MEC8043 – LabVIEW programming

Purpose of the exercise

LabVIEW forms part of the MEC8043 module. You will be learning to program in graphical form. Why are we asking you to do this? The purpose is three fold. Firstly, LabVIEW is used extensively in industry (see www.ni.com for more details) and therefore is not only a useful tool to know but also helps your course to be accredited. Secondly, we hope you will use LabVIEW in your projects, either to assist in data collection or you may specifically require it for some projects. Thirdly, this is also an exercise in good experimental practice, understanding uncertainty/noise and an introduction to signal processing. You must remember that modern engineering is all about intelligent systems and therefore to be an effective engineer, you need knowledge of electronics / signals / programming in order to engineer systems.

There will be a lecturer and several student demonstrators to help you with any questions you have while doing the work. There are no more formal lectures so as soon as you enter the computer cluster, feel free to get started.

Timescale

You have been assigned to one group and to three two-hour sessions. For the first 2 sessions, you are expected to complete tutorials 1 through to 4. Note that you can finish some of these in the Robinson Library clusters in your own time. Data acquisition hardware can only be used in F17 during the scheduled classes. On the last day you will acquire data from a sensor for your assignment. You will only have that two-hour session to complete the required programming and data collections for your assignment. It is therefore important to complete and understand the first 4 tutorials before session 3 begins. You will submit your program via Blackboard before the end of the last session, whereas you will have a few days to write up your report before the following Monday's hand-in.

Assessment

The requirements of the assignment are at the end of these tutorial sheets. Submission templates are available in Blackboard, where marking breakdown is also set out. If you fully understand what you are doing in these tutorials, you may be able to add extra functionality to your solution and also consider user requirements and aesthetics, this will gain you higher marks.

MEC8043 - Introduction to LabVIEW

Lecturer: Michele.Pozzi@ncl.ac.uk

Introduction

LabVIEW is just a programming language however rather than writing code, a series of symbols and connecting wires are used. Much of the usefulness of LabVIEW comes from the availability of many blocks of code, for loading/saving to a file, for input/output operations, for data analysis, etc.

We shall examine LabVIEW by working through several exercises by the end of which you should have a basic working knowledge of it. These tutorials will cover:

1. Introduction to LabVIEW basics
2. Program structure and presentation
3. File operations
4. Specialist icons – specific example on signal processing

Opening LabVIEW

To open LabVIEW proceed as follows:

- Go to the Start menu
- Locate or search for LabVIEW 2015, either 32bit or 64bit
- Select Blank VI.

Note: It is also recommended that you create a folder on your disk space, i.e. My Documents, to save your work to.

Note: LabVIEW 2015 is also installed in the Robinson Library clusters and other PCs around campus.

You will be presented with 2 windows:

- Front Panel
- Block Diagram

Note: The Block Diagram can be obtained from the Window → Show Block Diagram menu of the Front Panel. You can press Ctrl+E to quickly switch between the two or Ctrl+T to placed them side by side. Double-clicking an element in one window will highlight the corresponding element in the other (if it exists).

Two other windows which you will find helpful through the exercises are:

- Context Help

Note: The Context Help can be opened from the Help menu bar.

- Controls / Functions (depending on which window is active)

Note: This window can be opened by right-clicking in the appropriate window (Front Panel or Block Diagram).

We will use these windows as we work through the examples. There are 5 tutorials (1 hour each) and then an assignment to complete. You should open a new Blank VI for each of these 6 tasks. It is highly recommended that you save each one with a distinctive name (such as Tutorial1.vi) and **save regularly** (also with different filenames) as LabVIEW sometimes crashes and makes you lose all your work!

Tutorial 1. A simple monitor

Learning outcomes

By the end of this exercise, you should be able to write LabVIEW programs that:

- perform basic input operations,
- perform simple calculations,
- output information in both numeric and graphical form.

Building the Front Panel

Create input controls and output indicators

From Controls (right click in the Front Panel window if this does not show) select each of the following and place them on the Front Panel:

- Express → Num Ctrls → Num Ctrl
- Express → Num Ctrls → Pointer Slide
- Express → Num Inds → Num Ind
- Express → Num Inds → Meter

Notice how data flows left to right in LabVIEW: controls have (output) terminals on the right, so you can connect them to the input terminals that indicators have on their left. For better readability, you should strive to keep this direction flow.

Firstly, we want to change the range of the Meter. Click on the number 10 in the Meter and change this value to 100. The Meter on the Front Panel should update accordingly.

Double click on the text of Numeric of the input box and rename as 'Scaling Factor'. Similarly double click on Numeric 2 and rename as 'Output'.

Note: You can do this as soon as the box is created.

You have now set up the Front Panel for your simple monitor although at present it will not do much. As you have been creating this display, corresponding icons have appeared in the Block Diagram window. This is where you do the 'programming'.

Programming the Block Diagram

Multiply the two inputs and display the output graphically and numerically

Click on the Block Diagram and from the Functions window select Express → Arith & Com → Numeric → Multiply and place it between the icons in the Block Diagram. If you move the cursor over the Multiply icon its description will be given in the Context Help window – this currently shows there are two inputs x, y and an output x*y (although this is fairly obvious at present, when we get to more complex icons this Context Help will prove invaluable). Move the cursor over to the x-input of the Multiply icon in the Block Diagram window and it should change into a reel of wire. Left click at this point and move it to the small output arrow of the Scaling Factor icon. Left click again to finish the wire. If the wire is connected properly it should turn a thin orange colour.

Note: You can left click several times to get bends in the wire thereby making your diagram look better (which becomes important in complex programs). The thin orange line indicates a single double-precision value (as shown by the Context Help window). An integer would be a blue line. Array have thick line of appropriate colour. A variety of lines colours and styles are available depending on the data being 'sent' along this line.

Complete your diagram so that it looks something like Figure 1.1. Save the program as Tutorial1.

