Getting started with MATLAB for applied mathematics

Dr Gavin M Abernethy

Contents

1 Introduction to MATLAB

MATLAB is a computer package with its own programming language that is very widely used in applied mathematics, theoretical physics and engineering. It has many optional plug-in packages that have specific uses in modelling and simulation, signal processing and control systems.
Students may be provided with a MathWorks account. If you have not accessed your MathWorks account before, do so here. Select "Check for Access", then select your university and enter your university e-mail address. Once you have your MathWorks account, you can either download MATLAB or login to Matlab Online (described below) and activate it.

Important information:

- As well as downloading your own installation of the software, you can also access a browser-based version called "**Matlab Online**" (accessed here). This is the recommended method of using MATLAB for this course, as it avoids the issues of less-powerful computers (or if your work computer has restricted permissions), and can be accessed anywhere with your MathWorks account. It also has drive for storing your scripts online, allowing you to continue your work on any internet-connected computer.

- If you have completed all of this material, and would like a more comprehensive introduction you can also access the self-paced "OnRamp" tutorial here. This is a self-paced tutorial that is developed by the makers of the software themselves and should take you about two hours. If you do this early in the semester, you will find using the software for our purposes very easy.

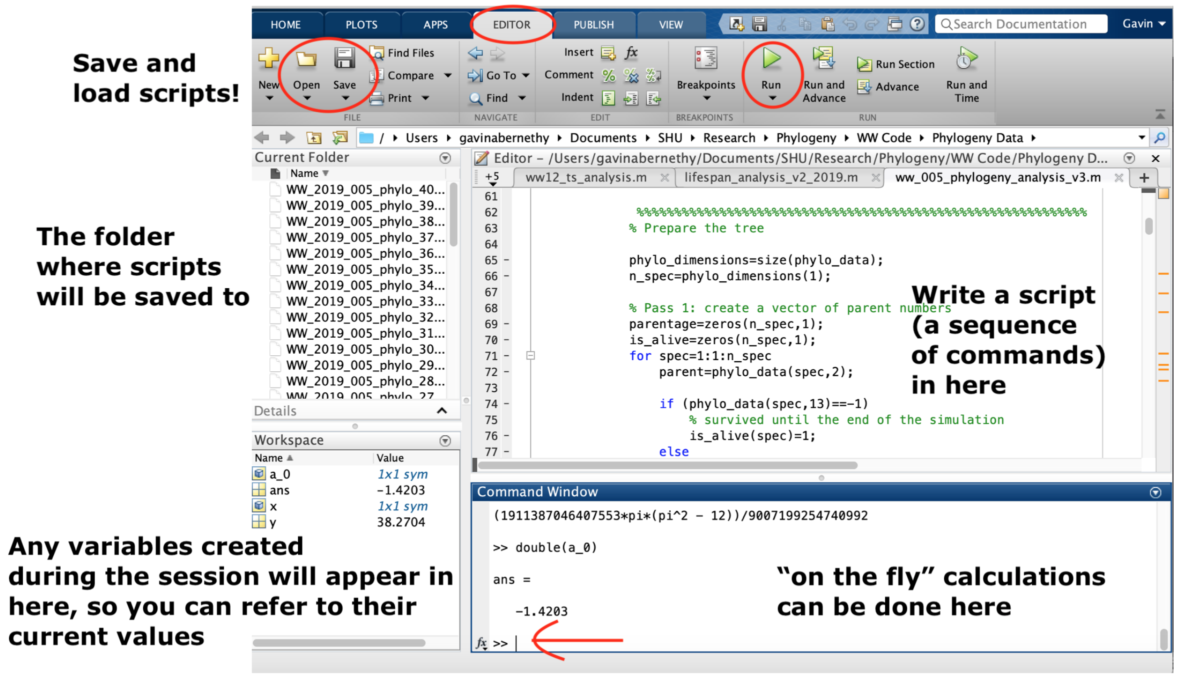
1.1 The MATLAB environment

There are basically three ways to interact with MATLAB:

One way is using scripts (extension .m), as in any other programming language. This means writing a document with a series of commands that can be saved, and when the script is RUN, they will all be executed in sequence. When working at home or on large projects, you should write your commands as a script.

The second way is using the command window. You can type a command after the prompt `>>` and when you press `Enter` that command will execute immediately. This is useful for doing some small tests on the fly, but if you wish you had done something differently you will basically need to type it all out again, and none of what you have typed will be saved for later use. This is why scripts should be used for everything except very trivial tasks.

When you open Matlab for the first time, you will want to ensure that the "Editor" tab is selected, then choose "New > Script". Then program should then look something like the below, although it might be rearranged and everything will be blank:



Name	Value
a_0	1x1 sym
ans	-1.4203
x	1x1 sym
y	38.2704

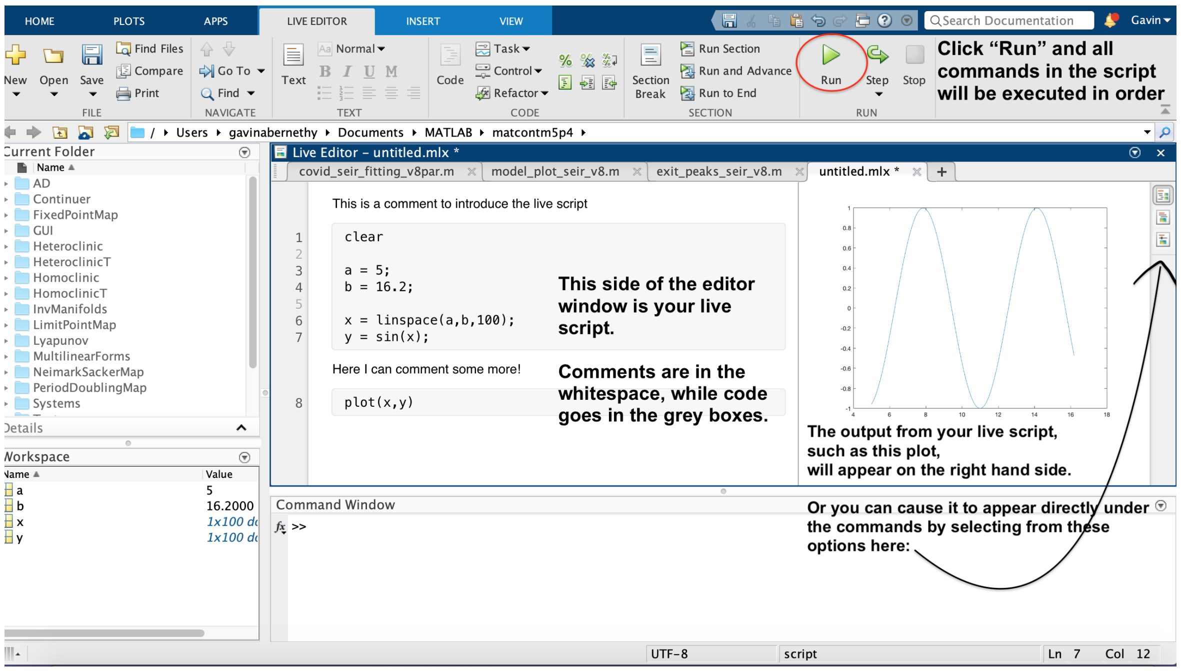
```
          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
         % Prepare the tree

         phylo_dimensions=size(phylo_data);
         n_spec=phylo_dimensions(1);

         % Pass 1: create a vector of parent numbers
         parentage=zeros(n_spec,1);
         is_alive=zeros(n_spec,1);
         for spec=1:1:n_spec
             parent=phylo_data(spec,2);

             if (phylo_data(spec,13)==-1)
                 % survived until the end of the simulation
                 is_alive(spec)=1;
             else
```

```
(1911387046407553*pi*(pi^2 - 12))/9007199254740992

>> double(a_0)

ans =

  -1.4203

>> 
```

The final option are Matlab Live Scripts (extension .mlx). These are quite similar in style to MuPAD (which is being phased out). They use all of the same commands and structure as a regular Matlab script that executes a series of commands, but it shows you "live" all of the outputs at every stage, either to the right or below the command. We will be using these a lot in lectures and the solutions to tutorials as a way of walking you through a series of steps. Unlike regular Matlab scripts, they cannot be opened in a regular text editor (e.g. Sublime Text) and can *only* be viewed and edited using the MATLAB program itself.

For this course, I **strongly recommend** that you produce all of your Matlab-based work in Live Scripts. In previous years, students have found this the most helpful way to store a code while seeing the output simultaneously and being able to easily partition your code with comments (click on "text" and "code" to switch between typing code in the grey boxes and comments in the surrounding whitespace). When you are typing commands in the Live Script, the program will suggest how to complete the command syntax, which may also be beneficial to you.

While in the "Editor" tab, select "New > Matlab Live Script". Then program should then look something like the below, although it might be rearranged and everything will be blank:



Click "Run" and all commands in the script will be executed in order

This is a comment to introduce the live script

```
clear

a = 5;
b = 16.2;

x = linspace(a,b,100);
y = sin(x);
```

Here I can comment some more!

```
plot(x,y)
```

This side of the editor window is your live script.

Comments are in the whitespace, while code goes in the grey boxes.

The output from your live script, such as this plot, will appear on the right hand side.

Or you can cause it to appear directly under the commands by selecting from these options here:

```
>>
```

1.2 Basics

- Use the command `clear` to tell Matlab to "forget" everything it knows about the work you have been doing so far. I usually include this at the beginning of any self-contained scripts, so that there are no conflicting variable names. For example, if I have been doing calculations with a matrix called A and then want to run a script that uses a number that is also called A, Matlab may get confused.

- At the end of any line, you may use a semi-colon (`;`) to suppress the output, meaning that it will not be printed to the command window. In a regular Matlab script you should do this by default unless you specifically wish to see the result of a particular command. However it is not necessary when using Live scripts.

- Use the percentage symbol (%) to make a comment in a script. It is good programming practice to include this frequently within your script, to make clear the structure (rather like paragraphs and subheadings in a written document) and to explain to the reader (and yourself!) the purpose of any non-trivial line of code.

- On the scroll bar to the right of the script window, you may see orange or red horizontal bars. These are warnings that there are problems with your code as it is currently written on the line indicated by the position of the bar. Red means a problem that will actually halt the code, such as an incomplete or unrecognised command, while orange is just a warning such as a variable that has been defined but never actually used for anything.

