

Download the ball\_demo.vi and bouncing\_ball.m from the Lab 8 section of the VLE and place into your working directory.

## Exercise 8.1 – MathScript Node in LabVIEW

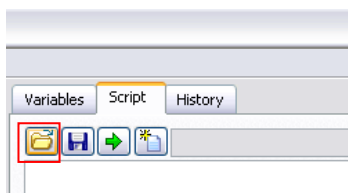
In this exercise we will look at implementing some MATLAB script, already written, in a MathScript node in LabVIEW to demonstrate LabVIEW's ability to run m-script. It will highlight the benefits of using LabVIEW with MathScript to offer the ease of creating a GUI in LabVIEW with the textual based code of MATLAB.

1. Launch LabVIEW from Departmental Software → Engineering → National Instruments → LabVIEW and open the ball\_demo.vi. This vi contains LabVIEW code that models a ball falling vertically due to gravity and a simplified impact using the coefficient of restitution to describe the ball's change in velocity between pre and post impact. The coefficient of restitution can be derived from:

$$C_R = \frac{v}{u}$$

where  $u$  is the initial absolute velocity of the first object,  $v$  is final absolute velocity of the object when the object hits a stationary surface.

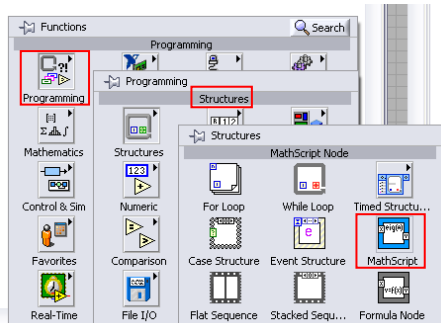
2. Have a look at the block diagram to see how this model is implemented in LabVIEW. Run the vi and play about with the dials to see the effect of varying the initial height, the coefficient of restitution and the initial velocity.
3. Now we want to look at the m-file we are going to run in LabVIEW. Open the MathScript Interactive Window in LabVIEW by going to tools → MathScript Window... in the menu. This opens the LabVIEW MathScript window, we will use this to check that the m-file will work. Click on the Script tab and then select the open icon. Locate bouncing\_ball.m in your local directory and open it.



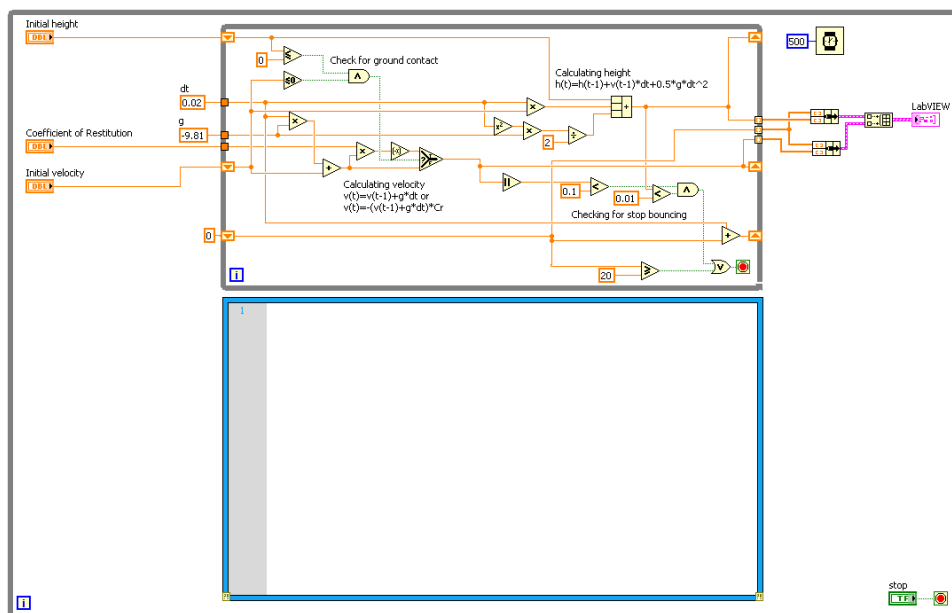
The script will appear in the window. Now run the script by clicking the green arrow (note it may take a while to run the first time while LabVIEW compiles the code). When it is done, you should see a figure showing the ball bouncing with  $C_R$  set to 1. To allow the user to change the values, they have to be typed

into the script, an input function added or a GUI would need to be developed in MATLAB. Instead we can place this pre-written script into a MathScript Node in LabVIEW and use a LabVIEW GUI to drive it and plot the outcomes.