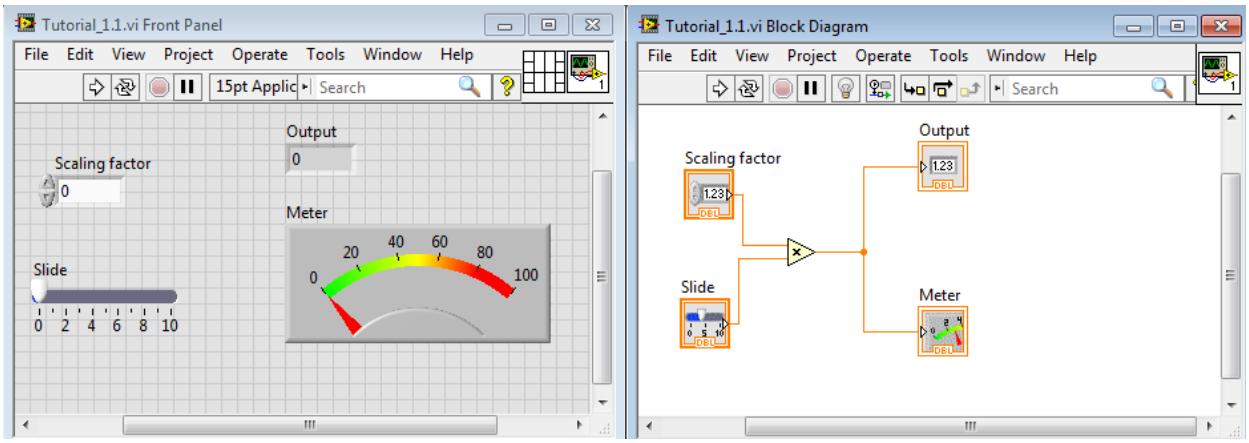

Figure 1.1: The components in the Front Panel are 'wired' via the Block Diagram

Run and debug the program

Set inputs and run the program

We need to set some input values. Place the mouse pointer over the slider until it turns into a hand (if the control is selected you need to de-select it first). Move the slider to some level, say about 5. Next either click in the Scaling Factor box to allow you to type a value or use the arrow keys at the side. Set this value to 6. To run the program, click on the arrow icon just below the Edit menu. You program gives a digital output and also a visual output in the form of the Meter.

Debugging

Let us pretend we have made an error in the programming. Click on one of the input wires to the Multiply icon and press Delete. Either the line is completely deleted or parts of the line are left (shown as dashed with a cross). This looks fairly obvious although in a larger program it may not be so obvious. Try running the program again: you should notice the 'run' arrow is now broken. When you click it, you will get an error message. Click on one of the error messages and it will highlight the part of the program that is causing the fault. Reconnect the wire (either wire between the loose ends or press Ctrl-B to delete any broken wires and then rewire the connection). The program should now run.

Faults are not always so obvious, for example it may be wired up but the wrong value is being sent. One way to monitor this is to Highlight Execution (select the light bulb in the menu bar of the Block Diagram window so that it becomes an 'illuminated yellow'). Now run the program. You will see in the Block Diagram window the values presented on each wire and the flow of the program. Again this can be very useful for debugging purposes; however it drastically slows down the program and could give problems if you are acquiring real data.

Adding extra features

Add an LED to indicate an optimal range

Say we want to be told when the output falls within an 'optimal range', such as between 40 and 60. Insert an LED into the Front Panel and label it appropriately. This will be dark green when off and light green when on (this can be changed by right clicking and selecting Properties). The actual programming will require two Comparison (in Programming→Comparison) and one Boolean operation (in Programming→Boolean). A quick way to create the constants 40 and 60 is to right click the input port of the comparison icons where you want to connect them and select Create→Constant. This short-cut works very well in many other situations, as long as LabVIEW has a way to tell what type of constant it should create. Wire up the appropriate icons as shown in Figure 1.2.

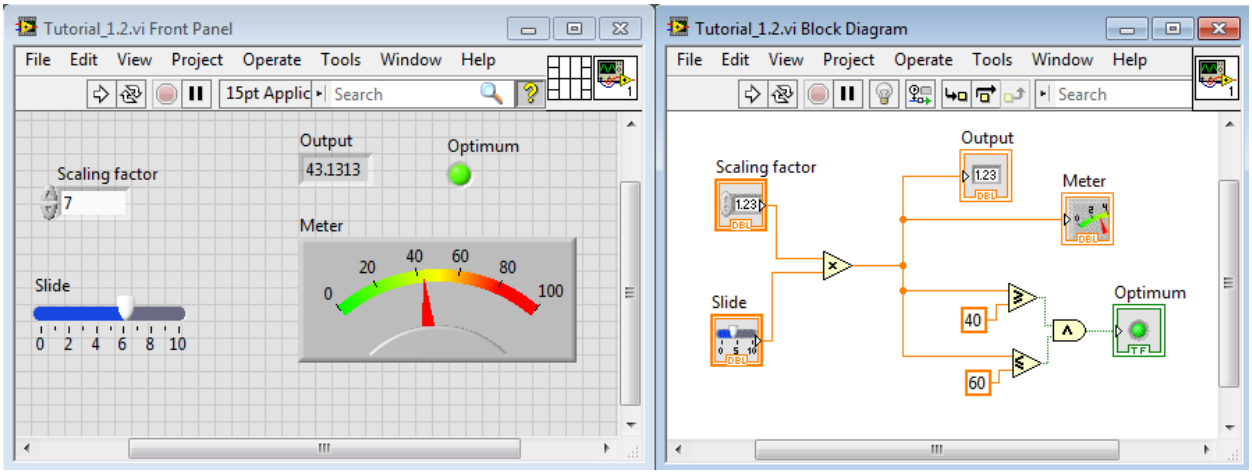

Figure 1.2: Comparison operators can be used to control an LED

Add a while loop for continuous monitoring

We are going to finish by putting the whole program in a continuous loop. From the Functions window, select Express→ Exec Control→ While Loop and create the loop so it encloses the whole program, see Figure 1.3. Note that a Stop button and icon have been automatically placed on the panel and in the loop, respectively. Run the program (without Highlight Execution selected) and you will see how fast the program works. Try adjusting the slider as the program runs. To stop the program either press your newly created STOP button or select the stop icon (red circle) in the menu bar. Pressing the button is better as the program exits "cleanly"; use the icon in the menu bar only as an emergency stop, as it may cause data loss.

The blue icon with **i** within is the loop count, so you can keep track of how many iterations have been done.

Summary

You have now covered the basics of LabVIEW programming. Although this application is itself simple, it should give you an indication of the simplicity behind LabVIEW. In the next tutorial, we shall examine further program control and presentation. Try the following exercise to reinforce the ideas learnt till now.

