the way you want it, you can generate the MATLAB code that will edit the plot automatically by going to 'File>Generate Code'. You can cut and paste parts of the code into your 'Live Script.'

Here's an example that generates a fancy plot

```
clear all
syms x t P(x,t)
P(t,x) = (1 + t^2/x)^(-((x + 1)/2));
fplot(P(t,1),[-3 3],'DisplayName','P(x,1)','Color',[0 0.447 0.741],...
'LineWidth',2);
hold on
fplot( P(t,11),[-3 3],'DisplayName','P(x,11)','Color',[0.85 0.325 0.098],...
'LineWidth',2);
axes1 = gca;
xlabel({'t'},'FontAngle','italic','FontName','Times New Roman');
title('The Student t distribution function');
ylabel({'P(x,t)'},'FontAngle','italic','FontName','Times New Roman');
set(axes1,'FontName','Times New Roman','FontSize',20,'XGrid','on','YGrid',...
'on');
legend1 = legend(axes1,'show');
set(legend1,...
'Position',[0.666 0.708 0.229 0.187],...
'FontAngle','italic',...
'FontSize',14);
```



You can use this as a template for most simple x-y plots you might want to create. You may find that your plot gets corrupted if you run sections of a Live Script individually – you can usually fix this by running the full Live Script again from the beginning.

The ‘hold on’ statement makes several curves appear on the same set of axes. If you want to make a new plot (with a new set of axes) you can use the ‘figure’ statement to open a new set of axes. For example

```
figure
syms x
fplot(x*sin(x),[0,4])
```

You can do far more than just x-y plots with a Matlab Live Script: for example, you can type any of the following into the matlab help search box:

- fcontour (for contour plots)
- fsurf (for a 3D surface)
- fplot3 (for a 3D parametric curve)

You can explore these when you need them – it is hard to remember all the different plot functions.

6.3 Calculus

MATLAB is good at calculus. For basic differentiation and integration, try

```
syms x deriv
deriv = diff(x^2,x)
int(deriv)
```

Notice that MATLAB has not included a constant of integration – you have to remember to add it in yourself if it is needed. You can do higher order derivatives as well

```
syms x f dfdx d2fdx2
f = x*sin(x)^2*cos(x)^2
dfdx = diff(f,x)
d2fdx2 = diff(f,x,2)
dfdx = int(d2fdx2)
f = int(dfdx)
```

Partial Derivatives: You can do partial derivatives

```
syms f x y
f = sqrt(x^2+y^2);
diff(f,x)
```

Maximizing a function As an example, let’s try to maximize

$$f(x) = \frac{x}{a+x}$$

We can do this using the usual calculus tricks: the maxima must be at points where $df / dx = 0$, so we can solve this for x and then substitute back into f . Here’s the MATLAB

```
clear all
syms f x a dfdx xatmax
f = x/(a+x^2);
dfdx = diff(f,x);
xatmax = solve(dfdx==0,x)
subs(f,x,xatmax)
```

Notice that MATLAB finds two solutions to $df/dx = 0$ - here, we just substituted them both back into f and found a value of f for both. The second solution is bigger than the first, so that must be the maximum. But we could also substitute the solutions one at a time, eg using

```
subs(f,x,xatmax(2))
```

Definite integrals: MATLAB can also do definite integrals

```
syms f x sigma
assume(sigma>0);
f = exp(-x^2/sigma^2);
int(f,x,[-inf,inf])
```

Try doing the integral without the `assume(sigma>0)`

```
clear all
syms f x sigma
f = exp(-x^2/sigma^2);
int(f,x,[-inf,inf])
```

The 'clear all' is important – otherwise MATLAB does not forget the 'assume.'

Numerical integration: Of course not all integrals can be evaluated exactly

```
clear all
syms f x
f = sin(sin(x));
int(f,x,[0,pi/2])
```

but you can get MATLAB to give you a numerical approximation (this only works if everything in the integrand is a number, it can't contain any symbols) – try this:

```
clear all
syms f x
f = sin(sin(x));
double(int(f,x,[0,pi/2]))
```

Taylor Series: If you love Taylor series, MATLAB is your friend

```
clear all
syms f x
f = sin(x)/x;
taylor(f,x,0,'Order',12)
```

The 'Order' parameter determines how many terms to include in the series (usually we only use one or two!)

Vector Calculus: MATLAB engineers appear to have passed multivariable calculus as well:

```
clear all
syms f x y z v
f = sqrt(x^2+y^2+z^2)
v = gradient(f,[x,y,z])
curl(v)
```

6.4 Vectors

You can create a vector in a Live Script by entering a list of numbers or variables separated by a comma (for a row vector) or semicolon (for a column vector)

```
clear all
syms v w a b c d e f
v = [a, b, c]
w = [d; e; f]
```

You can do all the usual operations on vectors

```
cross(v,w)
dot(v,w)
v+w
```

Can you remember how to get rid of the complex conjugates? We will discuss some more things you can do with vectors in the section on writing Matlab functions below

6.5 Matrices

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

$$\begin{array}{c} \text{Column 2} \\ \downarrow \\ \begin{bmatrix} 1 & 5 & 0 & 2 \\ 5 & 4 & 6 & 6 \\ 3 & 3 & 0 & 5 \\ 9 & 2 & 8 & 7 \end{bmatrix} \begin{array}{l} \leftarrow \text{row 1} \\ \\ \\ \leftarrow \text{row 4} \end{array} \\ \uparrow \\ \text{Column 3} \end{array}$$

You can create a matrix (with either numbers or symbols)

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

You can see individual elements of the matrix using `matrix(row,column)`

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

You can extract a whole column of A using

```
A(:,3)
```

(The `:` is shorthand for all the rows); similarly if you want a whole row you can use

```
A(2,:)
```

You can also assign values to elements one at a time

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

You can also create some special matrices

```
eye(4)
pascal(6)
magic(5)
```

You can do all sorts of things to matrices in Matlab (assuming you are the sort of person that likes doing things to matrices)

1. You can flip rows and columns with

$$B = \text{transpose}(A)$$

2. You can add and subtract matrices (provided they have the same number of rows and columns)

$$C = A+B$$

$$C-\text{transpose}(C)$$

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

$$D= B*A$$

to find the product of A and B. Also try the following

$$D= B*A$$

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

$$\text{det}(A)$$

$$\text{inv}(A)$$

$$\text{eig}(A)$$

$$[W,D] = \text{eig}(A)$$

These examples all used numbers, but MATLAB can handle symbolic matrix calculations as well

```
clear all
syms A v v1 v2 v3 a11 a12 a13 a21 a22 a23 a31 a32 a33
A = [a11,a12,a13;a21,a22,a23;a31,a32,a33]
v = [v1;v2;v3]
det(A)
A*v
```

You can use matrices to solve 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