- Return to the ball\_demo.vi and open the block diagram. Here you'll see the code implemented in the graphical programming language of LabVIEW, to use the script we've just tested we want to add a MathScript Node in the block diagram. Expand the while loop down to make space for the MathScript Node. The MathScript can be found in the Structures palette of the function browser as shown below:



You can then place the MathScript node on the block diagram. It should look like this:

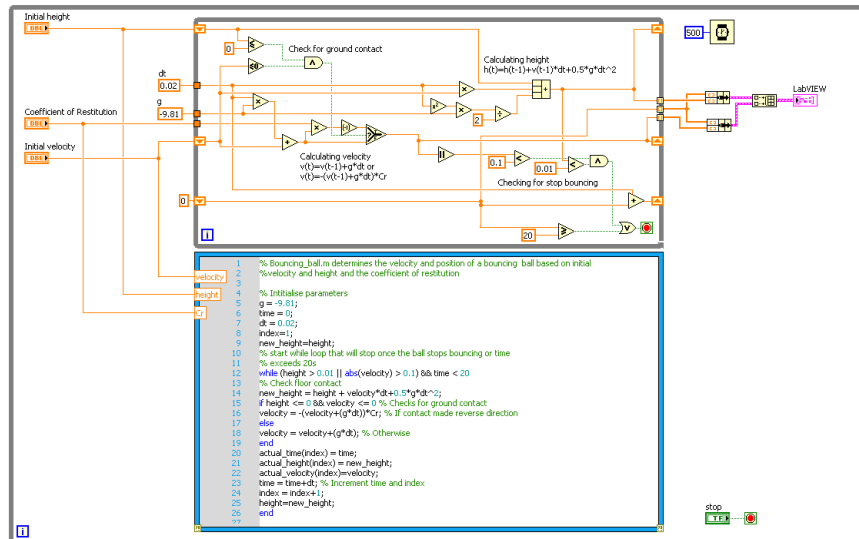


- We are going to use the script we have just tested in the MathScript Interactive Window. We can either cut and paste the code or we can import the .m-file by right clicking in the MathScript node and selecting Import... Locate bouncing\_ball.m and select okay. You should see the script appear in the box.
- Remove the clear and clc functions from the script, these will cause errors in the MathScript node.
- Now we want to drive the script from the front panel and display the output there too. To do this we need to define the inputs and outputs used by the MathScript node. The inputs we are going to add are height, velocity and Cr. We should remove their definition from the script by the following 3 lines from the script:  
`height=5;  
velocity=0;  
Cr=1;`  
 Notice red crosses may appear at the start of certain lines of the code, this is because a variable is no longer been defined. This will be rectified by the next step. This will be rectified shortly.

8. We also want to plot the data on the front panel. (Note: If you can't see the lines of code used for plotting, right clicking on the MathScript node and select Scrollbar from the Visible items menu, you can then scroll down). Remove the last 4 lines of code used for plotting:

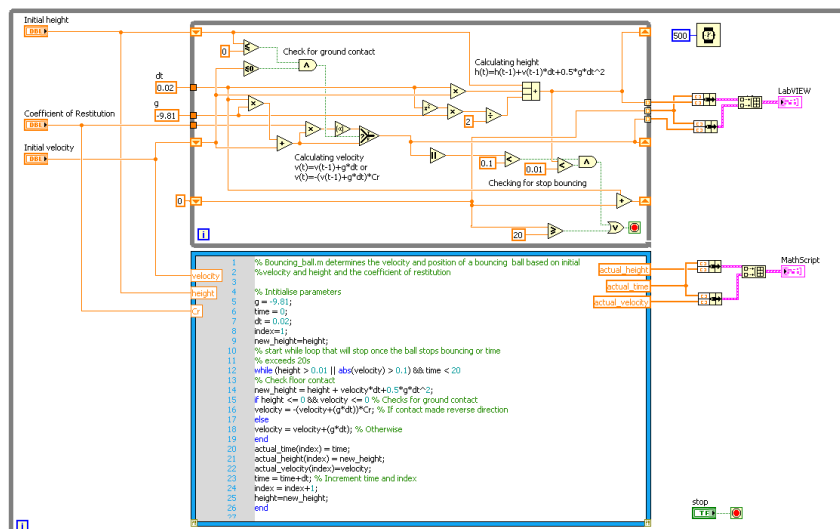
```
plot(actual_time, actual_height, actual_time, actual_velocity);
xlabel('time (sec)');
ylabel('amplitude');
legend('height (m)', 'velocity (m/s)');
```