Exercise (if you can't finish it within the first hour, move to Tutorial 2 and complete this later in the Robinson Library)

What if the output goes beyond 100? The Meter will not show this. Try these three options:

- Right-click on the Scaling Factor box and in Properties, set the value to be limited from 0 to 10.
- Reset the Scaling Factor box back to its original settings and instead put a LED on the screen to indicate this overflow.
- Rewire the While loop so that it will stop on this overflow. You now have two conditions (button press and overflow). You want the loop to stop on either, i.e. one OR the other. You will need to perform an additional Boolean operation …

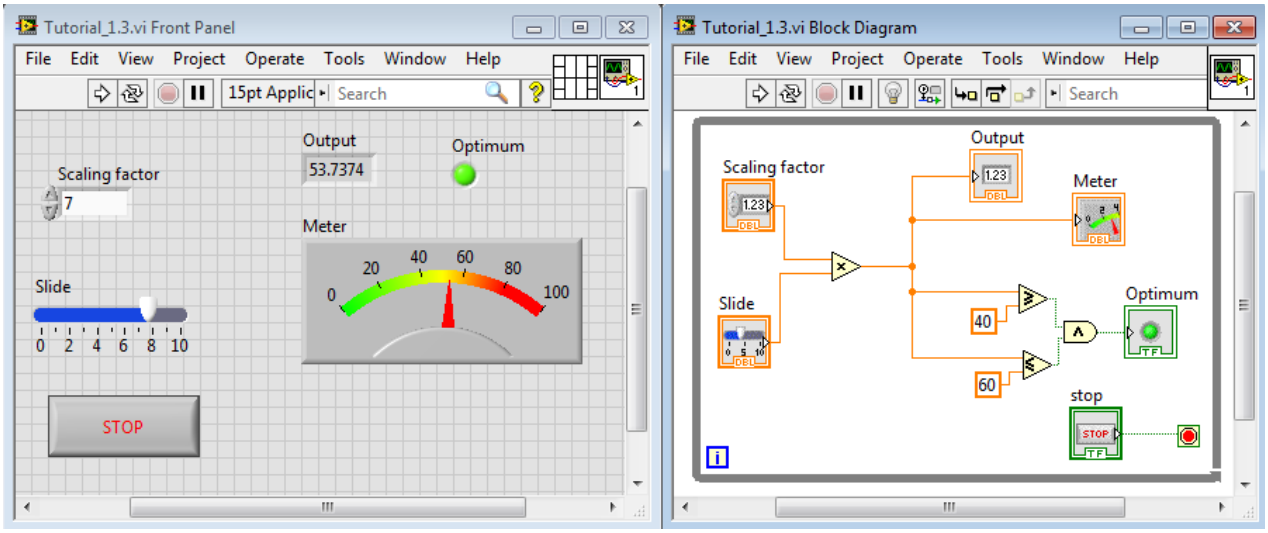

Figure 1.3: Incorporation of a While loop will allow the program to run continuously until activation of the STOP button

Tutorial 2: A simple chart recorder

Learning outcomes

By the end of this exercise, you should be able to write LabVIEW programs that:

- use trigonometric functions;
- use For loops;
- display data graphically;
- group data into arrays and bundles;
- perform a Fast Fourier Transform in LabVIEW;
- use shift registers and tunnels as appropriate.

Setting up the Front Panel

Place a chart and two numerical inputs on the Front Panel

Right-click the Front Panel, select Express→Graph Indica...→Chart. For the inputs: Express→ Num Ctrls → Num Ctrl. Label one input as Frequency (set value at 0.01), the other as Amplitude (set value at 5). Make sure no control is selected (by clicking on an empty spot), then in the main menu bar, select Edit → Make Current Values Default (so these values appear whenever the program is reloaded) and then save the program as Tutorial2.

Programming

Set up a For Loop to calculate a sine wave with the desired frequency/amplitude and plot it.

In the Block Diagram Window, we'll enclose all the icons within a For Loop: right-click the Block Diagram and select Programming (you may need to expand the Functions dialogue to see this) → Structures → For Loop. Draw the block around all your icons. Wire a Numeric Constant of 500 from the outside of the loop to the N in the corner. *A quick way is to right-click the blue N and select Create Constant.* The blue **i** is the loop index and will be incremented from 0 to 499 by 1 at each loop iteration. The loop will therefore repeat 500 times. We'll think of **i** as time, so it will increase by 1 s at each iteration, stopping and 500 s (but it's not *real* time, the code will be executed much faster!).

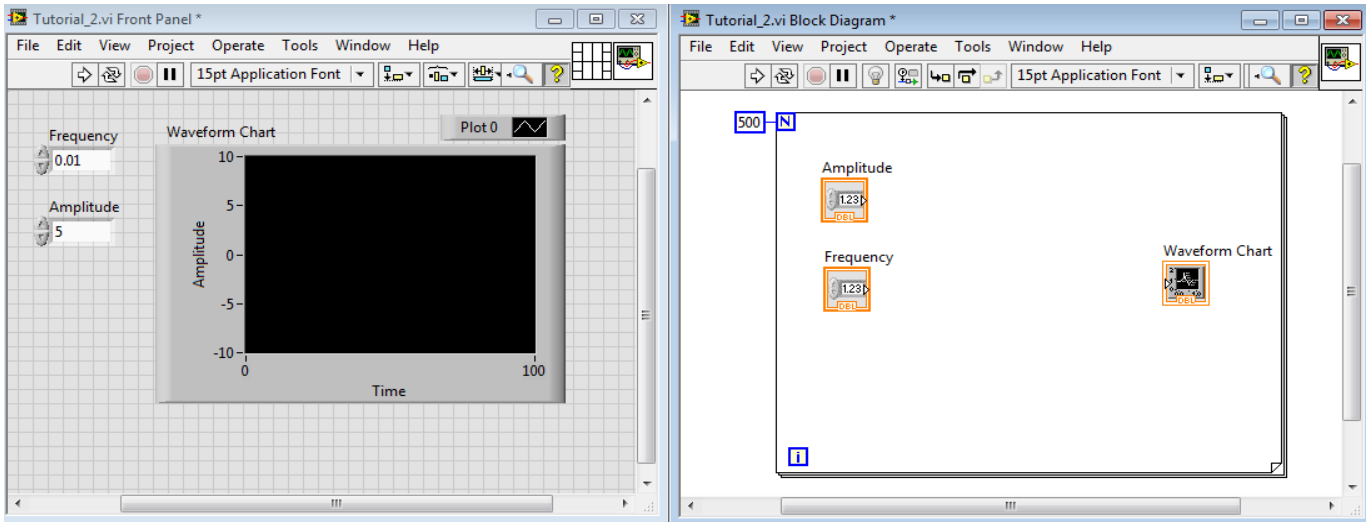

Figure 2.1: Using a For Loop for program control

We want to implement the expression: $y(i)=A\sin(2\pi f i)$, where *f* is in Hz and *i* in s. Right-click the Block Diagram, then Mathematics→Numeric→Math Constants→**2π**. For the product, Mathematics→Numeric→Compound Arithmetic (use Context Help to see how to change the + to a ×). You may have noticed a little red dot when you connected the i: the blue line indicates it's an integer; to multiply it by a double precision floating point (orange) a *type cast* (from int to double) is needed; the red dot warns you that this has been done automatically.