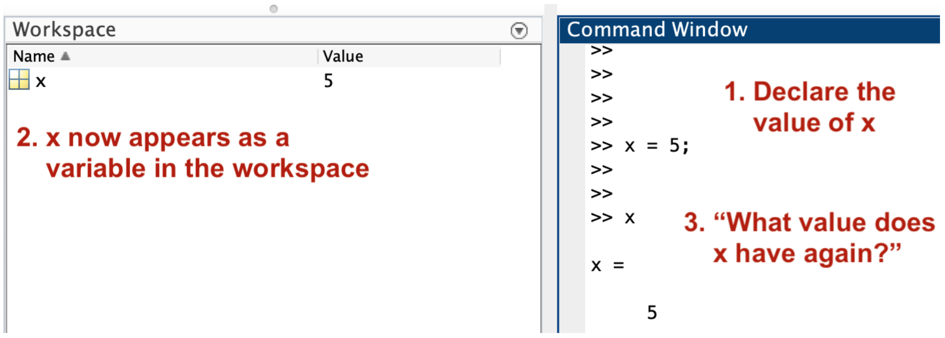
For example, the following very simple script would allow you to change the parameters a, b, c of a quadratic equation, and calculate the output $y = ax^2 + bx + c$ for some input x:

```
% This is a script to calculate a quadratic function
% Author: GM Abernethy
% Date: 13/08/2020

% First clear any previously-assigned variables:
clear;

% Parameters:
a = 1;
b = 3;
c = -2

% Set value of x:
x = 17;

% Calculate the quadratic y = ax^2 + bx + c
y = a*x.^2 + b*x + c
```

Note the use of comments, and that the final line lacks a semi-colon, so that when this script is run the value of y will be printed to the screen.

1.3 Numeric and symbolic variables

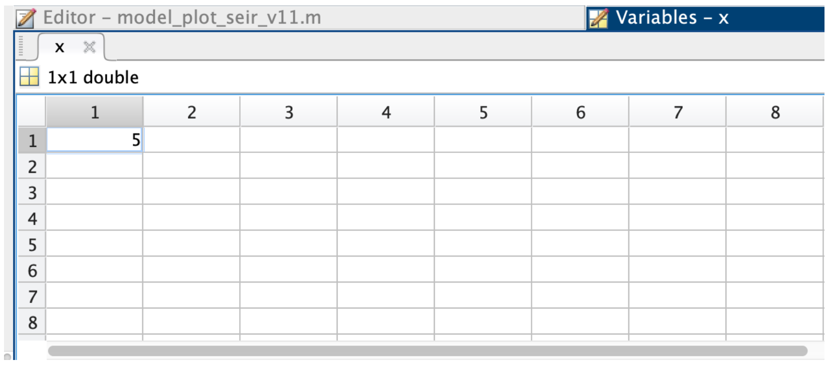
A **variable** is a quantity with an alphabetic name that you can refer to, and which can be used to store information in MATLAB. Before use, they need to be "declared", meaning that you tell MATLAB that, for example, x is a variable and that it has a value of 5. To do this, we would just type `x = 5` in the command window, or as a line in our script before x needs to be used. You will notice that x appears in the "Workspace" (see the figure). To check the value currently assigned to x, you can type "x" in the command window:


Workspace
Command Window
1. Declare the value of x
2. x now appears as a variable in the workspace
3. "What value does x have again?"

Name	Value
x	5

```
>>
>>
>>
>>
>> x = 5;
>>
>>
>> x

x =

     5
```

Alternatively, you can double-click on x in the workspace, which will open the "Variables" window.



	1	2	3	4	5	6	7	8
1	5							
2								
3								
4								
5								
6								
7								
8								

This is more useful for when you want to inspect a variable containing a large amount of data (for example, a 100×5 data set that has been imported from a spreadsheet).

What we have just described is an example of using specifically a "numeric variable". This is one of the two different types of data that we will use in Matlab: **numeric** and **symbolic** variables.

Numeric variables are, quite simply, variables that hold a decimal number. Matlab can do lots of calculations with them very quickly, and they are accurate to a high number of decimal places. Matlab can only understand what to do with a numeric variable *after* you have assigned it a value, for example by typing `T = 4` Then you can refer to T in further calculations, and Matlab will know that it means "4".

Symbolic variables are needed when you want Matlab to reason about expressions and formulae, such as integration or Fourier series. They need to be declared as symbolic, but not necessarily assigned a value, before using them. For example, if you wish to declare that y varies as a function of x (in particular, $y = \sin(x)$), you could write:

`syms x` This tells Matlab that x is a symbolic variable.
`y = sin(x)` Now Matlab knows that y depends on x in this way, and is also a symbolic variable.

If we were then to write `x=10; eval(y)` Matlab would then think of x as a numeric variable equal to 10, and would return the value of y based on this.

1.3.1 Commands for symbolic variables

When Matlab is using symbolic variables (for example when working with integration or matrices), it will never make any approximation. So if your answer was $17\pi - \frac{1}{3}$, and the calculation had involved symbolic computation, Matlab would give you an answer stated as "$17\pi - \frac{1}{3}$", as there is no way to state it any more simply without making *some* approximation. However, you may wish to obtain an approximate decimal answer (it will still be to a very high degree of precision!), so you can force Matlab to give you this using `double( )`. This converts a symbolic variable to a double-precision numeric variable.

Example

$$y = \int_0^\pi x \ \mathrm{d}x \quad = \quad \frac{1}{2}\pi^2 \quad = \quad 4.9348\ldots$$

```
syms x
y = int( x, x, 0 , pi )
double(y)
```

If Matlab gives you an answer in symbolic variables that looks very complex, it might sometimes be possible to simply or factorise it further. You can try this using the command `simplify( )`. For example, to factorise the x out of $x^2 + 13x$:

```
syms x
simplify( x^2 + 13*x )
```

Matlab can also show you symbolic expressions in a "nice" display format using the command `pretty( )`. Compare the outputs of the following commands:

```
syms x
x^(-0.5)
pretty( x^(-0.5) )
```

1.3.2 Converting numeric variables to symbolic

Whilst `double( )` converts a symbolic variable to a numeric variable, so that we can obtain a decimal approximation, we can also use `sym( )` to convert an existing numeric variable to a symbolic variable.