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

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

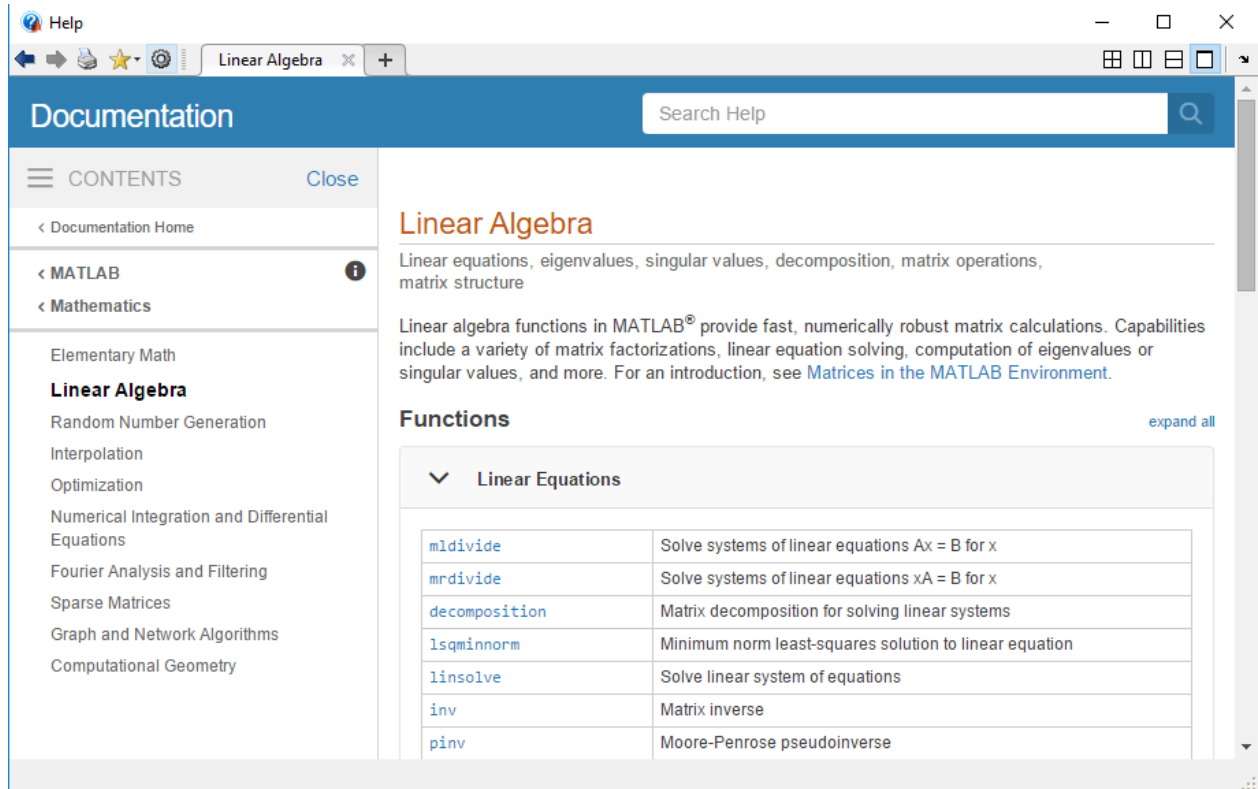
To solve these in MATLAB, you would simply type

```
clear all
A = [3,4,5;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

$$A*x$$

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



6.6 Solving differential equations

What is a differential equation? Differential equations are mathematical models used by engineers, physicists, mathematical biologists, economists, and other people with no life. 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 $g(t)$. Basically, it says that if g is small, it increases slowly; if g is large, it increases quickly. If you know the value of g at time $t=0$ (eg $g(0) = g_0$), the equation can be solved to see that $g(t) = g_0 \exp(kt)$ (you should know how to do this – separate variables and integrate).

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

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

where k is a constant. We will see many other examples. All the differential equations we solve will come from Newton's law $\mathbf{F}=\mathbf{ma}$ (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).

Matlab is very good at solving differential equations. We will discuss two methods. In a 'Live script' we usually try to solve an equation with symbols. But solving differential equations is a lot like integration – you can only get symbolic solutions if you are lucky. Many equations have to be solved numerically. For this purpose we usually write a matlab function (in an M file) and usually also need to write some simple programs to process the results. This will be discussed in more detail in Section 10.

For now we focus on symbolic solutions. Here's how to solve the GDP example

```
clear all
syms diffeq g(t) t k g0
diffeq = diff(g(t),t) == k*g(t)
initial_condition = g(0) == g0
g(t) = simplify(dsolve(diffeq,initial_condition,symvar(g(t))))
```

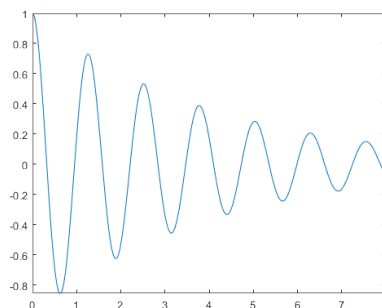
Here the 'symvar' function tells MATLAB to solve for $g(t)$. You can solve higher order differential equations too – for example, here is the differential equation that predicts how a spring-mass system (like a simple model of a suspension) vibrations – you will get to know and love¹ this equation later in the course

```
clear all
syms x(t) t omega_n zeta x0 v0
diffeq = (1/omega_n^2)*diff(x(t),t,2) + 2*zeta/omega_n * diff(x(t),t) + x(t)==0
Dx = diff(x)
initial_condition = [x(0)==x0, Dx(0)==v0]
x(t) = simplify(dsolve(diffeq,initial_condition,symvar(x(t))))
```

The answer looks scary – we'll discuss what the equations mean in class – but if you substitute numbers and plot the answer

```
fplot(subs(x(t),[omega_n,zeta,v0,x0],[5,0.05,0,1]),[0,8])
```

you should see a nice damped vibration response.

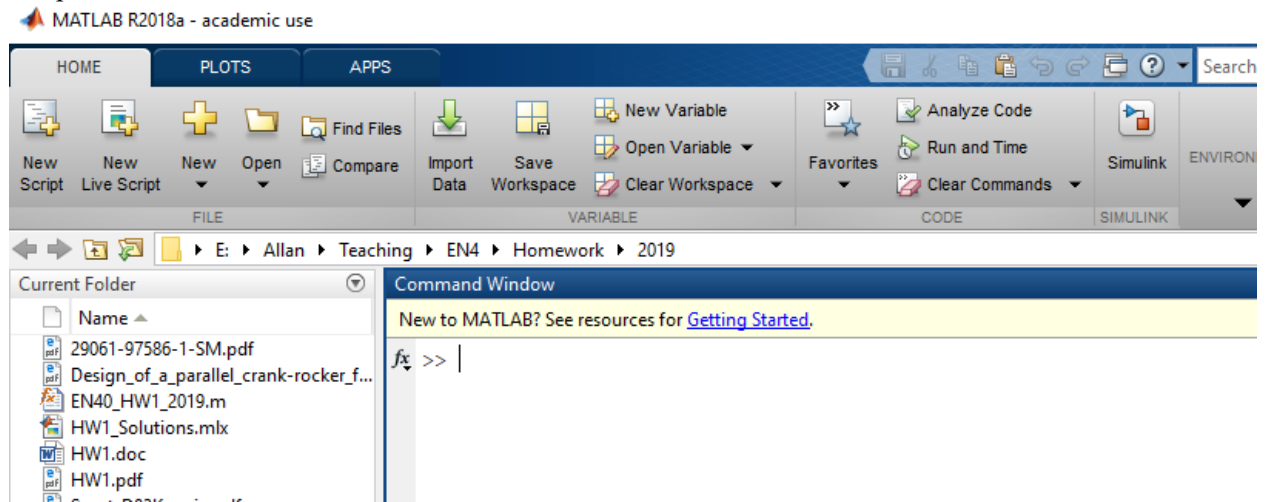


¹ Well OK maybe not love...

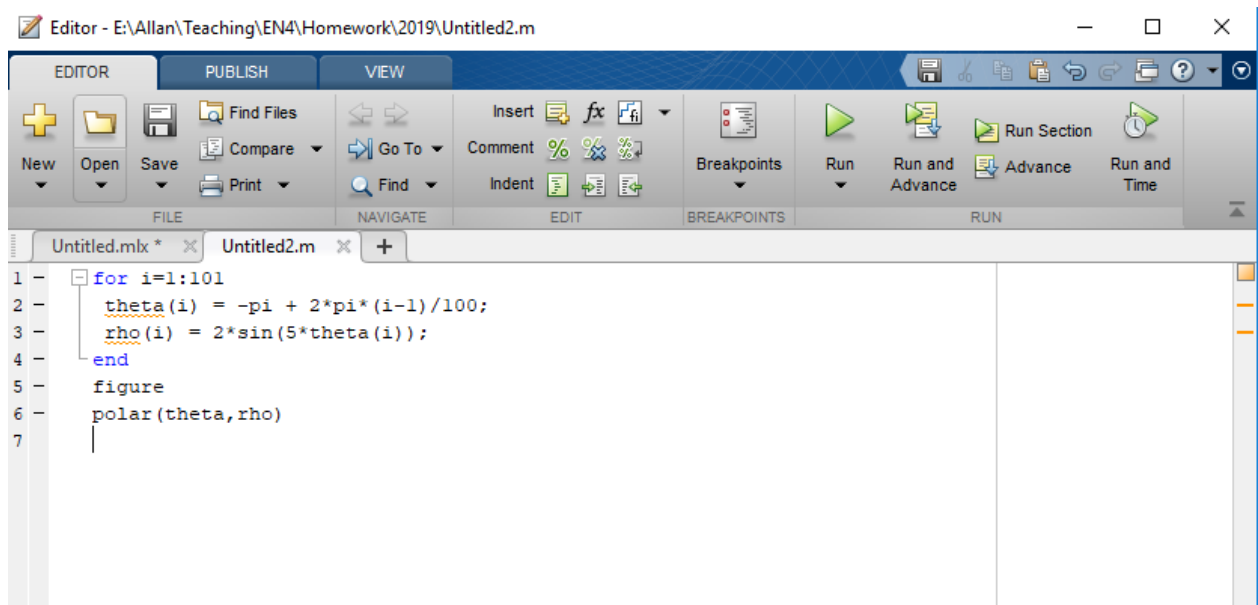
7. Working with M files

So far, we've run MATLAB through a 'Live script.' This works well for doing algebra, plotting, and some basic programming. Live scripts are not usually used for more complicated matlab programming however. For example, for some of the engn40 projects, we need to write matlab functions that can be called by other matlab programs, or (for the quadcopter project) from a python code.

For this sort of programming we use a matlab 'M-file.' 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)
```