Feed the output from this multiplication into a Sine icon (Mathematics→Elementary & SpecFn→Trigonometric Fn → Sine) and then multiply the output by the Amplitude input. Finally wire the output into the Waveform Chart as shown in figure 2.2.

Run the program. You will only see the last period as the Waveform Chart by default only shows 100 points at a time (the last ones). Click on the 399 on the left of the x-axis and change it to 0, you will see 5 periods. If you run

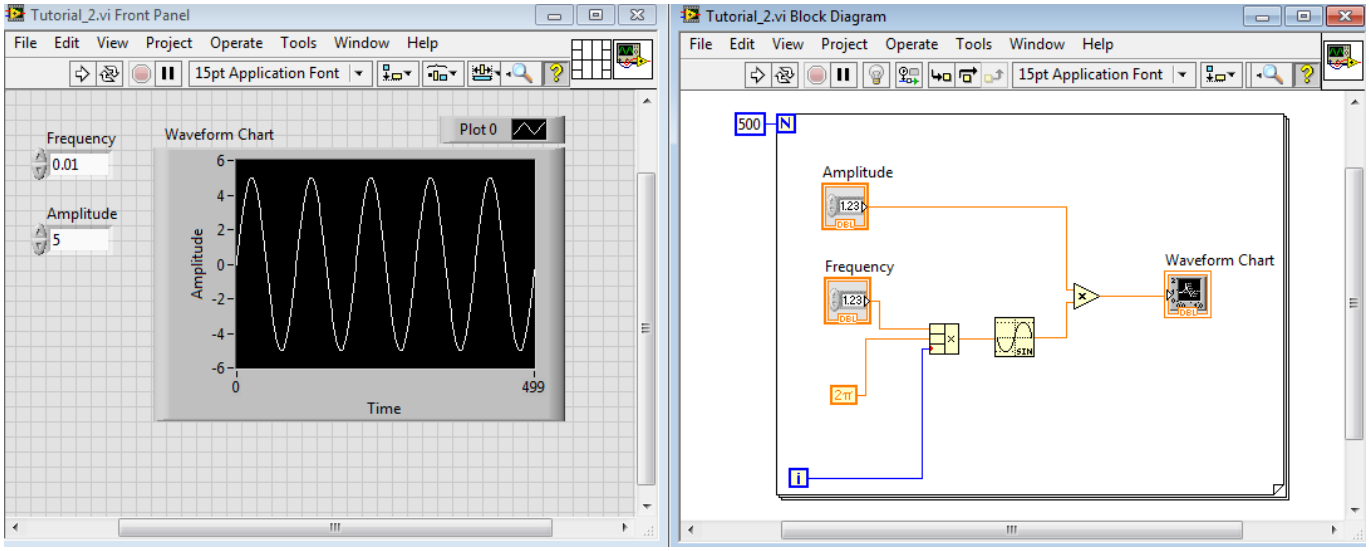

Figure 2.2: Using trigonometric functions

the program again, you will see the following 500 points, and so on. To clear the Chart, right-click on it, then Data Operations→Clear Chart (but you don't need to do this every time you run the program).

Frequency components

A very important characteristic of signals you may acquire from sensors is their frequency components, as calculated with the Fourier Transform (in digital signal analysis you'll often find the acronym FFT, where the first F stands for Fast and describes the algorithm used to calculate it). Now we will do a spectral analysis of the waveform we calculated (i.e. we'll see its spectral content, the amount of power associated with each frequency).

Calculate the spectrum of the waveform and plot it on a Waveform Graph

Right-click the Front Panel and select Express→Graph Indica...→**Graph** (*not* Chart!). Right-click in the Block Diagram and select Signal Processing→Spectral Analysis→**Auto Power Spectrum**. The signal is generated by the For loop; since we want the spectrum of the complete signal, both Graph and Spectrum icons should be outside the loop. The Spectrum VI will give us the correct results only if we supply the right inputs. The *signal* is the *y(i)* we calculated before, so pick it off the orange line that goes to the Waveform Chart. Note that on crossing the border of the loop, the line becomes thicker, indicating that whereas before it was a single value, now it's an array (or vector) of values (from all iterations). The little yellow square is a *tunnel*; right-click on it and notice how you could change its behaviour (Tunnel Mode); here we want Indexing, which means that the values calculated are put one after the other to form an array.

The second input to the Spectrum VI is dt; this is the *sampling time*, the time interval that separates the points in the Signal. From the expression we used to calculate the sinewave, you can see that dt=1 [second].

To get the correct scaling on the waveform graph, we'll bundle together the data with some "timing" information to make a Waveform Data Type (**WDT**). Select Programming→Cluster,Class&Variant→Bundle. Stretch it to have 3 inputs. On the first, goes a 0 as the initial frequency, then goes df from the spectrum VI (as the interval separating the values along x-axis), finally the data themselves (the Power Spectrum). When you connect this bundled data to the Wfm Graph, it will change colour to purple to acknowledge that a WDT is connected. Note that it is more common to use WDT for time-dependent data, but they are useful any time you have equally spaced points: you only give the start, the increment, and the actual data.

Your program should look like Figure 2.3. Save it now, as you'll need it later.

A useful item not shown by default is the **Graph Palette**, which you can activate by right-clicking the Graph and selecting Visible Items→Graph Palette. It gives you quick access to zooming and panning tools.

After zooming in, you can notice that all the "power" is present at a frequency of 0.01 Hz, as a sine wave has only one frequency! What value did you expect for that "power"? Let's check that we understand the Power spectrum...

If we have a sinusoidal voltage across a resistor, the instantaneous electrical power dissipated (Joule effect) is: $P(t) = \frac{V^2(t)}{R} = \frac{A^2 \sin^2(2\pi f t)}{R}$

which means that the average power dissipated during one cycle (period T) is:

$$P_{av} = \frac{1}{T}\int_0^T P(t)dt = \frac{1}{T}\int_0^T \frac{A^2 \sin^2(2\pi f t)}{R} dt = \frac{A^2}{2R}$$

which can be written as:

$$P_{av} = \frac{V_{rms}^2}{R} \qquad \text{if we define:} \quad V_{rms} \stackrel{\text{def}}{=} \frac{A}{\sqrt{2}}$$

We can assume that our input to the Auto Power Spectrum VI was in volt. As we haven't given R, what the VI is returning is not really the power (in watt), rather the V_{rms}^2, which is closely related to P_{av}, as shown above (if you prefer, it's giving us the power that would be dissipated by a 1 Ω resistor). We should also notice that this is a *discrete* Fourier Transform, hence the calculated power falls into *bins*. All bins will have the same width, e.g. 0.1 Hz, and the calculated power comes from all frequencies in that bin, e.g. from 10.0 to 10.1 Hz. To stay with the example, if you want to know the power associated to the range of frequencies between 10 and 11 Hz, you need to add together the power in each of the 10 bins between 10 and 11 Hz.