Compare the outputs of the following commands:

```
x = 1/3
y = sym(1/3)
```

As we can see, if we declare 1/3 in MATLAB, this is actually stored as a high precision numeric approximation of 0.333333333. Whilst to make clear that we want to work with the precise fraction, we can use `sym(1/3)`

2 General calculations

2.1 Algebra

Like a calculator, Matlab will use the order of operations rules (you may know this as “BODMAS”, “BIDMAS”, or “PEDMAS”) when deciding what order in which to evaluate operations (multiplication, addition, etc.) so you will need to think carefully about where brackets are required!

Addition and subtraction use `+` and `-` as you would expect.

To multiply two variables use `*` as is the case in most programming languages.

Raise numbers to a power using `.^`

Square roots, sine, cosine, logarithms and exponentials are written as functions, with the input in brackets: `sqrt( )` `sin( )` `cos( )` `ln( )` `exp( )`

Example 1:

$$y = 6(\pi - 27^3) + \frac{5}{19} + 42\pi^{-2} - \cos(4\pi)$$

```
y = 6*( pi - 27.^3 ) + 5/19  + 42*pi.^(-2) - cos(4*pi)
```

Example 2:

$$y = \frac{13x - \pi}{(14 + x)(2 - \cos(x))}$$

```
y = ( 13*x - pi )/( (14 + x )*(2 - cos(x) ) )
```

Example 3:

$$y = \sqrt{2x} - \mathrm{e}^{-3x}$$

```
y = sqrt( 2*x ) - exp( -3*x )
```

2.2 Integration

Integration uses the `int` command. This takes several arguments: the function to be integrated, the symbolic variable you are integrating with respect to, and (optionally) the lower and upper limits for definite integration. The variable (usally t or x) must be declared as symbolic first.

Example 1:

$$\int 3x + 1 \ \mathrm{d}x$$

```
syms x
int( 3*x+1 , x )
```

Example 2:

$$\int_0^{5\pi} \sin(2t) \ \mathrm{d}t$$

```
syms t
int( sin(2*t) , t , 0 , 5*pi )
```

In this second example, we added two arguments: the lower and upper bounds for t. The order of these is important - the limits *must* come after the function and then the variable.

2.3 Differentiation

Differentiation uses the `diff` command. As with integration, this requires two arguements: the function to be differentiated, and then the symbolic variable you are differentiating the function with respect to.

Make sure that you declare the necessary symbolic functions beforehand using `syms` , as shown in the examples.

Example 1:

$$\frac{\mathrm{d}}{\mathrm{d}x}\big(\sin(2x)\big)$$

```
syms x
diff( sin(2*x), x )
```

This can be combined with the `solve` function to help us determine the location of stationary points.

Example 2:

Let's say we wanted to determine the values of x where $y = \sin(2x)$ has a stationary point (that is, the gradient is equal to zero):

```
syms x
solve( diff( sin(2*x), x ) == 0, x )
```

Be aware that:

- When asking MATLAB to check an equality, as in this case where we are asking *when* is the derivative equal to zero (rather than *telling* MATLAB to assign a value of zero to a variable), we need to use `==` rather than a single equality symbol.
- There are multiple solutions (infinitely-many, in fact, due to the periodicity of sin and cos) and MATLAB has only returned the simplest one of these.

2.3.1 Higher order derivatives

To obtain a higher-order derivative, such as the second derivative, you can simply nest multiple calls of the `diff` function so that the output of differentiating once is then passed as the input to the next call of the command. For example, to obtain $\frac{\mathrm{d}^2y}{\mathrm{d}x^2}$ where

$y = x^3$:

```
syms x
y = x^3
diff( diff( y, x), x)
```

However, this is a bit inelegant if we frequently need to use higher-order derivatives such as the fourth or fifth derivative. As an alternative to nesting multiple copies of the command, we can pass the order as an argument. So instead of the above, we could instead write:

```
syms x
y = x^3
diff( y, x), x, 2)
```

and so for a different example, we could easily obtain the 6^{th} derivative of $\sin(3x)$ w.r.t. x using:

```
syms x
y = sin3*x)
diff( y, x), x, 6)
```

2.3.2 Partial differentiation

Partial differentiation is actually the same as regular differentiation in MATLAB, using the `diff` command with two arguments. The only difference is that you will need to remember to declare all variables as symbolic first. For example:

$$\frac{\partial}{\partial x}\big(xy^2 + 3y\sin(x)\big)$$

```
syms x y
diff( x*y*y+3*y*sin(x) , x )
```

3 Visualising data

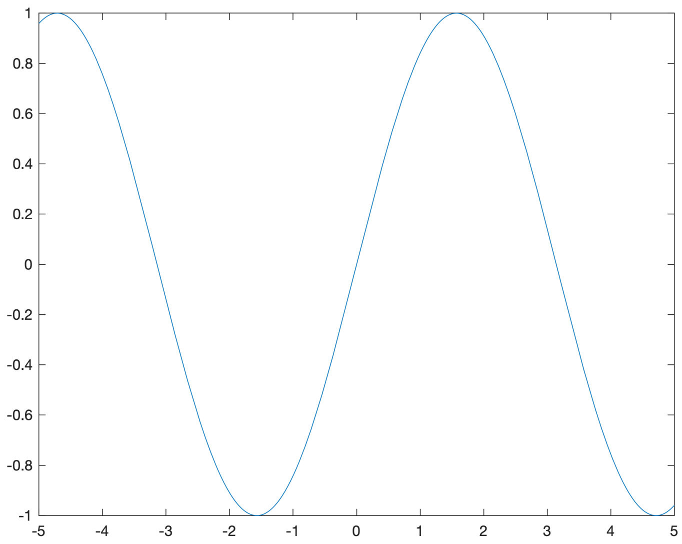
3.1 Curve plotting

MATLAB has many different commands for seemingly-basic functions such as curve-plotting, that all work in slightly different ways and situations.