9. Now we need to add the inputs and outputs to the MathScript Node. On the left hand edge of the MathScript node, right-click and select Add input. A small box will appear, type *height* into it. Do the same for *velocity* and *Cr* (remember script is case sensitive). These can now be wired to the Initial height, Coefficient of restitution and Initial velocity controls respectively as below:

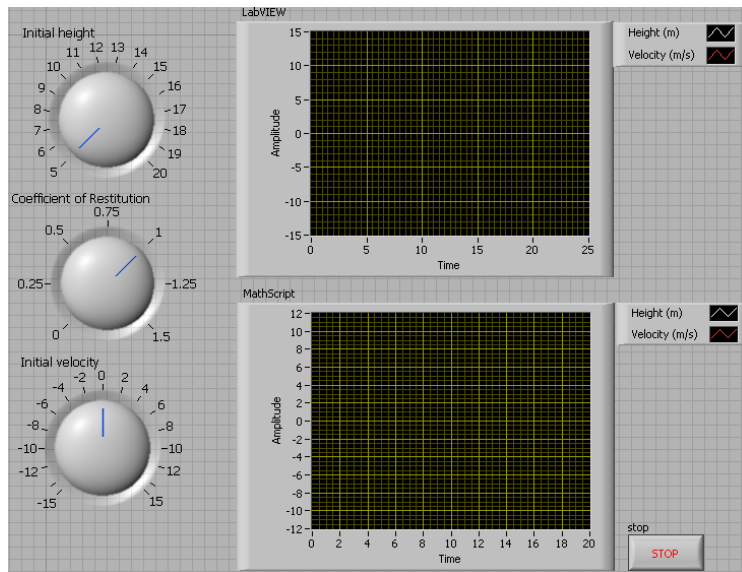


10. Now add the Outputs in the same manner. Unlike inputs where the data type is defined by the type of wire connected, the data type of the output needs to be defined. Firstly run the VI, this runs the script and will allow the output data types to be defined. Stop the vi and now add the outputs by right-clicking on the right-hand edge of the MathScript node and selecting Add Output, do this for the actual\_time, actual\_height and actual\_velocity outputs.

11. We want to display the outputs on another XY graph. Copy the two bundles, the build array and LabVIEW indicator placed at the right hand side of the LabVIEW while loop and paste it to the right of the MathScript node. Now wire the outputs of the node to the bundle blocks, the first of each should be actual\_time while the actual\_height and actual\_velocity for the second inputs respectively. Rename the second indicator to MathScript. Your diagram should now look like this:



12. Rearrange the front panel to look like the below:



Save and run the vi. You can change the dials and see that the outcome from the LabVIEW code and the MathScript. Look back at the block diagram and compare both sets of code. It is personal choice which is easier to read and which is quicker to write. They both have benefits and using the MathScript function allows you to take advantage of both.

13. If you finish go back and have a further look at the MathScript Interactive Window.

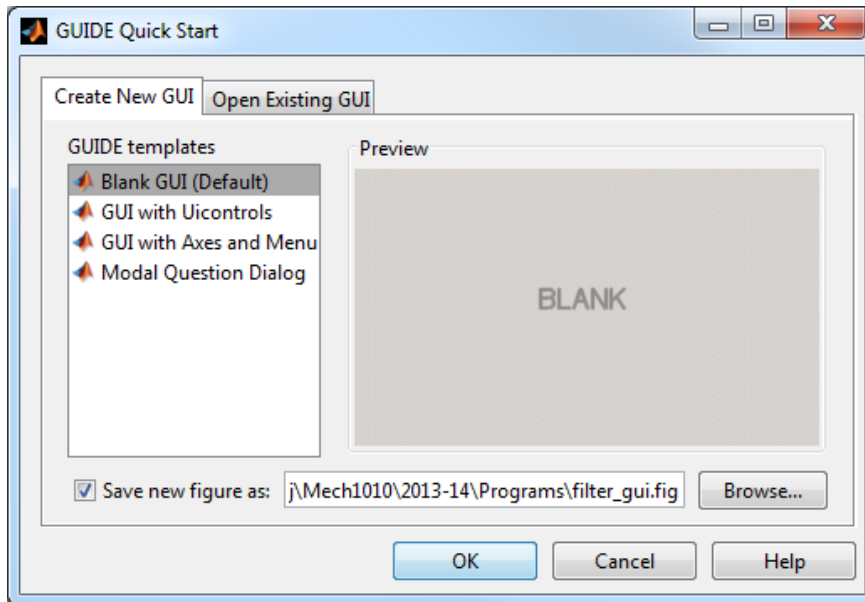
NOTE: There is an error in the way LabVIEW runs this code, check the difference between the two outputs, see if you can find the flaw.

