In this lab you will look at using the Symbolic Mathematics toolbox and a number of the features it offers. You'll then undertake an exercise using the curve fitting toolbox.

Exercise 9.1 – Symbolic Mathematics.

We are going to have a look at using the Symbolic Mathematics toolbox that comes with MATLAB. The toolbox is useful as it allows us to manipulate symbolic expressions and equations simply, including the ability to integrate, differentiate, plot and solve equations. You can either open a new script file if you want to save what you've done or work in the command line.

Follow the instructions below:

1. We are going to define a number of symbolic variables, a, b, c and x. To do this we can either use the sym function in the format sym('x') but will need to do this four times or if we use the syms command we can create all the variables in one go: syms a b c x.
Check the workspace and you should see that there are now four variables with the value set at 1x1 sym.

2. We now want to create a symbolic expression. As our variables are already defined we can simply type out the equation we want. In this case we are going to create a quadratic equation. Type: y = a*x^2+b*x+c
You should see that a new variable has appeared in your workspace, again y is a 1x1 sym.
If we hadn't already defined a, b, c and x as variables we could have created the expression using the sym function, i.e. `y = sym('a*x^2+b*x+c')`. If you do this method, without previously defining the variables, only one sym variable will appear in the workspace, y. If all variables had already been defined you could remove the single quotes i.e. `y = sym(a*x^2+b*x+c)`.
Once we've created symbolic expression, we can use standard algebraic expressions to manipulate them i.e. `*, /, +, -` and `^`. Have a go at creating some new symbolic expressions by creating a new expression: `z = a*x`. Now try the following: `q = y*z, r = z/y, s = z+c, t = y-x^2`.

3. y above is classed as a symbolic expression, we can also create symbolic equations using the sym function. E.g. `equation1 = sym('a*x^2+b*x+c = 0')` Notice the difference between the two.
Again a symbolic variable will appear in the workspace called equation1. Generally we will work with expressions rather than equations although it will depend on what we are trying to achieve.

4. There are a number of functions available to manipulate symbolic expressions.
Create a new expression:

i. `z = sym('(x-4)*(x-6)')`
Now try the following functions:

```
z = expand(z)
z = factor(z)
z = collect(z)
```

Note how these functions work.

ii. Another useful functions is simplify, which uses a set of rules that define what constitutes a simple format and converts your equation according to such rules.

Try `z = simplify(z).`

Now enter a symbolic equation `z = sym('x^2 = (x-4)*(x-6)')`

Now use `simplify(z).`

Hopefully you are starting to see some benefits to this toolbox.

5. We are going to look at the example of an object falling due to gravity now.

i. Clear your workspace. Start by creating the following symbolic variables: g, t, v0 and h0 to represent acceleration due to gravity, time (s), initial velocity (m/s) and initial height (m) respectively. We will start with the simple equation a = g. Type in `a = g` to create the expression.

ii. We now want to integrate 'a' with respect to 't' to find the velocity, 'v'. We will now use the int function. If we use v = int(a) then 'a' will be integrated with respect to 'g' as that is the only symbolic variable in the expression which isn't what we want. If we want to integrate with respect to 't' we need to add a second input argument** v = int(a, t). You'll note that the result does not contain a constant of integration. This is always the case with the int function. However we can add one ourselves.
Use `v = v + v0.`

iii. Again, to find height we want to integrate 'v' w.r.t. 't'. Do this and as before add a constant h0.
We should now have the equation for the height of the object at time t, if we were given the initial velocity and height.
The above integration finds the indefinite integral, by adding additional arguments we can find the definite integral, i.e. v = int(a, t, 0, 4), where we are integrating 'a' w.r.t. 't' between t=0 and t=4, which in this case would give us v = 4*g.

** Note that if you have an expression that contains a variable 'x', int will automatically integrate w.r.t. 'x' so the second input argument is not required

6. If we had been given the equation for h, and were asked to find the equation for acceleration we could use the diff function. As with the int function we need to specify the variable we are going to differentiate w.r.t. unless that variable is 'x'.

i. Use `v = diff(h,t)` to find the equation for velocity. It should match the previous solution.

ii. Use diff again to get 'a'.

iii. To undertake the double differentiation in one step you can either nest a diff function within another, i.e. `a = diff(diff(h,t),t)` or add a further input argument to diff: `a = diff(h,t,2),` the 2 specifying the degree of the derivative.