3.1.1 Symbolic (recommended for most of this module)

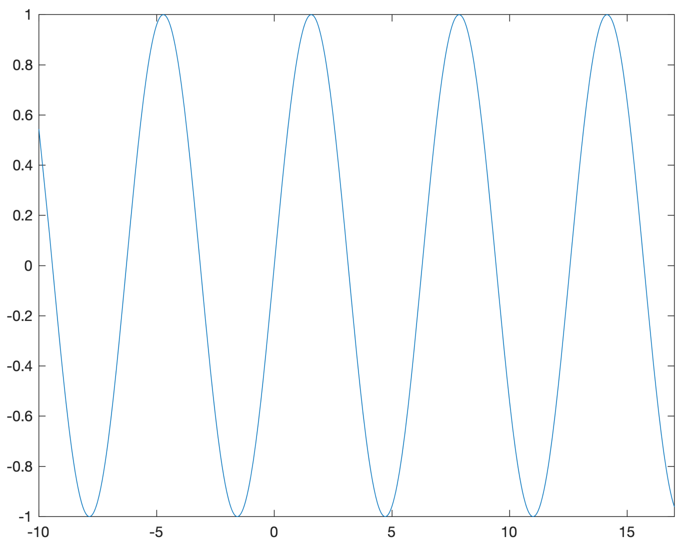
For visualising a simple function, such as a sine wave or a Heaviside function, we can declare it as a symbolic variable and then use `fplot` to visualise it over the default range of x, which is $[-5, 5]$:

```
syms x
y = sin(x)
fplot(x,y)
```



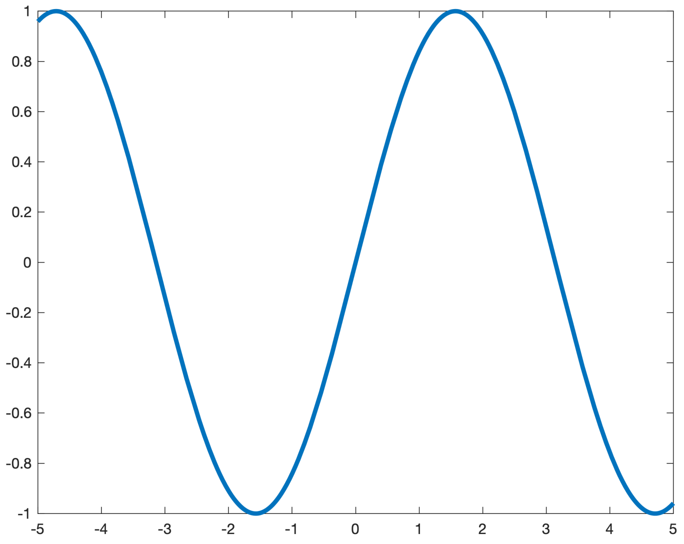
If we want to draw the graph for a different range, say $-10 < x < 17$, this is implemented as an optional argument (extra terms separated by commas) when the `fplot` command is used:

```
syms x
y = sin(x)
fplot(x,y,[-10, 17])
```



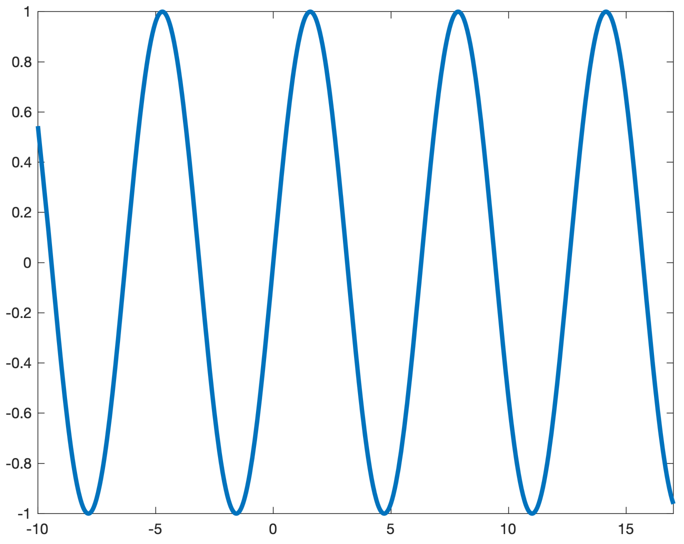
These curves are a bit thin and hard to see. Fortunately, almost everything can be customised in MATLAB, so a clear improvement would be to draw a thicker curve when plotting. Like the range of x, this information is included within the `fplot` command:

```
syms x
y = sin(x)
fplot(x,y,'LineWidth',3)
```



and we can combine multiple of these optional arguments - so to change *both* the range of x and the curve thickness:

```
syms x
y = sin(x)
fplot(x,y,[-10, 17],'LineWidth',3)
```



3.1.2 Numeric

This is not really needed for this module, but it is much more powerful for real applications in research.