**End of exercise 8.1**





## Exercise 8.2 – Creating a Graphical User Interface

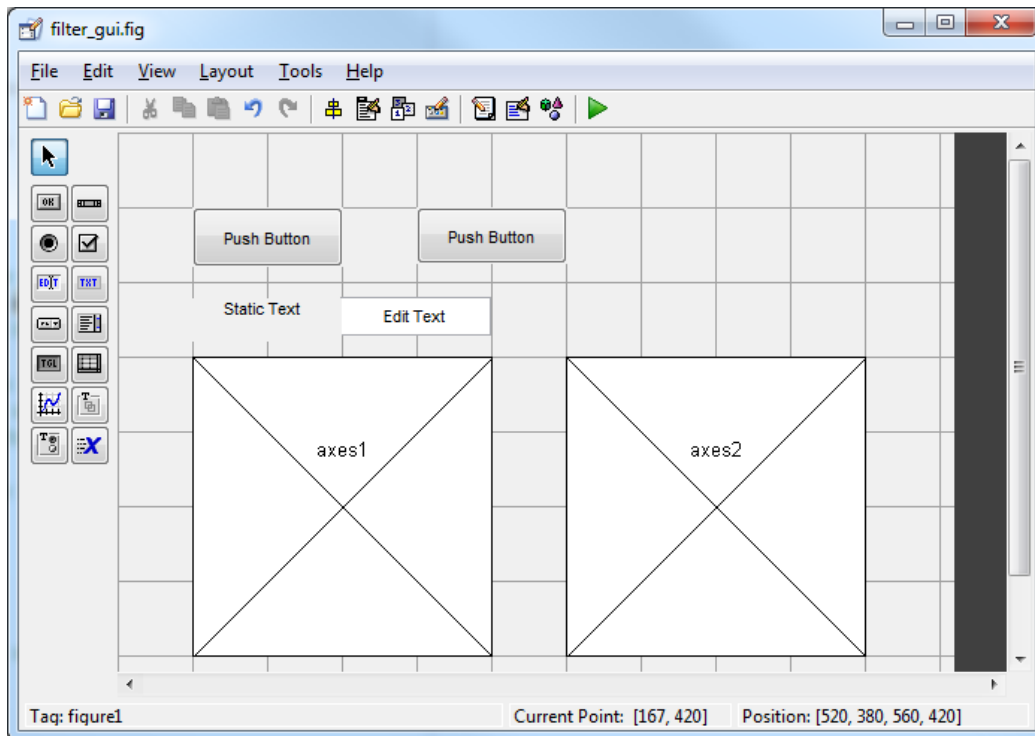
In this exercise we will look creating a Graphical user interface for the data filter program we created in a previous's lab. From it we should see that while more complex with MATLAB than with LabVIEW we can still create User Interfaces with relative ease using the GUIDE toolbox.

1. Open MATLAB and make sure you are set to the correct working directory. Type `guide` in the Command Window to open GUIDE. The GUIDE Quick Start window will open. On the **Create New GUI** select **Blank GUI (Default)** from the GUIDE templates then check the *Save new figure as:* box and enter the name `filter_gui.fig` saving to your working directory and click OK. This will open the GUIDE editor window and also the associated `.m` file containing the code used by the GUI.



2. We are now going to build a simple GUI which will feature two push buttons, a text box and two axes. All of these UI objects can be place on the figure workspace by clicking on the relevant button on the toolbar.