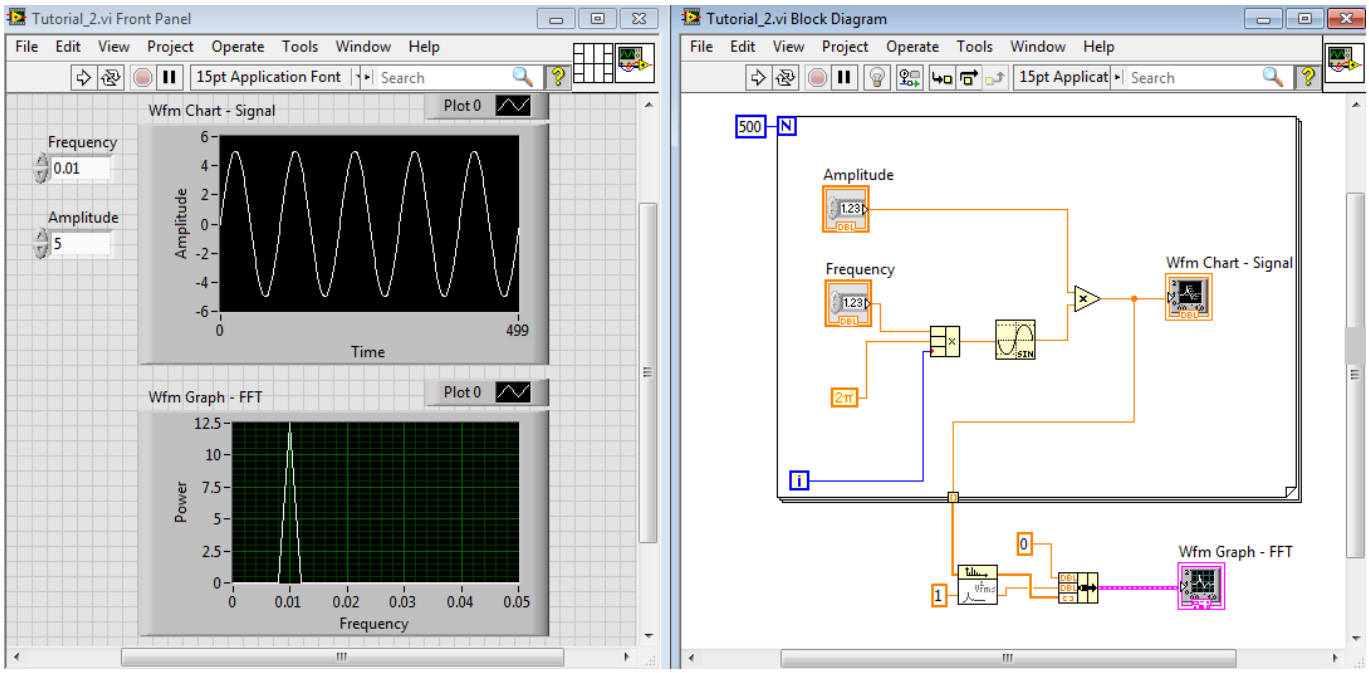

Figure 2.3: Calculation of the power spectrum

A more realistic signal

We can make the spectrum more interesting and realistic by simulating attenuation (e.g. energy loss during a vibration, which will progressively reduce the amplitude of the sine wave) and by adding some noise. Reproduce the block diagram on the left of Figure 2.4. We have here used the **Expression Node**, found inside the Numeric Palette, which is a quicker way to introduce complex maths dependent on only one variable (rather than building it with individual operation blocks). Also, the Random Number (0-1) was introduced to simulate noise. Start experimenting with Amplitude=50 and Frequency=0.01 Hz. Note that we've changed the number of iterations to 5000 to get a smoother FFT (the longer you acquire data, the higher is the frequency resolution, i.e. the smaller are the bins in the spectrum). Also note that you may only see the last 500 points of the 5000: right-click on the graph, select Chart History Length ... and increase it to 5000; then adjust the scale on the graph. Explore the spectrum and experiment by changing all parameters in the program.

Shift Registers

In programming, it often happens that you have an iteration loop (like the For loop used above) and need to access the values that were available in the previous iteration. One way to do this is by use of Shift Registers. To make one (Figure 2.4.Right), draw the wire to the side of the loop, as you did before to make a tunnel. Once you have the

tunnel, right-click on it and select "Replace with Shift Register"; it is good practice to make these to the right, and LabVIEW will move them there. You will notice that another arrow appears on the left side. So: you send the output of one iteration to the right and pick it up from the left at the next iteration – at the first iteration, the register can be initialised from outside the loop, as done in the figure by the value of 0 getting into it.

Reproduce the block diagram on the right of Figure 2.4 and see if you can figure out what mathematical operation it performs. Note that we have plotted two curves on the same waveform chart by bundling together the data we want to plot (again found in the Cluster, Class & Variant Palette). We have also added an explicit type conversion (the icon with DBL within); these are found among the Numeric Palette → Conversion → To Double Precision Float and are used when you want to ensure that LabVIEW works with the right type of variable (**i** is originally an integer; try to remove the DBL conversion and see what happens!).

Exercises *(if you can't finish these today, complete them later in the Robinson Library)*

1. Modify the program (the version before you added the exp function) to permit the meaningful calculation and plotting of a sine wave of frequency 20 Hz.
   Hints: think of *i* as time (and there's no need to plot 500s of signal)
   500 samples (i.e. calculation points) in 5 periods (which last 250 ms at 20 Hz) is a good choice
2. Replace the Wfm Chart used to plot the signal with a Wfm Graph; build a suitable WDT to send to it, so that the x-axis scaling is correct (even after exercise 1).
3. The noise introduced above was one-sided, i.e. always positive. Modify the program to make it more realistic by introducing a noise that can be either positive or negative.