For plotting curves from numeric data sets, or anything that isn't necessarily a pre-defined symbolic function, we can use the standard `plot` command. However, this requires two arguments: one vector containing all the x-coordinates of the points that you wish to join up, and a second vector of equal length containing the corresponding y-values.

For example, to plot the graph of $y = \sin(x)$ between $x = 0$ and $x = 5$:

```
x = linspace(0,5,1000)
```

This creates a vector of 1000 evenly spaced values between 0 and 5. These will be the x-coordinates of the points that `plot` will then join up.

```
y = sin(x)
plot(x,y)
```

As with the `fplot` command, you can include optional arguments in the `plot` command to customise the graph, such as changing the line thickness:

```
plot(x,y,'LineWidth',5)
```

3.2 Scatter plots

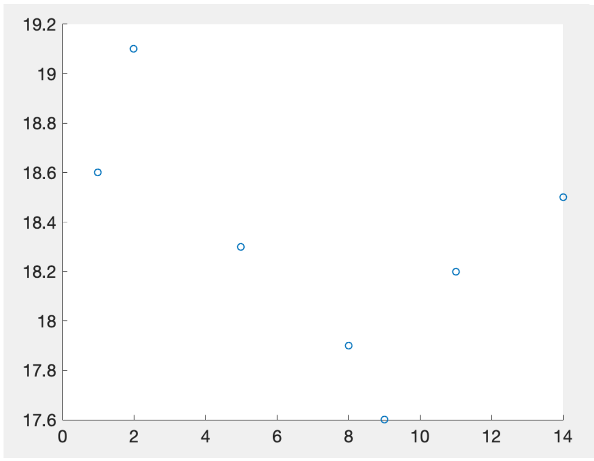
If we are working with a real, collected data set of discrete points, it would not be appropriate to draw a curve joining the actual data up.

Instead, we would probably want to draw a scatter plot showing the actual points (and then perhaps compare this with a curve of a fitted function). We can achieve this with the `scatter` function.

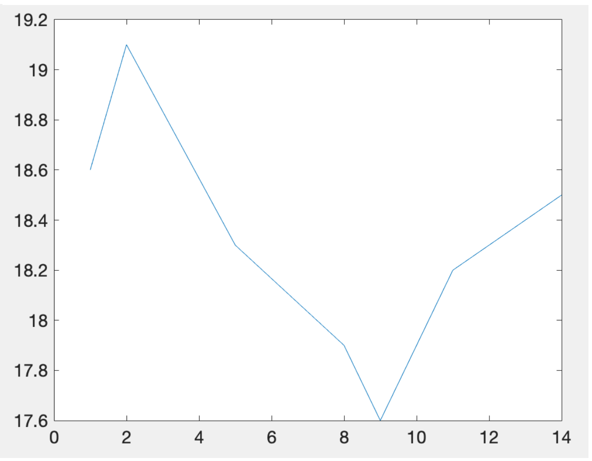
For example, imagine we had recorded the temperature in the Sheaf building on seven non-consecutive days in a two-week period. If we store the days in a column vector t (for time), and the recorded temperature in a column vector T (also of length seven):

```
t = [ 1  2  5  8  9  11  14 ]
T = [ 18.6  19.1  18.3  17.9  17.6  18.2  18.5 ]
scatter( t, T )
```

We can add extra arguments to the `scatter` command to change the size, shape and colour of the points.



(a) scatter(t,T)



(b) plot(t,T)

3.3 Labelling axes

After drawing an image with `fplot` , `plot` , `scatter` or some other command, we can add a title and labels to the axes using additional commands.

To label the x-axis “Time (days)” and the y-axis “Temperature (celsius)”, and add a title to our scatter plot:

```
xlabel(‘Time (days)’)
xlabel(‘Temperature (celsius)’)
title(‘Temperature in Sheaf during a two-week period’)
```

Note that you must include the quotation marks.

3.4 Changing the limits in view

There are many other customisation options for images produced in Matlab. Probably the most useful one is for when you want to change the limits on the axes that are in view of a graph *that has already been plotted*.

To do this, follow-up with the commands `xlim` and/or `ylim` as appropriate.

For example, if we wanted to zoom in on the graph between $x = 1$ and $x = 2$, we would use the command:

```
xlim([1 2])
```

and similarly for the y-axis using `ylim`

Note that this is only useful if the graph has already been plotted within the range that you wish to see. If this is not the case, you will need to redraw the graph.

Let's put everything together to draw a nice graph of the function $y = \sin(x)$, controlling the range, line thickness and labelling the axes. Rather than having the graph fill the window by default, we'll use `ylim` to zoom out and get a better view. First, we'll increase the font size so our axes labels are easier to read:

```
syms x
y = sin(x)
```

Now we set the font size, then plot the graph over our desired range of x:

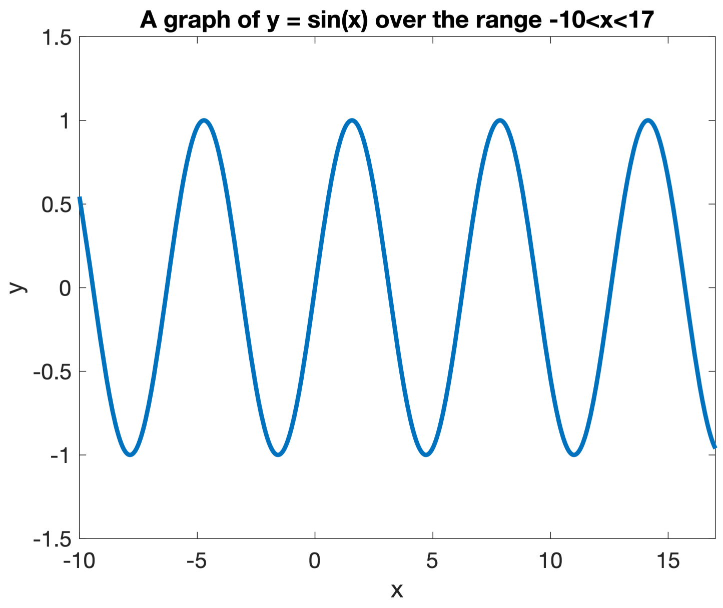
```
set(0,'DefaultAxesFontSize',15)
fplot(x,y,[-10,17],'LineWidth',3)
```

Then add the title and axis labels:

```
xlabel('x')
ylabel('y')
title('A graph of y = sin(x) over the range -10<x<17')
```

Finally, use `ylim` to zoom out the y-axis:

```
ylim([-1.5, 1.5])
```



3.5 Plotting multiple plots on one image

You can also use the command `hold on;` to plot multiple functions on the same image, as otherwise each new use of `plot` will replace the previous graph.

For example, say we have two different sets of results $y1$ and $y2$ for a given set of x-values. To plot them on the same graph and add a legend:

```
plot( x , y1 )
hold on
plot( x , y2 )
legend('Data set y1', 'Data set y2')
```

You must then use `hold off` if you want to subsequently plot a different graph altogether.

For example, let's say we want to plot both $\sin(x)$ and $\cos(x)$ on the same graph, each over the range $-10 < x < 17$, but with different line thicknesses:

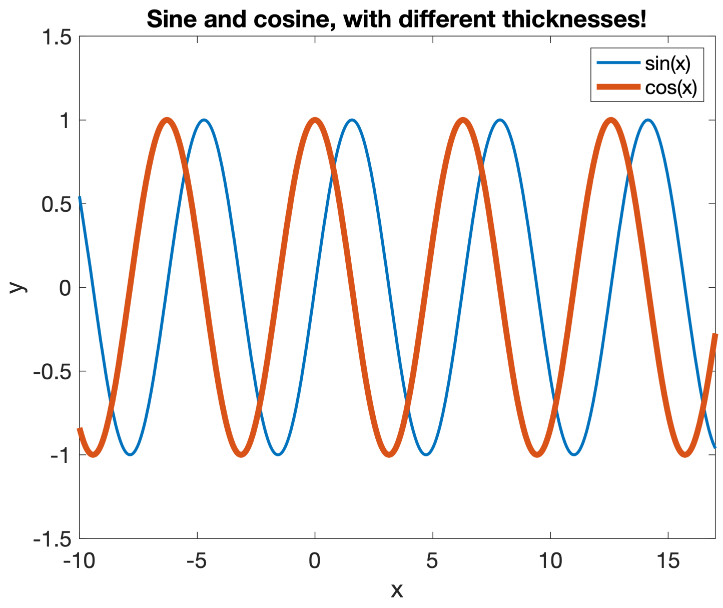
```
syms x
y1 = sin(x)
y2 = cos(x)
```

Now we plot the two graphs on the same axes:

```
set(0,'DefaultAxesFontSize',15)
fplot(x,y1,[-10,17],'LineWidth',2)
hold on
fplot(x,y2,[-10,17],'LineWidth',4)
```

Then add the legend, title and axis labels:

```
legend('sin(x)','cos(x)')
xlabel('x')
ylabel('y')
title('Sine and cosine, with different thicknesses')
ylim([-1.5, 1.5])
hold off
```



4 Fourier Series

4.1 Piecewise functions

Fourier series often requires constructing piecewise-continuous functions. These can be made from a combination of Heaviside step functions.

For example, a signal $f(t)$ that is zero for $t < 0$ and has a constant value of 4 for $t > 0$ can be described as:

$$f(t) = 4H(t)$$

And this function can be declared and plotted in Matlab:

```
syms t
f = 4*heaviside(t)
fplot( t, f )
```

4.2 Fourier series

To find the Fourier series of a function $f(t)$, let's say up to the 4^{th} partial sum (up to the terms including a_4 and b_4), we will need to:

- Declare t as a symbolic variable: `syms t`
- Perform the interal to calculate a_0
- Perform a series of integrals to calculate a_1, a_2, a_3, a_4
- Perform a series of integrals to calculate b_1, b_2, b_3, b_4
- Assemble the approximation of the Fourier series by adding together:

```
FourierApprox = a0/2 +  a1*cos( w*t)  +  b1*sin(w*t)
                     +  a2*cos(2*w*t) +  b2*sin(2*w*t)
                     +  a3*cos(3*w*t) +  b3*sin(3*w*t)
                     +  a4*cos(4*w*t) +  b4*sin(4*w*t)
```

- Plot both the actual function and the approximate Fourier series on the same graph.

See earlier in this document for a guide to integration and plotting multiple graphs.

Once these are plotted, we can see how well the approximate Fourier series fits the true function. If it is a poor fit, we may need to build a more accurate Fourier partial sum by adding in some more terms (a_5, b_5, a_6, b_6, etc.) and plotting again.