(Don't worry about what the commands actually do for now – we will discuss them in more detail soon). 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 (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)

8. Writing MATLAB functions in m files *This is an important section of this tutorial*

We usually use M-files to define new MATLAB *functions* – these are programs that can accept user-defined data and produce a new result. For example, many engineers find this function to be a useful substitute for social skills:

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 s = pickup(person)
% function to say hello to someone you admire
s = ['Hello ' person ' you write beautiful matlab code']
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.)

3. Save the file (accept the default file name, which is `pickup.m`)
4. Type the following into the command window
`>> 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.)

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

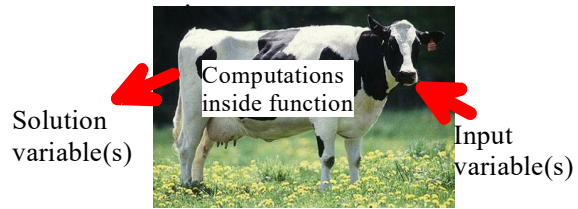
```
function solution_variable = function_name(input_variables...)
```

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

```
solution_variable = ...
end
```


Note also 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.

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. Note that

- Functions can accept more than one input variable; for example this (useless) function will multiply two numbers together

```
function product = multiplyonly(a,b)
% function to multiply two numbers
product = a*b;
end
```

To run this from the command window, save the function in a new M file (use the default name) and use

```
>>answer = multiplyonly(1,4)
```

(the input variables must be numbers otherwise you will get an error. For example

```
>> answer = multiplyonly(x,y)
```

will bomb.

- Functions can return more than one output variable; for example this will both multiply and divide the input variables

```
function [product,ratio] = multiplyanddivide(a,b)
% function to multiply two numbers
product = a*b;
ratio = a/b;
end
```

Run this from the command window using

```
>> [p,r] = multiplyanddivide(1,4)
```

- Functions can accept and return lots of different types of data besides just variables containing numbers. 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 func_to_plot(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
```

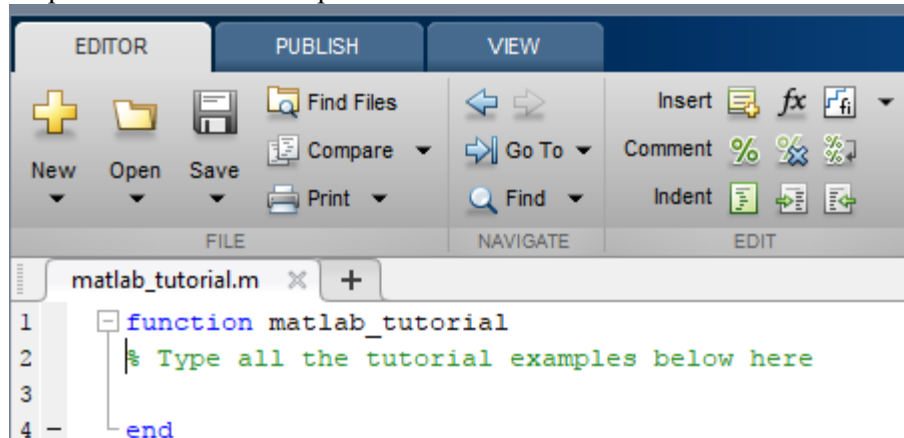
Then save the M-file, and try plotting a cosine, by using

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

Don't worry about the programming concepts in this script –those will be discussed later- the point to notice is that you can pass a *function* into another function, using the 'function handle' @(function name). This is important, because the built-in MATLAB differential equation solvers, optimizers, and equation solvers use this feature to allow a user to specify what functions they want to integrate, or minimize.

9. Basic programming concepts

You can do all the examples in this section in a new M file. Start by creating a new script (not a Live script) and set it up to look like the example below:



then save the file. Type all the examples that appear below between the ‘function’ and ‘end’ lines. It will be important to save most of the homeworks you do in ENGN40 as a function with this general format.

9.1 Loops

Loops are used to repeat calculations many times. Try the example below (we show the whole script)

```
1 function matlab_tutorial
2 % Type all the tutorial examples below here
3 a = 0
4 for i=1:10
5     a = a + 0.1
6 end
7 valueofa = a
8 aminusone = a-1.0
9 end
```

Run the script with the green arrow, or through the command window. Look at the MATLAB command window to see the results of the calculations – every calculation will be displayed so it’s quite a lot of garbage.

The lines that read “for i=1:10 a= a + 0.1; end” create 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 i 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 i=1 and ends with i=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. The error occurs because it is not possible to store most real numbers exactly with a finite number of binary digits. MATLAB always converts your script into binary code; runs the calculation with simple binary operations; then converts the answer back into something you can read. You have to live with the limitations of binary computations.

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.

9.2 Creating vectors with a loop

You can create a vector containing regularly spaced data points very quickly with a loop. Add the following lines to your script

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

Here, the 'clc' clears the command window. 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.

Notice you get a warning from MATLAB about v changing its length. You can fix this by creating the vector before the loop

```
clc
v = zeros(11,1);
for i=1:11
    v(i) = 0.1*(i-1);
end
v
```

(You have to run the script and look at the command window again to see what happens). This creates a vector with 11 rows and 1 column, and then modifies it. This gives you faster code. It doesn't make much difference in small scripts like this, but when you write programs that need to process huge vectors and matrices, programming style becomes very important.

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
```

Notice that MATLAB doesn't get upset about the fact that you made v with only 11 numbers – the new ones get added on the end automatically.