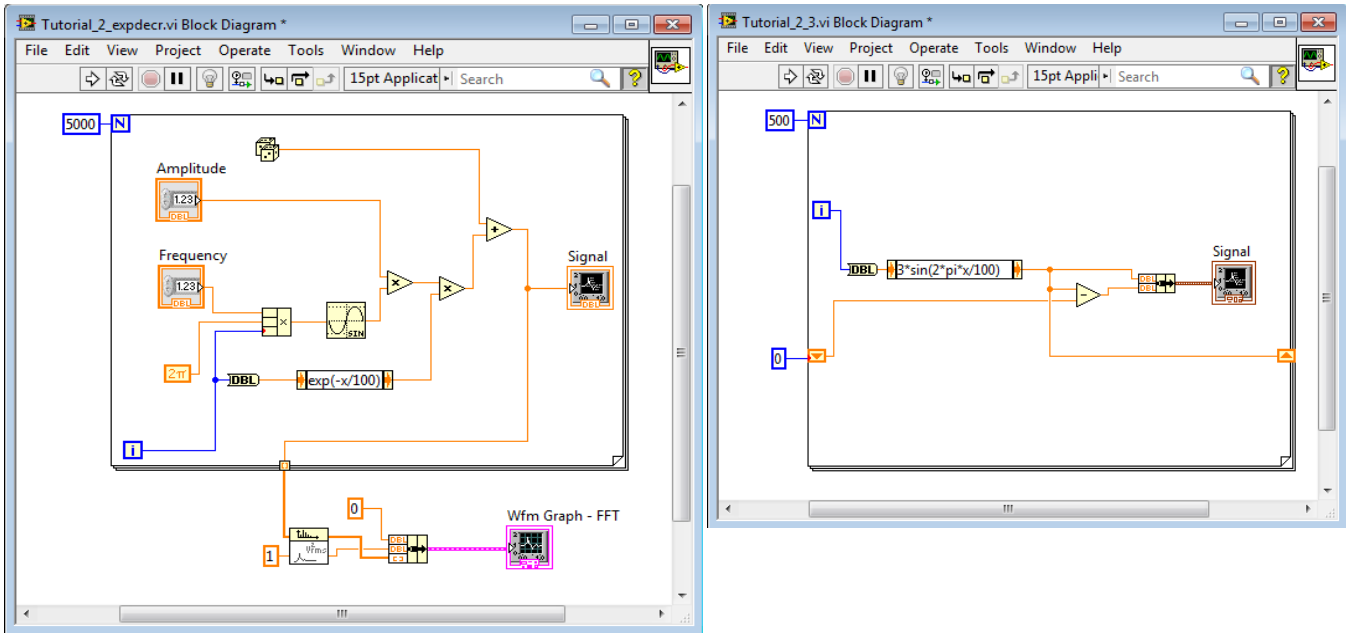

Figure 2.4: LEFT: a more realistic generated signal; RIGHT: a sine wave mysteriously operated upon via Shift Registers.

Tutorial 3: Data acquisition (DAQ)

Learning outcomes

By the end of this exercise, you should be able to:

- write LabVIEW programs that output control signals;
- write LabVIEW programs that acquire signals.

Introduction

The purpose of this exercise is to write a simple program that generates a voltage; then, we'll read it back in. Once you understand how to do this, you should then be able to use a PC with a NI USB-6008 (or higher specification unit) for a multitude of acquisition and control projects.

Connect the hardware

Connect the USB-6008 unit[1] to the PC using the USB cable on your desk.

Start a new LabVIEW program

On connection, you should be presented with a series of options. Select 'Begin an Application with This Device' and once LabVIEW opens, click on the Blank VI under the New section. If you don't get this dialogue, just create a Blank VI as usual.

Configure the DAQ Assistant to generate a fixed voltage on channel 0

Right click in the Block Diagram window and place a DAQ Assistant (Express → Input → DAQ Assist). A wizard will open to help you select the options you want for this DAQ. Click on Generate Signals → Analog Output → Voltage. For this DAQ device, we have got 2 analogue outputs channels. Select **ao0** in this case and then click on Finish. You will then be presented with various options. You should have an output range of 0V to 5V and a Generation Mode of 1 Sample (On Demand). This means that the generated voltage will change only when we execute this VI with a different value. Click OK to finish.

Create a program to generate a sinusoidal voltage

Place a For Loop around the DAQ Assistant and set the count to 200, add controls for Amplitude and Frequency just like you did in tutorial 2. Also add a DC bias, as the DAQ can only create positive voltages between 0 and +5V. Set the controls like this: Amplitude = 2, Frequency = 0.01; DC bias = 2.5. Connect the output of the calculated sine wave to the data input in the DAQ Assistant. Your program should look like Figure 3.1.

*Note: we have only generated the signal and the graph just shows what we **sent** to the DAQ device, another instrument could be used to check the output at pin 14 (**AO0**).*

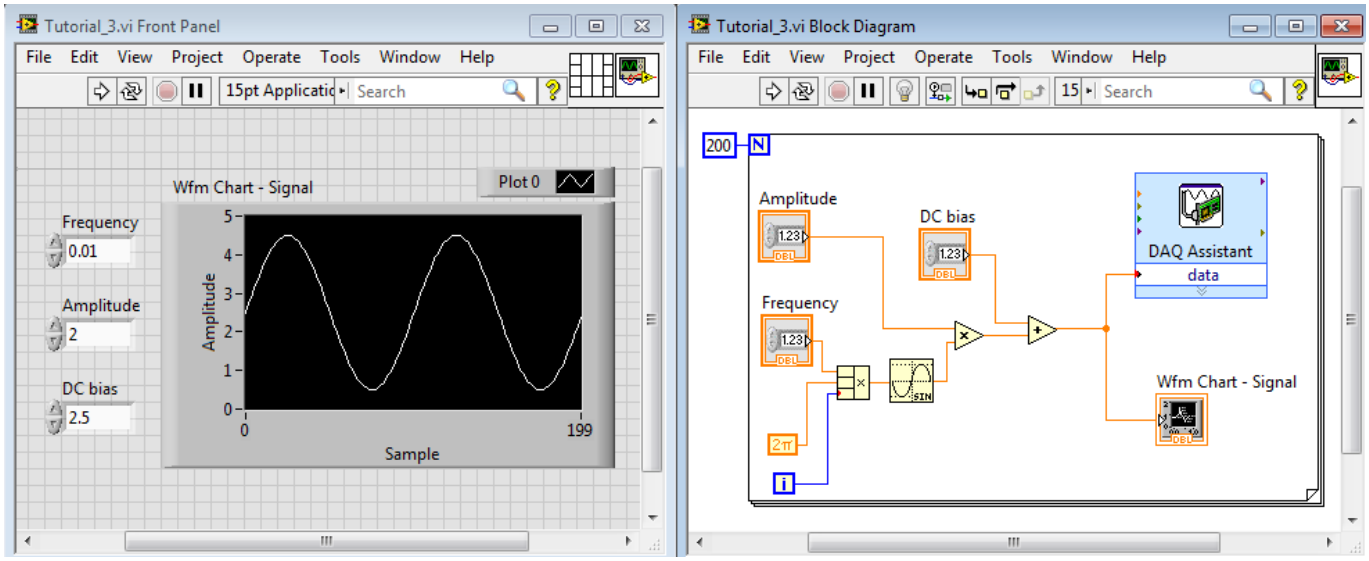

Figure 3.1: Generating a sinusoidal voltage.