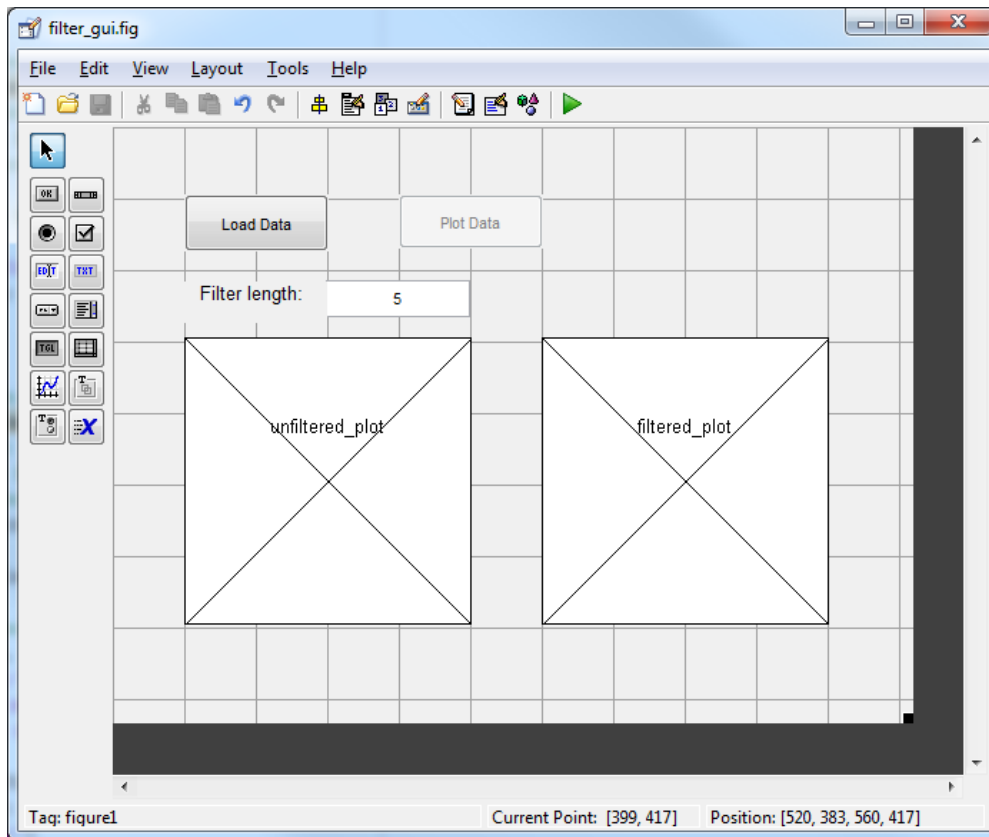
Click the icon for a push button:  then left-click in the blank figure and drag the mouse to place the push button, you'll notice the default text is Push Button, we will change this shortly. Add another push button and then an edit text box: , a static text box:  and two axes: . Arrange the objects on the figure to look like this:



3. We are now going to alter some of the properties of these UI objects. Firstly we will look at the first push button. If we click on it we should see in the bottom left of the window the text Tag: pushbutton1. Now double click on the push button. This will open the properties inspector for pushbutton1. What we see here is a list of all the properties in the left column and the values these properties are set to in the right column. Have a look at all the properties you can alter for a push button. We are now going to change two properties, firstly look for the String property. This is the text that appears on the push button. In the value column, change this to: Load Data. Now we are going to change the Tag, this is how you will access the properties of the pushbutton using `handles.tag`. Change this to `load_data`. Note that tag must follow the same rules as variable names, so no spaces or punctuation other than an underscore. Once you've typed the new tag, remember to press enter otherwise the tag may not be updated, you can check by selecting an object and looking at the tag shown in the bottom left corner of the GUIDE window.
4. Now repeat this for the other 5 objects, The table below shows the properties for each object that should be changed:

UI Object	Property Name	Property Value
pushbutton1	String	Load Data
	Tag	load_data
pushbutton2	String	Plot Data
	Tag	plot_data
text1	String	Filter length:
	FontSize	10
edit1	String	5
	Tag	filter_length
axes1	Tag	unfiltered_plot
axes2	Tag	filtered_plot

5. Your GUIDE window should now look something like this:



We are now ready to start adding functionality to our code. Save the filter\_gui.fig in the GUIDE window. You should see the editor window update to take into account the new UI objects we've added to the GUI. Close the GUIDE window.

- Have a look at filter\_gui.m in the Editor. You'll see a number of functions in the file. The top one filter\_gui sets up the figure, we should not make any changes to this function. Scroll down the page and we should see the second function filter\_gui\_OpeningFcn. This function executes just before the GUI is made visible. You can put code in here for initialising values in text boxes or disabling push buttons. For this exercise we are going to set the *Enable* property of the Plot Data button to *off*. To do this we will use the set function. This is of the form:

```
set(handles.tag, 'PropertyName', 'PropertyValue')
```

In this case our *tag* is plot\_data, our property name is 'Enable' and our 'PropertyValue' is 'off'. Note these are case sensitive and the apostrophes should be included. Place this at the end of the filter\_gui\_OpeningFcn.