9.3 Dot notation for operations on vectors

You can often manipulate the numbers in a vector using simple MATLAB commands. For example, if you add

```
w = sin(v)
```

to the end of your script MATLAB will compute the sin of every number stored in the vector v and return the result as another vector w . 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

```
v^2
```

This will cause MATLAB to bomb, because the square of a column 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).

9.4 Operations on vectors using a loop

Dot notation can be hard to use until you get used to it. It is often clearer to do operations with a loop. For example, instead of writing $w = v.^2$, you could use

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

Here, ‘for $i=1:\text{length}(v)$ ’ repeats the calculation for every element in the vector v . The function $\text{length}(\text{vector})$ determines how many components the vector v has (101 in this case).

Using loops is not elegant programming, and slows down MATLAB. For ‘real’ programming it should be avoided, but for simple scripting and homework problems loops are fine.

9.5 Creating matrices with nested loops

You can ‘nest’ loops inside each other to do more complicated operations. Before you run this script, see if you can work out what $A(3,3)$ is for yourself. Then run the script to see if you were right.

```
clc
for i=1:3
    for j=1:3
        A(i,j) = i+j;
    end
end
A(3,3)
```

9.6 How to find the number of rows and columns in a matrix

If you need to find out the number of rows and columns in a matrix you can use

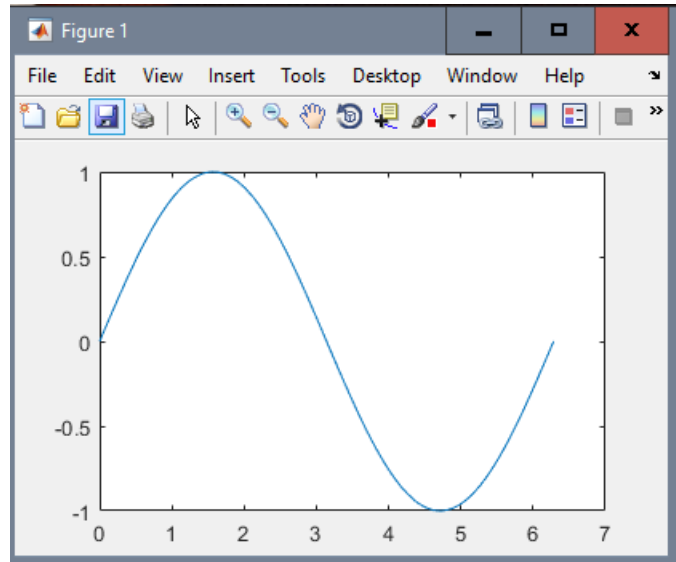
```
[nrows,ncols] = size(A)
```

9.7 Using plots to display curves and surfaces stored in vectors and matrices

We already looked at how to do plots in ‘Live scripts.’ Plotting in MATLAB functions is a bit different. This is because functions are usually working with numbers, whereas live scripts usually work with symbols. To plot the results of calculations with numbers, we store curves and surfaces as vectors and matrices, and then use MATLAB plot functions to display them. Try this in a new script:

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

then use ‘Run’ to see the result.



Here, x and y are just rows of numbers. MATLAB plots the graph by joining up successive pairs of x - y points with straight lines. For the plot to work, x and y must be vectors, and they must have the same length.

You can use the steps described in section 6.2 to edit the plot by hand, and generate the MATLAB code to make the plot look fancy.

```
close all
for i=1:101
    x(i)= 2*pi*(i-1)/100;
end
y=sin(x);
plot(x,y);
grid('on')
xlabel({'x'}, 'FontAngle', 'italic', 'FontName', 'Times New Roman');
title('A Sin');
ylabel({'sin(x)'}, 'FontAngle', 'italic', 'FontName', 'Times New Roman');
```

To plot multiple lines on the same plot you can use (add these to the bottom of your script)

```
hold on
z = sin(2*x);
plot(x,z)
```

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
```

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. You can use the File>Generate code menu to write the matlab code that does this automatically, and then paste it into your m file.

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);
figure
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 $0 \leq x \leq 1$, $0 \leq y \leq 1$, you can use

```
clear 'x'
clear 'y'
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)
```

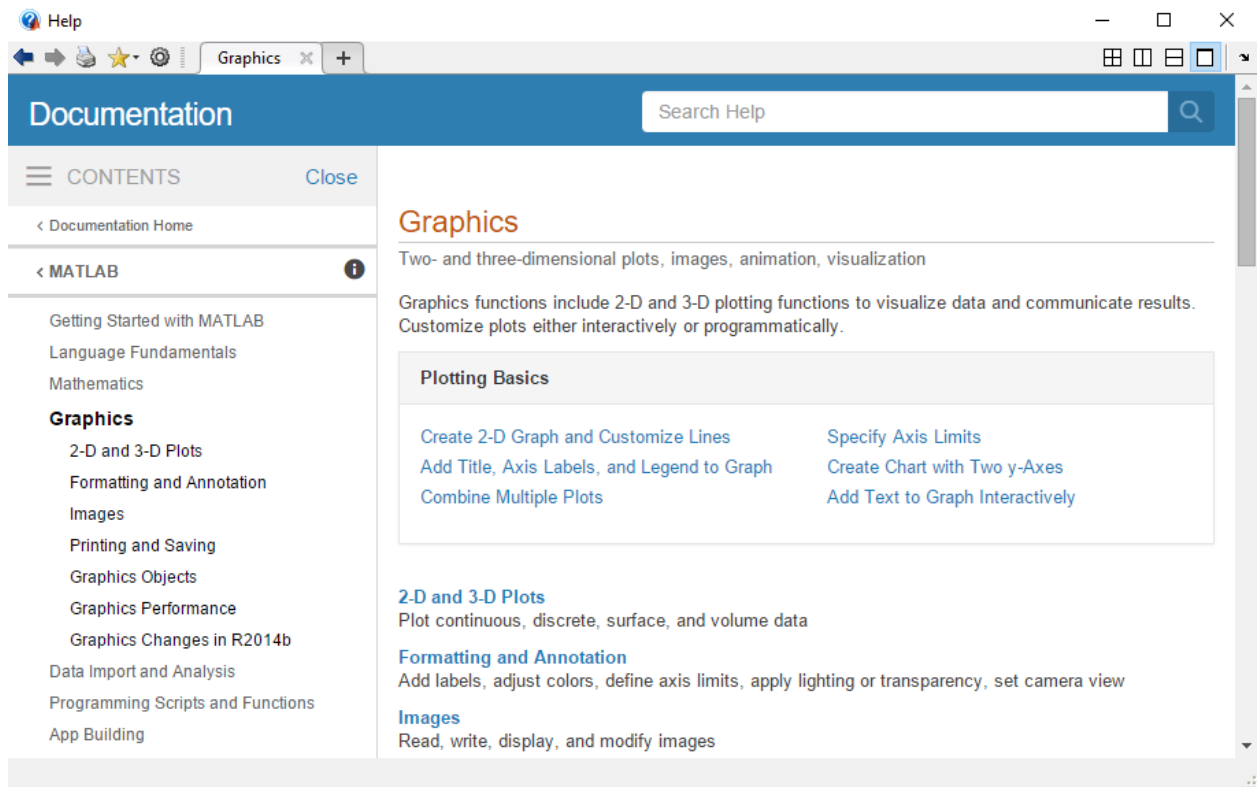
You need the 'clear' because x and y have already been defined to be vectors with sizes bigger than 51, so we need to erase them before creating smaller vectors. The next two lines of this sequence should be familiar: they create row vectors of equally spaced points. The third needs some explanation – 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 (or just google matlab plotting)



9.8 Conditional statements

Let's try plotting a function with a discontinuity in it – for example (from an old ENGN40 homework problem)

$$w = \begin{cases} \cos(v) & \cos(v) \geq \sin(v) \\ \sin(v) & \cos(v) \leq \sin(v) \end{cases}$$

as a function of v for $0 \leq v \leq 4\pi$.

We can do this with a loop (there are other, faster ways – this is not elegant code!)