4.3 Fourier transform of functions

We can perform a Fourier transform of a function $f(t)$ from the time domain to the frequency domain using the `fourier` command, which has three arguments:

- The first argument is the function f.
- The second is the variable that f is a function of (in this case, t).
- The final argument is the variable we are transforming to,which is frequency. This uses the variable ω, but we'll just use w as a stand-in.

Thus, to store the Fourier transform of $f(t) = H(t) - H(t - \pi)$ and as a variable F:

```
syms t w
f = heaviside(t) - heaviside(t - pi)
F = fourier(f, t, w)
```

4.4 Fourier transform of discrete data

We will go through the process of discrete Fourier transforms in Lecture 11. In this document we will just go over the new Matlab commands requried during the procedure.

To take the discrete Fourier transform of our sampled data, stored in a column vector f, we use the Fast Fourier Transform algorithm:

```
F = fft(f)
```

The result will be a vector F of complex numbers, consisting of a real part and an imaginary part. We only want the magnitudes (or "absolute values") of these numbers, which means their distance from the origin. To store these in a column vector m:

```
m = abs(F)
```

5 MATLAB and Matrices

5.1 Declaring matrices

To name and store a matrix in Matlab, use square brackets and write the list of elements in each row from left to right, starting with the top row. Separate the elements in each row by a space, and separate the rows themselves with a semicolon.

To declare the following matrix:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

we write:

```
A = [1 2 3; 4 5 6];
```

We can also create an empty matrix containing only zeros, with a particular set of dimensions. This is sometimes useful for preparing to store large data sets incrementally in the matrix. For example, to create an empty 2×7 matrix F:

```
F = zeros(2,7)
```

5.2 Matrix algebra

Addition, subtraction, scalar multiplication and matrix multiplication all work very simply in the way you would expect.

```
A = [1 2; 3 -4];

B = [4 -6; -1 2]

C = A + B          (matrix addition)

D = A - B          (matrix subtraction)

E =  5*A           (scalar multiplication)

F = A*B            (matrix multiplication)
```

Don't forget that the order of matrix multiplication is very important!

5.3 Determinant, Transpose and Inverse

Simple MATLAB commands exist for most important matrix operations.

```
A = [1 2; 3 -4];          (declares the matrix A)

B = transpose(A);         (matrix B is the transpose of A)

C = inv(A);               (C is the inverse matrix of A)

d = det(A);               (obtain the determinant of A)
```

Remember that the inverse does not exist for a non-square matrix. If you attempt this, you will receive the error:

```
Error using inv. Matrix must be square.
```

and the script will fail. However, you must also remember that the inverse of a square matrix does not exist if the determinant is zero, and you should always check this before attempting to calculate the inverse. If you ask Matlab for the inverse of a matrix with zero determinant it *will not fail*, and you will simply receive the message:

```
Warning: Matrix is singular to working precision.
```

It will be up to you to realise that this means you should not proceed.

5.4 Eigenvalues and Eigenvectors

MATLAB can automatically determine the eigenvalue and eigenvector pairs of a square matrix for you, using the Symbolic Math Toolbox:

```
A = [1 2; 3 -4];

B = sym(A);

[vecA,valA] = eig(B);
```

This first creates a symbolic version B of the matrix (which Matlab needs to do in order to handle more abstract mathematical operations) and then produces two matrices, $valA$ contains the eigenvalues on the diagonal (with zeros elsewhere) and $vecA$ contains the eigenvectors in each column. The ordering between the two corresponds, so the first column of $vecA$ is the eigenvector corresponding to the eigenvalue in the first diagonal entry of $valA$.

To find a unit vector, we need to first obtain the magnitude of the vector, and then divide the vector by this magnitude:

```
X = [3; 17];          (declare a vector X)

mag = norm(X);          (obtain the magnitude of X and store it as mag)

unitX = X/mag;          (obtain the unit vector and store it as unitX)
```

5.5 Solving simultaneous equations

To use this method does not require any additional commands, but a combination of what we have learned. Let's revisit Example 1 from Section 4.2 and carry out the method in Matlab:

$$5x + 2y = 10$$

$$4x - 3y = 14$$

```
A = [5 2; 4 -3];

B = [10; 14];

X = inv(A)*B
```

Note that this will provide decimal answers. To obtain the precise fractions that we found when solving by hand, we simply ask Matlab to convert the answer to a symbolic variable:

```
sym(X)
```

5.6 Accessing the elements of a matrix

An element (i.e. one of the entries) of a matrix M can be referred to using the format $M(row, column)$.

For example, consider the matrix:

$$A = \begin{pmatrix} 16 & 3 & 2 & 12 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

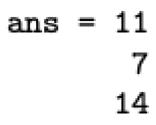
We can declare this matrix and then access the element $A_{3,2} = 6$ using:

```
A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]

A(3,2)
```

We can also access a range of values using a colon, so $M(i : j, k)$ refers to the rows i to j of the k^{th} column. For example:

```
A(2:4,3)
```

accesses the 2nd, 3rd and 4th entries of the 3rd column of A and gives the output:

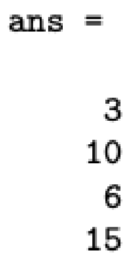


```
ans = 11
       7
      14
```

Or we can leave the colon by itself to refer to an entire row or entire column. For example:

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

accesses the entire second column of matrix A and gives the output:



```
ans =

     3
    10
     6
    15
```

You can access a range of rows and columns simultaneously, so for example you could extract the elements of the first two rows and the first two columns of A and assign them to a new 2×2 matrix B using:

```
B = A(1:2, 1:2)
```