7. Often, once we've manipulated our equations, we may want to either replace the symbolic variables with numeric values to find a particular solution or we may want to solve the equation or set of equations. Firstly we'll look at the subs function.

i. Suppose we want to find the height of the falling object after 2 seconds falling under gravity after starting at an initial velocity of +5m/s and an initial height of 20m. We can substitute these values into the symbolic expression, h. We can make a single substitution by using the function subs. To replace g with --9.81 we will use `subs(h, g, -9.81)` where the first argument is the expression or equation into which we are substituting, the second the variable we are replacing and the third, the value it is being replaced with (this can be either a numeric value or another symbolic variable). You'll see that g has been substituted into the equation although MATLAB has rearranged the equation slightly.

ii. If we want to substitute more than one variable at a time we can use the following `ht2 = subs(h, {g, t, v0, h0}, {-9.81, 2, 5, 10}).` You'll see this gives you the solution for the height at t=2.

8. Suppose we want to find the time at which the object will hit the ground, where h = 0. We can use the solve function to rearrange the equation for h to find t. The best way to do this is to define it as a symbolic equation.

i. Clear the workspace and define 5 symbolic variables, h, t, g, v0 and h0.

ii. Now define the symbolic equation using sym function: `equation = sym('h = (g*t^2)/2+v0*t + h0')`

iii. To solve this equation for 't' we can use the solve function, t = solve(equation, t). You'll notice there are two results as the equation is a quadratic, in our case only one of these solutions will be correct as one will give a negative time.

iv. To find the result, we can now use the subs function again, replacing h0 = 20, v0 = 5, g = -9.81 and h = 0. Note you'll get two solutions, with the positive value being the correct one. You could have indexed the correct equation of t by using t(1) in the subs function.

solve will also work for solving sets of simultaneous equations as an alternative to using matrices. Define each equation as a symbolic equation and then use solve(equation1, equation2, equation3) E.g.

```
equation1 = sym('2*x-1*y+3*z = 5');
equation2 = sym('-2*x+6*y+4*z = 20');
equation3 = sym('-3*x+3*y-3*z = -3');
[x,y,z] = solve(equation1, equation2, equation3)
```

Note that x, y and z are still symbolic variables in the workspace, to see them as numerics use double(x) etc…

9. Another feature of the symbolic toolbox is the ability to create plots of symbolic expressions. Let's plot the height of the object falling under gravity from time 0 to 2.59sec when it hits the ground. To do this we can use the ezplot function which works the same as the standard plot but rather than plotting data from a vector it will plot the result of a symbolic expression.

i. Let's return to defining our expression for the height of the object putting in the constants ourselves.
```
h = sym('(-9.81*t^2)/2+5*t+20')
```
ii. Now use the ezplot function which takes the symbolic expression as the first input argument and the range as a second argument, if no second argument is used, range defaults to -2*pi to 2*pi.
```
ezplot(h, [0 2.59])
```
You should now see the plot of the height of the object between time = 0 and 2.59s
You can then add labels etc as normal.

There are also versions of surface, mesh and contour plots that will work for symbolic expressions (ezsurf, ezmesh etc.). It's worth having a look at them. Use MATLAB help to provide a starting point.

This has been an introduction to the Symbolic Mathematics toolbox in MATLAB including a number of the basic functions it has. It is worth looking further into this toolbox as it could provide a useful tool during the rest of your degree.

If you have time, ensure the spirograph.m and spirograph.fig are in your working directory. Have a look at the code in spirograph.m, it uses the symbolic math toolkit to generate Spirograph images. Note that as it uses ezplot to plot the symbolic expressions, it is limited to 300 data points which limits the representation of the curves. Run the file and play about with the parameters to make a lovely picture.

End of exercise 9.1

Exercise 9.2 – Basic Curve Fitting for bridge members.

We will undertake an exercise using the Polyfit function. You'll work on the data from previous year bridge member testing as part of MECH1206. This includes the breaking loads for the tensile and compressive members of the bridge. You will create lines of best fit for those data sets, giving you the equations that relate the breaking loads to member dimensions.
We'll then use polyval to evaluate the polynomial and then plot the results. Download the tensile_test.csv, comp_test_6x10.csv and comp_test_10x10.csv from the VLE and save to your working directory. These contain the breaking loads and associated dimensions for all the tests you undertook. For the tensile tests the first column is member width in mm and the second is the breaking loads in newtons. For the compression tests the first column is the length of the member in mm and the second is the breaking loads in newtons.

Follow the instructions below:

1. Create a new .m file and save it as strength_fit.m. Clear the workspace and command window. Load the data from each CSV file, ignoring the header. You should have three matrices of data, *tensile_data*, *comp_6x10_data* and *comp_10x10_data*.

2. Separate data into the two vectors, for tensile data create the vectors *width* and *tensile_breaking*. Width will be the first column in *tensile_data* and *tensile_breaking* the second column. Do the same for the second and third data sets creating *length_6x10*, *comp_6x10_breaking*, *length_10x10* and *comp_10x10_breaking*.