```
v = 0:0.05:4*pi;
for i=1:length(v)
    cosv = cos(v(i));
    sinv = sin(v(i));
    if (cosv>sinv)
        w(i) = cosv;
    else
        w(i) = sinv;
    end
end
figure
plot(v,w)
```

The first line shows a new way to create a vector of roughly equally spaced points (except for the last point). The loop iterates over each entry in the vector v and calculates the cosine and sine of the i th entry in the vector².

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
elseif (a second test)
    calculation to do if the first test is false and the second test is true
else
    calculation to do if both tests are false
end
```

You can use a number of different logical tests (on numbers): examples are

$(a>b)$, $a<b$, $a>=b$, $a<=b$, $a==b$, $a\neq b$

The last two here check if a and b are equal, or not equal, respectively. You can make more complicated operations with the logical ‘and’ and ‘or’ operators, eg

```
if (a>b || a<100)
    do something here
end
```

will execute the statement between the ‘if’ and ‘end’ if $a>b$ or $a<100$.

² Make sure you understand why the code uses $\cos(v(i))$ instead of $\cos(v)$ – if you don't see this try to figure it out by reading sections 9.3 and 9.4 in the tutorial, or ask someone to explain!

Similarly

```
if (a>b && a<100)
    do something here
end
```

will execute the statements if $a>b$ and $a<100$. You can use multiple $\&\&$ and $\|\|$ to construct very complicated conditions, if you wish.

Note that MATLAB also uses conditional statements to operate on entire vectors – this is an advanced topic and we won't discuss it here. But be aware that if you type $(a<b)$ and a and b are vectors, MATLAB will not throw an error, because this is a meaningful statement to MATLAB. In the simple codes we write in ENGN40, however, we would normally want to do something like $(a(i)<b(i))$ to check if the i th entry in a is less than the i th entry in b . Be careful to include the (i)

10. 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 an 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
    ysqared(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.*** A bit like an engineer at a party...

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`. This convention is called 'variable scope' and is an important programming concept – different languages have different conventions for variable scope.

11. Numerical solutions to differential equations

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.

11.1 Solving a basic differential equation using the command window.

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(@(t,y) simple_ode(t,y),[0,20],0);
>> plot(t_values,y_values)
```

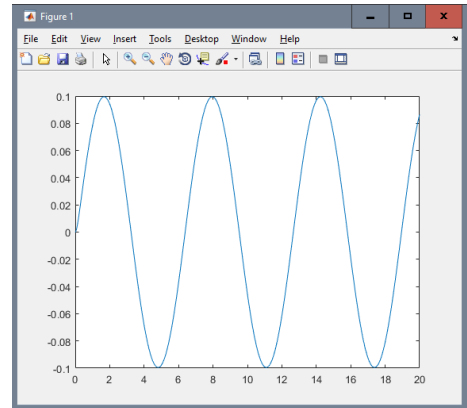
Here,

`ode45(@(variables) function_name(variables),[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 on the right.

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(@(t,y) simple_ode(t,y),[0,20],0);
```

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

1. @(t,y) simple_ode(t,y) specifies 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.

$$t_values = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ \vdots \end{bmatrix} \quad y_values = \begin{bmatrix} y(t_1) \\ y(t_2) \\ y(t_3) \\ y(t_4) \\ \vdots \end{bmatrix}$$

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 – Section 9 - if you need to).

It's important to understand that MATLAB has not given you a formula for y as a function of time (if you want a formula, you have to use the method described in Section 6.5 – but this only works in very special cases where the differential equation happens to have an exact solution). In the example here, MATLAB has given you *numbers* for the value of the solution y at a set of different times. By default, MATLAB chooses the times for each solution point, but you can control this if you like.

HEALTH WARNING: It is not easy to understand how MATLAB is using the ‘simple_ode’ function. This is discussed in a bit more detail in Section 13.9 – for now the important thing is to work out how to use the function, but you could glance through 13.9 and see if it makes any sense to you – if it does, it might be clearer how ode45 works.

11.2 Solving a basic differential equation in an M-file *This is an important section of this tutorial*

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

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

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 time t=0 to t=20, with y(0) = -2
    initial_y = -2;
    time_range = [0,20];

    [t_values,sol_values] = ode45(@(t,y) diff_eq(t,y),time_range,initial_y);

    plot(t_values,sol_values); % Plot the solution

end % end statement terminating the solve_bernoulli function

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

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 the ‘diff_eq’ function comes *after* the line that computes the solution to the ODE (the line with ‘ode45’ in it), but 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 is common to put the functions near the end of the file so they can easily be found. In fact the diff_eq function could even be stored in a different file (this is how we would write professional code, but it’s a pain submitting homework assignments when they consist of many files, so we won’t do it here).

The solution to the ODE is computed on the line

```
[t_values,sol_values] = ode45(@(t,y) diff_eq(t,y),time_range,initial_y);
```

The function handle `@(t,y) diff_eq(t,y)` is used to tell MATLAB two things:

1. Which function to use to compute the value of dy/dt . MATLAB will be repeatedly using your function internally to compute dy/dt at many different times between $t=0$ and $t=20$, but you don’t see this happening.
2. The `@(t,y)` tells matlab that the variable y is differentiated with respect to t in the equation. These are dummy variables, so, for example this would work just as well:

```
[t_values,sol_values] = ode45(@(x,z) diff_eq(x,z),time_range,initial_y);
```

The main reason that 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. We will discuss this in class, but if you can't wait to find out, you can read section 13.9 and see if you can work out what's going on by yourself.

11.3 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. We can do this using the script shown below

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

close all % Closes any open plot windows.

% Define parameter values below.
k = 10;
F = 1;
omega = 1;
time_range = [0,20];
y0 = 0;

[t_values,sol_values] = ode45(@(t,y) diff_eq(t,y,k,F,omega), ...
    time_range,y0);

plot(t_values,sol_values);

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

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

Notice that:

1. We can add additional variables to our differential equation function
`function dydt = diff_eq(t,y, k, F, omega)`
The variables can appear in any order you like.
2. You have to specify that y is differentiated with respect to t with `@(t,y)`
3. You have to list all the variables in the function in the `ode45` line:
`@(t,y) diff_eq(t,y,k,F,omega), ...`
The order has to be consistent

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.

11.4 Common errors

The examples below show the errors you get if you get functions in the wrong order when you solve a differential equation. Start with a code that works:

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

close all % Closes any open plot windows.

% Define parameter values below.
k = 10;
F = 1;
omega = 1;
time_range = [0,20];
y0 = 0;

[t_values,sol_values] = ode45(@(t,y) diff_eq(t,y,k,F,omega), ...
    time_range,y0);
    plot(t_values,sol_values);

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

Example 1 Now try moving the line that solves the ODE to the top of the script:

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

close all % Closes any open plot windows.