1 It is always a good idea to read the manuals for the instruments you use: http://www.ni.com/pdf/manuals/371303n.pdf

Read back the voltage

We are going to use the DAQ itself to read back in the voltage we generated. In the following, ***do not over tighten the screws in the terminal,*** *hold the screwdriver simply with your fingertips*. Connect a wire from pin 14 (labelled AO0: Analogue Output 0) to pin 2 (labelled AI0: Analogue Input 0). Connect another wire between pin16 (GND) and pin 3 (AI4). This will allow us to do a differential measurement, where we look at the voltage difference between AI0 and AI4. *Alternatively, specially if you need all the 8 AI channels, you could use a single ended (RSE) measurement. In this case, you would only need to connect your signal to AI0, and the ground of your sensor to the ground of the DAQ.*

Add a sequence and a DAQ acquisition task to read in the voltage

If we just paste in more blocks, we will not know which blocks are accessed first. This is known as a ***race condition***. In general, LabVIEW will execute a block as soon as all the data it needs are available. You can use the Highlight Execution ( ) feature to see how data travels down wires; blocks that are waiting to be executed are greyed out. In our program, we want the DAQ to generate the output and *then* acquire the input. So we need to force this sequence of events.

Select Programming → Structures → Flat Sequence and place the sequence around the sine wave generation. If you right click on the border of this sequence, you can then select *Add Frame After*. Create another DAQ Assistant, this time in the second sequence frame, and configure it for input with Acquire Signal → Analog Input → Voltage → ai0. Set input range to -10V to 10V and 1 Sample (On Demand). You will also need to set Terminal Configuration to Differential.

The output is classed as Dynamic Data: we need to convert this to a scalar type for further processing. Insert a Convert from Dynamic Data (Express → Sig Manip → From DDT) after the DAQ and set it to Single Scalar output. We can combine the desired and acquired signals into one graph: move the Wfm Chart out of the sequence (but still inside the loop). Bundle the two signals together (Programming→Clusters,Class&Variant→Bundle) and fit the result into the Chart. See Figure 3.2.

In the Front Panel, stretch out the legend of the plot, so you see that there are a Plot 0 and a Plot 1. The order is decided by the order in which you bundled them. Following good practice, change the appearance of the acquired data by showing only the data-points (for this, you can right-click the legend itself).

Pushing the limits: acquiring a low-level signal

This is all well, and the data read by the DAQ are almost identical to the ones we tried to generate. Now try to reduce the amplitude to 0.02 and the offset to 0.025. You should notice that the intended signal is still a smooth sine wave (it's calculated by the PC!), but the acquired points are more stepped. This is known as *digitization (or quantisation) error*. It comes from the process of converting the analogue voltage to a binary number. If the real voltage falls in between two adjacent digital numbers, one or the other will be returned, nothing in between[2]. What we can do, is pre-amplify the signal so that the digitization error is minimized. The USB-6008 will do this

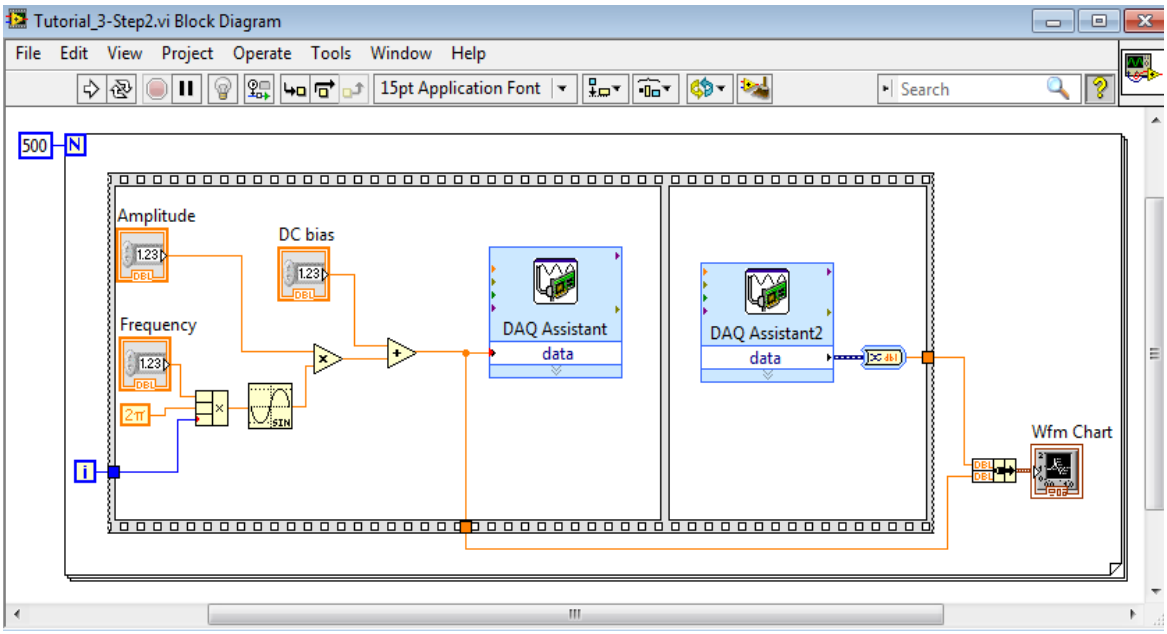

Figure 3.2: Acquiring and plotting data

2 You can check on the USB-6008 that it works with 12bits, corresponding to $2^{12} = 4096$ values. So when the range is ±10V, each step is 20V/4096 ≈ 5 mV.