The next function filter\_gui\_OutputFcn is for returning outputs to the command line, we'll not be editing this function. Below these three functions are the functions we are interested in. We should have 3 callback functions load\_data\_Callback, plot\_data\_Callback and filter\_length\_Callback. We also have the filter\_length\_CreateFcn which sets some properties of the edit text box. We are interested in the first two Callbacks.

- The first one we are going to look at is the load\_data\_Callback. The code contained within this function will execute when the user presses the Load Data button. At present this is empty aside from some comments. Now open your enforcer.m file from week 6 in the Editor. If you didn't complete the exercise, download the solution from the Lab 6 section on the VLE.

In enforcer.m select the lines of code that load in the data and change the filename to noisy\_data.csv and the two lines of code that create the vectors unfiltered\_data and time. Cut them from data\_filter.m and paste them at the bottom of the load\_data\_Callback function in filter\_gui.m.

- We now have two further tasks for this Callback, firstly we need to be able to pass the variables unfiltered\_data and time to other Callbacks and secondly we need to enable the Plot Data button:

- a. To pass the data between Callbacks, we are going to use the `guidata` function. We could use globals but this way is perhaps neater. `guidata` can be used to pass the data using the handles data structure, as such we need to place our variables into the handles data structure. A handle is simply a number that points to a graphical object and is used to access the properties of the object. The handles structure already contains all the handles to UI objects but we wish to add our data to it too. To do this we simply add handles and then a full-stop before any variable we wish to add so we should end up with:

```
handles.unfiltered_data = data(:,2);
handles.time = data(:,1);
```

We now need to store the handle data in the user interface so it can be retrieved by other Callbacks. To do this we use the function `guidata` passing it the first input argument as a handle to the GUI figure, in this case `hObject` and then the data we wish to store, in this case it is the `handles` structure. So we should have:

```
guidata(hObject, handles)
```

at the bottom of the `load_data_Callback`.

- b. Now we have loaded the data, we wish to enable the Plot Data push button. To do this we will use the `set` function as in step 6, but this time the property value will be set to 'on'. Place this at the end of the `load_data_Callback` function.

9. So we have now completed the code that will run when the user presses the Load Data button. We are now going to look at what needs to happen when the Plot Data button is pressed, now it is enabled. When the Plot Data button is pressed we are going to call our UDF `moving_average_filter`. This will take two input arguments, our unfiltered data and the length of our filter, `n`. We need to get the value of `n` from the text box in the GUI, allowing the user to update the filter length. To access the value we need to use the `get` function, this takes the form

```
result = get(handles.tag, 'PropertyName')
```

in this case our tag is `filter_length` and our 'PropertyName' is 'String'. This will return the numeric value in the text box as a character string. We need to use the function `str2num` to change it to a numeric. So the code will look like:

```
length_of_filter = get(handles.filter_length, 'String');
n = str2num(length_of_filter);
```

in the `plot_data_Callback`. We can now call our UDF in the normal way, however we need to maintain the structure of the handles data so when we pass the UDF the `unfiltered_data`, we need to use `handles.unfiltered_data`:

```
filtered_data = moving_average_filter(handles.unfiltered_data,n);
```

10. Finally we are going to plot the unfiltered data against time and place it in our first axes, specified by the tag `unfiltered_plot` and plot our `filtered_data` against time and place in the second axes, specified by the tag `filtered_plot`. We can use our plot functions as normal except we need to pass the handle to the object as the first input argument. Here is the code that you will use for the unfiltered plot:

```
plot(handles.unfiltered_plot, handles.time, handles.unfiltered_data, 'r');
xlabel(handles.unfiltered_plot, 'time (sec)')
ylabel(handles.unfiltered_plot, 'position (m)')
title(handles.unfiltered_plot, 'unfiltered data')
```

Now do the same for the second plot with the filtered data, note that filtered data has not been added to the handles data structure so just `filtered_data` will suffice.

11. Save your `filter_gui.m` and run it. You should see the GUI window open, test the functionality of your code by loading data and then plotting data. Also update the filter length and plot the data again. If the spacing of your GUI isn't quite right, launch GUIDE and open your saved .fig file and rearrange the UI objects to make more space, then save your figure again before running you m-file.

If you finish, have a look at other properties that you can change in UI objects and also the figure window itself.

## End of exercise 8.2