[t_values,sol_values] = ode45(@(t,y) diff_eq(t,y,k,F,omega), ...
    time_range,y0);

% Define parameter values below.
k = 10;
F = 1;
omega = 1;
time_range = [0,20];
y0 = 0;
plot(t_values,sol_values);
end
function dydt = diff_eq(t,y, k, F, omega) % Function defining the ODE
    dydt = -k*y + F*sin(omega*t);
end
```

This bombs (check the command window to see the error). The reason is that MATLAB executes statements in the following order (i) It scans your file for functions, and stores them in memory so they are ready to be used; (ii) it executes the statement in the main function from the top down. So, in this example, MATLAB (i) stores the `diff_eq` function in memory; (ii) closes figure windows; (iii) attempts to solve the differential equation. But it can't do this because the variables `tstart`, `tstop`, and `initial_y` have not been given values.

Example 2: Try fixing the problem by defining `tstart`, `tstop`, and `initial_y` properly, but leave the definition of `k`, `F` and `omega` where they are

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

    close all % Closes any open plot windows.
    tstart = 0;
    time_range = [0,20];
    y0 = 0;

[t_values,sol_values] = ode45(@(t,y) diff_eq(t,y,k,F,omega), ...
    time_range,y0);

% Define parameter values below.
    k = 10;
    F = 1;
    omega = 1;
    plot(t_values,sol_values);
end
function dydt = diff_eq(t,y, k, F, omega) % Function defining the ODE
    dydt = -k*y + F*sin(omega*t);
end
```

Again, this bombs with a whole string of errors. The reason is a bit more difficult to understand. This time, Matlab (i) stores the `diff_eq` function in memory; (ii) closes figure windows; (iii) assigns values to `tstart`, `tstop` and `initial_y`; (iii) attempts to solve the differential equation. When it solves the differential equation, MATLAB needs to use your `diff_eq` function, which has to compute a numerical value for `dydt` given values for `t`, `y`, `k`, and `F`. But the values of `k`, `F` and `omega` have not yet been specified, which causes MATLAB to crash.

Example 3: Try forgetting one of the `k`, `F` and `omega` variables

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

close all % Closes any open plot windows.

% Define parameter values below.
    k = 10;
    F = 1;
    omega = 1;
    time_range = [0,20];
    y0 = 0;

[t_values,sol_values] = ode45(@(t,y) diff_eq(t,y,k,omega), ...
    time_range,y0);
    plot(t_values,sol_values);
end
function dydt = diff_eq(t,y, k, omega) % Function defining the ODE
    dydt = -k*y + F*sin(omega*t);
end
```

This produces **Undefined function or variable 'F'**.

Example 4: Try forgetting one of the k, F and ω in the ode45 line

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

close all % Closes any open plot windows.

% Define parameter values below.
k = 10;
F = 1;
omega = 1;
time_range = [0,20];
y0 = 0;

[t_values, sol_values] = ode45(@diff_eq(t,y,k,omega), ...
    time_range, y0);
plot(t_values, sol_values);

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

This produces **Not enough input arguments**. (This always makes me think of this old [Monty Python](#) sketch)

11.5 Solving simultaneous differential equations *This is an important section of this tutorial*

Most problems in engineering involve more complicated differential equations than those described in the preceding section. Usually, we need to solve several differential equations at once. As an example, we will solve a homework problem from 2012. The famous ‘[Belousov-Zhabotinskii](#)’ reaction is an example of a chemical reaction that results in cyclic variations in concentration of the reagents (you can find [movies](#) showing the reaction). Reactions of this type are of interest in biology, and a number of theoretical models have been developed to describe them. One example is the ‘cubic autocatalator’ reaction, which is described by differential equations

$$\frac{dx}{dt} = 0.6 - xy^2 - 0.1x \quad \frac{dy}{dt} = xy^2 - y + 0.1x$$

Here, $x(t)$ and $y(t)$ represent the concentrations of two reacting chemicals. The point of the equations is to predict how the concentrations fluctuate with time. The two equations specify how quickly each concentration varies, given the concentration values.

Our mission is to calculate and plot the time variation of x and y , given that at time $t=0$ $x=y=1$.

To do this, we must first define a function that will calculate values for both dx/dt and dy/dt given values of x, 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

$$\frac{d}{dt} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0.6 - xy^2 - 0.1x \\ xy^2 - y + 0.1x \end{bmatrix}$$

This is an equation that has the general form

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

where \mathbf{w} is a column vector

$$\mathbf{w} = \begin{bmatrix} x \\ y \end{bmatrix}$$

If we write a MATLAB function to compute $d\mathbf{w}/dt$ given \mathbf{w} , then MATLAB will be able to solve the differential equation. Here's a script that does this.

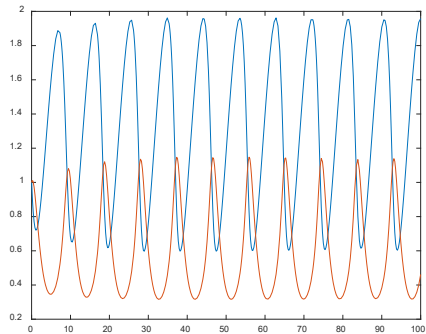
```
function I_love_chemical_engineering
% Function to solve the cubic autocatalator equations

    close all % Closes any open plot windows.
    time_range = [0,100];
    x0 = 1;    % Initial value of x
    y0 = 1;    % Initial value of y
    initial_w = [x0;y0]; % Initial value of the vector w

    [t_values,sol_values] = ode45(@(t,w) diff_eq(t,w),time_range,initial_w);
    plot(t_values,sol_values);

end
function dw_vectordt = diff_eq(t,w_vector) % Function defining the ODE
    x = w_vector(1); % x is stored as the first element in w_vector
    y = w_vector(2); % y is stored as the second element in w_vector
    dxdt = 0.6-x*y^2-0.1*x;
    dydt = x*y^2-y+0.1*x;
    dw_vectordt = [dxdt; dydt]; % The time derivative of w_vector
end
```

You should see a plot that looks like the figure. The graph shows the variation of x (blue line) and y (red line) with time. The concentrations fluctuate about once every 10 seconds or so (this was the point of the calculation!)



Some notes on this example:

- The general procedure is *exactly* the same as for solving a single differential equation. The only change is that now the unknown variable is a vector, the initial value of the variable is a vector, and the solution contains values for both x and y .
- A common error when solving vector valued differential equations is to forget to make the `dw_vectordt` column vector. Try this by editing your code to
`dw_vectordt = [dxdt, dydt];`

This produces the error

**Error using odearguments (line 93)
@(T,W)DIFF_EQ(T,W) must return a column vector.**

Again, this is in classical geek and incomprehensible to most of us – just remember that if you get an error that has something to do with column vectors it is usually a problem in your differential equation function.

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. In this calculation, the variable ‘t_values’ 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 ‘sol_values’ is a *matrix*. To see what’s in the matrix, suppose that solution values x, y were computed at a bunch of different times t_1, t_2, t_3, \dots . The variables ‘time_values’ and ‘sol_values’ then contain a vector and a matrix that look like this

$$t_values = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ \vdots \end{bmatrix} \quad sol_values = \begin{bmatrix} x(t_1) & y(t_1) \\ x(t_2) & y(t_2) \\ x(t_3) & y(t_3) \\ x(t_4) & y(t_4) \\ \vdots & \vdots \end{bmatrix}$$

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 x at time times(i) can be extracted as sols(i,1), while the solution for y is sols(i,2).

Understanding this lets us plot the solution in many different ways (try them by editing your script):

- To plot only x as a function of time use `plot(t_values, sol_values(:,1))`
- To plot only y as a function of time use `plot(t_values, sol_values(:,2))`
- To plot x on the x axis – y on the y axis use
`plot(sol_values(:,1), sol_values(:,2))`

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

11.6 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 $\mathbf{y} = [y_1, y_2, \dots, y_n]$ is the solution to your ODE. MATLAB computes the *relative error* for each variable, defined as:

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

(“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. You can pay some more tuition and sign up for a numerical methods course in a future semester to find out...). 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 I_love_chemical_engineering
% Function to solve the cubic autocatalator equations

close all % Closes any open plot windows.
time_range = [0,100];
x0 = 1;    % Initial value of x
y0 = 1;    % Initial value of y
initial_w = [x0;y0]; % Initial value of the vector w

options = odeset('RelTol',0.00001);
[t_values,sol_values] = ode45(@(t,w) diff_eq(t,w),...
    time_range,initial_w,options);
plot(t_values,sol_values);

end
function dw_vectordt = diff_eq(t,w_vector) % Function defining the ODE
x = w_vector(1); % x is stored as the first element in w
y = w_vector(2); % y is stored as the second element in w
dxdt = 0.6-x*y^2-0.1*x;
dydt = x*y^2-y+0.1*x;
dw_vectordt = [dxdt; dydt]; % The time derivative of w
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} .

11.7 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 chemistry example you might be interested in finding the time when the two concentrations x , y first have equal values. The script below shows how to do this.