3. Now use the polyfit function for each of the three data sets. The relationship will be linear for we can use 1 for the third input argument. Call polyfit with the inputs *width*, *tensile_breaking* and the polynomial order. E.g.

```
tensile_coeffs = polyfit(width, tensile_breaking, 1);
```

   Do the same for both the compression data sets.

4. Now we have three sets of coefficients for the various member tests. Each will have to two coefficients, the first will be the gain and the second the offset. We want to generate the data to plot the lines of best fit. Use the polyval function, passing the coefficients and the width/length vectors. This will return the breaking load vectors that can be used for our plots. E.g.

```
tensile_fit = polyval(tensile_coeffs, width);
```

   Again, do the same for the compression data sets.

5. Now we want to display the line equations to the user. To do this we'll use fprintf three times. You will need to use the coeffs for the second input argument. The first input argument will be a string with two placeholders (%). The result of the first fprintf function should say:

```
Line of fit for tensile members is f = m*w + c.
```

   Where m is the gain and c is the offset coefficient. The result should three lines of text in the command window.

6. Now we want to create three plots for each of the tests. For the first, include the trend line by plotting the *tensile_fit* against *width* and the original data *tensile_breaking* against *width*. The latter should be plotted using single markers by using the linespec 'o'. Do the same for the compression tests. Don't forget to add labels, legends and a title.

7. Save your script and run it. You should get the line equations in the command window and three figure windows with the original data and the trend lines on.

End of exercise 9.2

Exercise 9.3 – Basic Curve Fitting.

As a bonus if you finish exercise 9.2 we can do another polyfit exercise, this time with a higher order polynomial. Reading some data from file we will use the polyfit function to determine the correct order of polynomial to model the relationship between two sets of data.
We'll then use polyval to evaluate the polynomial and then plot the results. Download the position_data.csv from the VLE and save to your working directory. This contains the position data for a buggy along with the sample time. It has two columns of data, the first is time in seconds, the second position data. I want to see if you can find a relationship between position and time.

Follow the instructions below:

1. Create a new .m file and save it as curve_fit.m. Clear the workspace and command window. Load the data from the CSV file and separate into two variables, time and position.

2. Plot the position data against time using circles to mark the data points by using the linespec 'o' argument. Label the plot and hold the plot using `hold on,` we'll return to the plot later.

3. We now wish to find the line of best fit for the data. We will use the polyfit function a number of times to produce the coefficients. We'll try 2nd, 3rd, 4th and 5th order polynomials generating a set of coefficients for each. Call polyfit passing time, position and the polynomial order. E.g.

```
coeffs_2 = polyfit(time, position, 2);
```

Do the same for the further polynomial approximations.

4. Now we have four sets of coefficients for the various polynomial approximations. We want to generate the data to plot the curves. Use the polyval function, passing the coefficients and the time vector. This will return the position vectors that can be used for our plots. E.g.

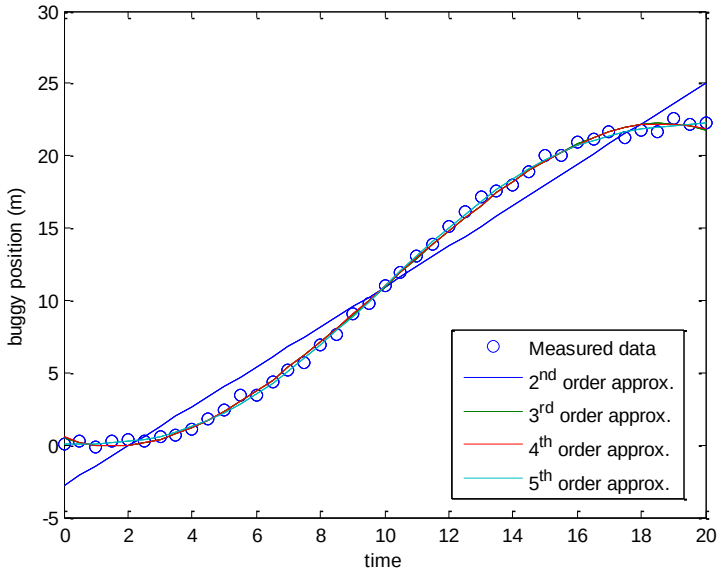
```
position_2 = polyval(coeffs_2, time);
```

5. Now plot the four curves on the same plot as our original data. Add a legend too, the plot should look like the figure below.

6. Finally we want to quantify how close the fit is to the original data. We will use the sum squared error, that is we find the error between the modelled value and the measured value, then square it using the equation:

$$SSE = \sum_{i=0}^{n} (x_m - x_p)^2$$

where x_m is the measured data and x_p the predicted data. Calculate and display the SSE for each polynomial. Display the SSE for each to determine the best fit.



Save the program and run it to see the above plot and show the SSE for each polynomial fit.

End of exercise 9.3