```

function I_love_chemical_engineering
% Function to solve the cubic autocatalator equation

close all % Closes any open plot windows.
time_range = [0,100];
x0 = 1;    % Initial value of x
y0 = 1;    % Initial value of y
initial_w = [x0;y0]; % Initial value of the vector w

options = odeset('RelTol',0.00001,...
    'Events',@(t,w) event_function(t,w));

[t_values,sol_values] = ode45(@(t,w) diff_eq(t,w),...
    time_range,initial_w,options);
plot(t_values,sol_values);

figure
yminusx = sol_values(:,2)-sol_values(:,1);
plot(t_values,yminusx);

```

```

function dw_vectordt = diff_eq(t,w_vector) % Function defining the ODE
    x = w_vector(1); % x is stored as the first element in w
    y = w_vector(2); % y = stored as the second element in w
    dxdt = 0.6-x*y^2-0.1*x;
    dydt = x*y^2-y+0.1*x;
    dw_vectordt = [dxdt; dydt]; % The time derivative of w
end

function [ev,stop,dir] = event_function(t,w_vector)
    x = w_vector(1); % x is stored as the first element in w
    y = w_vector(2); % y = stored as the second element in w
    ev = y-x; % 'Events' are detected when eventvalue=0
    stop = 1; % stop if event occurs
    dir = -1; % Detect only events with devdt<0
end

```

Notice that the code now produces two figures – the first figure shows x and y as a function of time, as before (notice that the code stops when $x=y$, as required).

The second plot shows the value of $y-x$. Notice that $y-x$ goes to zero, again, as expected.

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

```

options = odeset('RelTol',0.00001,...
    'Events',@(t,w) event_function(t,w));

```

The second line tells MATLAB that you want to look for a special event, and that a function called 'event_function' 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 [ev, stop, dir]. ***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 [ev, stop, dir]. Here's what the variables do:

1. The value of 'ev' 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. 'ev' will always be some function of time t and the variables stored in the vector w . Here, we are simply interested in finding when $y=x$, so we set 'eventvalue= $y-x$ '.
2. 'stop' tells MATLAB whether or not to continue the calculation if it detects the event. If you set 'stop=1' it will stop; otherwise if you set 'stop=0' it will continue.
3. 'dir' gives you some additional control over event. In general, the variable 'ev' 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. You can tell MATLAB this by assigning the correct value to the variable 'dir' as follows
 - If 'dir=0' MATLAB will detect an event each time 'eventvalue=0'
 - If 'dir=1' MATLAB will respond only when eventvalue=0 and is increasing
 - If 'dir= -1' MATLAB will respond only when eventvalue=0 and is decreasing.

Try changing the code to `dir = +1;` and see what happens. Notice that the calculation still stops when $y=x$, but now $y-x$ is *increasing* when the calculation terminates.

11.8 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 section 13.2. 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

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

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`.

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 $t=20\text{sec}$. Finally, look at the ‘for $i=1:nsteps \dots \text{end}$ ’ 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

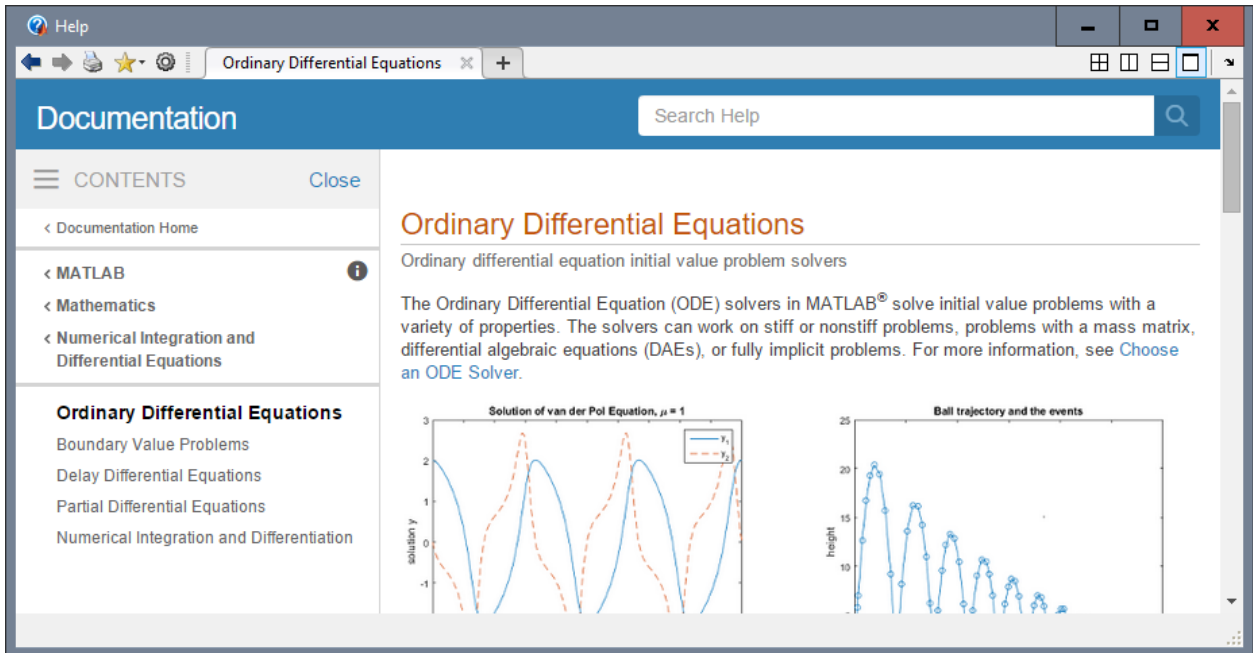
```
>> [t_values,y_values] = my_ode45(@(t,y) simple_ode(t,y),[0,20],0);
>> plot(t_values,y_values)
```

into the MATLAB command window (if you get an error, make sure your `simple_ode` function defined in Section 13.2 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.

11.9 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



12. 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.

12.1 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(@f fofx(x), [-100,100])

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


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

12.2 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(@(x) fofx(x), [3])
end
function f = fofx(x)
% Function to calculate f(x)=(x-5)^2+4
    f = (x-5)^2+4;
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(@(w) zofw(w), [1, 2])
end

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
```

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 .

12.3 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

$$x_{\min} \leq x \leq x_{\max} \quad y_{\min} \leq y \leq y_{\max}$$

or, for something a bit more complicated, we might want to search over a triangular region

$$0 \leq x \leq x_{\max} \quad x + 2y \leq 4$$

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

$$(x - 3)^2 + (y - 8)^2 \leq 10$$

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

$$0 \leq x \leq 10 \quad x + 2y \leq 4 \quad 3x - 10y = 14$$

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 \mathbf{w} . Then, we must write all our constraints in one of 3 forms:

$$\mathbf{w}_{\min} \leq \mathbf{w}$$

$$\mathbf{w} \leq \mathbf{w}_{\max}$$

$$\mathbf{A}\mathbf{w} \leq \mathbf{b}$$

$$\mathbf{C}\mathbf{w} = \mathbf{d}$$

Here, $\mathbf{w}_{\max}, \mathbf{w}_{\min}$ are vectors specifying the maximum and minimum allowable values of \mathbf{w} . \mathbf{A} and \mathbf{C} are constant matrices, and \mathbf{b} and \mathbf{d} are vectors. For our example, we would write

$$\mathbf{w}_{\min} = \begin{bmatrix} 0 \\ -\infty \end{bmatrix} \quad \mathbf{w}_{\max} = \begin{bmatrix} 10 \\ \infty \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 4 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 3 & -10 \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} 14 \end{bmatrix}$$

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(@(w) zofw(w), [1,2],A,b,C,d,wmin,wmax)

end
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
```

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

```
>> constrained_minimization_example
```

```
Local minimum found that satisfies the constraints.
```

```
Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the optimality tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.
```

```
<stopping criteria details>
```

```
optimal_xy =
|
| 4.2500 -0.1250
```

All the big words mean 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(@(var) objective_func(var), initguess, A, b, C, d, lowerbound, upperbound)`

1. The ‘*objective_func*’ 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 \mathbf{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 \mathbf{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 $\mathbf{Ax} \leq \mathbf{b}$, where \mathbf{A} is a matrix, and \mathbf{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 \mathbf{A} and \mathbf{b} .
 4. The arguments \mathbf{C} and \mathbf{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 \mathbf{C} and \mathbf{d} .
 5. The *lowerbound* and *upperbound* variables enforce constraints of the form $\mathbf{w}_{\min} \leq \mathbf{w} \leq \mathbf{w}_{\max}$, where *lowerbound* is the lowest admissible value for the solution, and *upperbound* is the highest admissible value for the solution.

13. 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

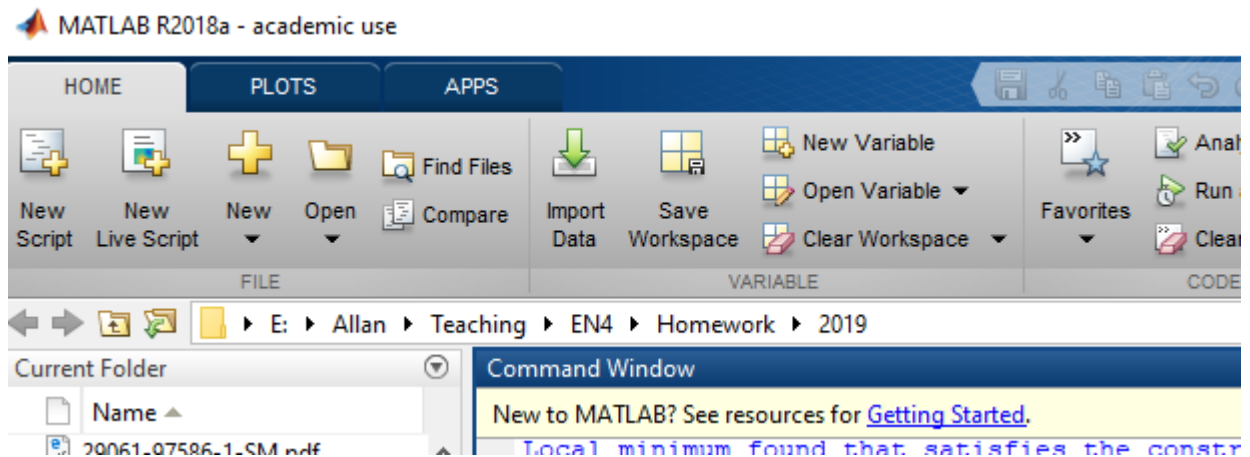
```
>> B = csvread('examplefile1.dat');
>> B
```

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'

Reading and writing datasets as tables: Many engineering calculations store data in tables, or in a spreadsheet. A table in Matlab consists of columns of numbers, character strings, or other data types. MATLAB has two commands: 'readtable' and 'writetable' that will read this sort of dataset, and can handle several file types, including text files, csv files, and Excel spreadsheets. **HEALTH WARNING:** these commands behave differently in MATLAB 2020 and earlier versions – we will assume you are using the latest MATLAB.

You can create a table in MATLAB much the same way that you create a matrix, but the data don't have to be numbers:

```
T = table([1789;1797;1801],...
         [1797;1801;1809],...
         {'John Adams';'Thomas Jefferson';'Aaron Burr'})
T.Properties.VariableNames = {'Start Year','End Year','Vice President'}
T.Properties.RowNames = {'George Washington';'John Adams';'Thomas Jefferson'}
```

You can then save this to an excel file using

```
writetable(T, 'Presidents.xlsx')
```

To read the table again, you can use

```
T2 = readtable('Presidents.xlsx', 'ReadRowNames', true)
```

You can access bits of the table by indexing them, using the same conventions that are used to access parts of a matrix. MATLAB will recognize the data types in the columns, but you can only extract bits of the table that have the same data type. For example,

```
M = T2{:, 1:2}
```

will extract the dates into a matrix M, but

```
M = T2{:, 1:3}
```

will produce an error. You can extract the vice presidents into a cell array VPS with

```
VPS = T2{:, 3}
```

As you see, 'readtable' and 'writetable' are powerful commands – they have huge numbers of optional arguments that give you a lot of control over how data is read and written to your file. It can take some hunting around in the matlab manuals or online to find out how to do what you need, however.

Formatted text files: With a bit of work, you can directly write text files with a prescribed format. 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 = 7662830000;
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 Nov 10 2018 %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 line 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 array'` – 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, 10th and 50th 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...

14. 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.

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',@(t,w) events(t,w));
[t_vals,sol_vals] = ode45(@(t,w) odefunc(t,w,c),...
    [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(0.1); % This waits for a short time between frames
end

end
function dydt = odefunc(t,w,c)
    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
    % for the point when y(2)=0 and y(2) is decreasing.
    y = w(2);
    eventvalue = y;
    stopthecalc = 1;
    eventdirection = -1;
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 discrete points, we put everything about the solution
% in a variable called sol. We can use this variable to reconstruct
% the solution later.
sol = ode45(@(t,w) odefunc(t,w,I0),[0,time],y0,options);
close all
count = 0;
ax_length = 1.2*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)

end

function dwdt = odefunc(t,w,I0)
% 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

```

COPYRIGHT NOTICE: This tutorial is intended for the use of students at Brown University. You are welcome to use the tutorial for your own self-study, but please seek the author's permission before using it for other purposes.

A.F. Bower
School of Engineering
Brown University
February 2021