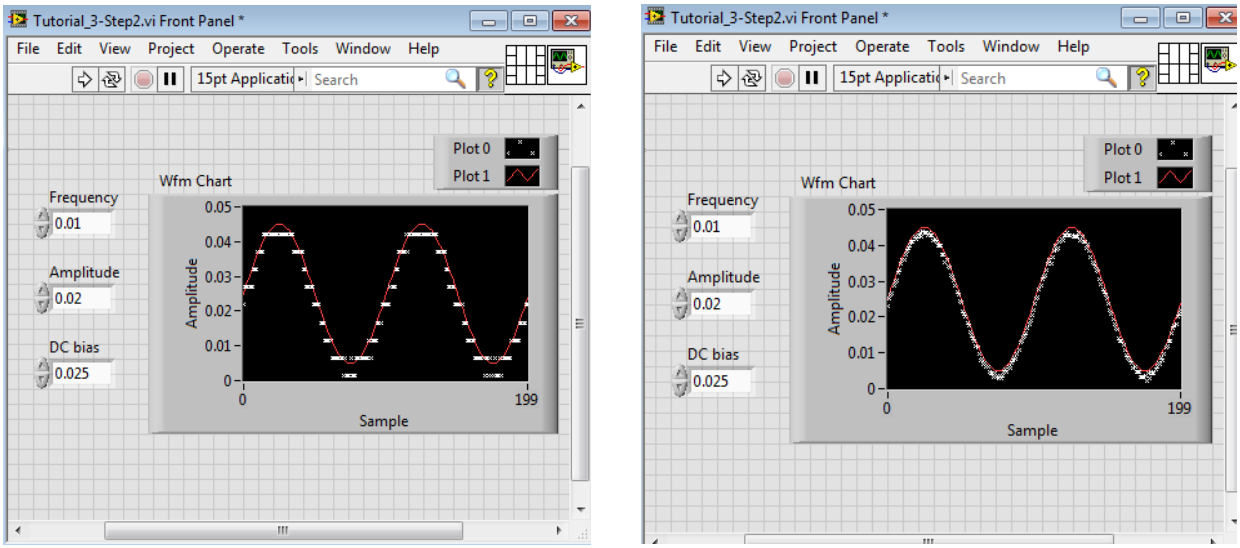

Figure 3.3: Selecting the correct range for the input (by changing the settings within the DAQ Assistant).

automatically, if we only set the correct input range… So, double-click the DAQ Assistant for the acquisition and change the Signal Input range to cover the range you expect for your signal: Max=0.1 V and Min = 0 V. When you re-run the program, you'll see that the steps have disappeared (check Figure 3.3 for what you expect to see). Note that if you set a range that is too small, your signal will be *clipped* (the tops will be cut away): adjust the range to avoid this.

Relevant information is found on p.12 of the User Guide for USB 6008/9:

> **PGA**—The programmable-gain amplifier provides input gains of 1, 2, 4, 5, 8, 10, 16, or 20 when configured for differential measurements and gain of 1 when configured for single-ended measurements. The PGA gain is automatically calculated based on the voltage range selected in the measurement application.

Signal processing

Even if we have selected a better range, the read-in values are not exactly like the signal we wanted to generate. Part of the problem is that what we plot as desired signal is just what we computed, not what actually gets generated (and there is a digitization problem here too). Another part of the problem is that we are measuring very low signals of tens of mV, and electrical noise becomes more important. Dealing with noise is a complex business. First we should do all we can to reduce it, by removing its sources and its ways of getting into our signal. Once we've done our best there, we can use software filtering.

Place a Butterworth filter (Signal Processing → Filters → Butterworth) in your program. Now if you right-click near the top, where the tooltip says "filter type" (the icon must not be selected) you can Create→Constant; set it to Lowpass (in our case the noise is at frequencies higher than the sine wave, so we remove all high-frequency components). Wire your data to the X input. We require two more inputs. By using our For Loop to create the data,

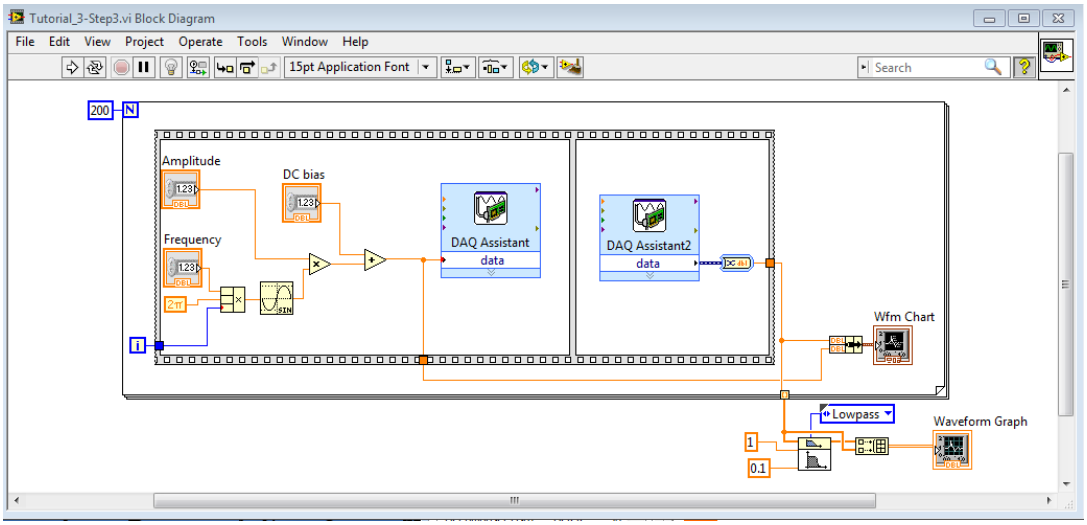

Figure 3.4: Filtering the acquired data to remove noise

our time step is essentially 1 [s] therefore our sampling frequency is 1/(1 [s])=1 [Hz]. Wire a constant 1 to this input. As for the LowCutOff frequency, our wave has frequency of 0.01, so we can put a bit more, like 0.1, to make sure we don't attenuate it and, at the same time, we cut off as much noise as possible. It is good to keep an eye on the effect of the filter, so create a waveform graph and place it outside the For Loop. A Wfm Graph can plot two traces, provided we combine the two arrays into a matrix. Use the Build Array for this (Programming→Array→Build Array). Your program should look like Figure 3.4

Now experiment the effects of the cutoff frequency, trying a few values down to 0.01 and also a few up to 0.499. At low values, you should notice a smoother curve, however it becomes smaller (attenuation) and shifted to the right (phase shift). At higher values, the filtered and unfiltered data are very similar because we don't really cut away anything. When filtering, you have to choose a good cut-off frequency to compromise between cutting noise and distorting your signal.

Exercises

if you can't finish these within the hour, move to Tutorial 4; exercise 3 can be completed without the DAQ using the modifications in Figure 3.5.

1. Remove the wire between GND and AI4. Restore the Amplitude to 2 V and the DC bias to 2.5 V. In the DAQ Assistant for acquisition change the input range to -10, 10 V, but leave the mode to Differential. Is there any change in the data acquired? can you explain it?
2. Having removed the wire in the previous exercise, go back to the DAQ Assistant2 (acquisition) and change the mode to RSE. What is happening now? Discuss the difference of results between this condition and the previous exercise.
3. Add appropriate code to calculate and display the power spectrum of the signal you have acquired.

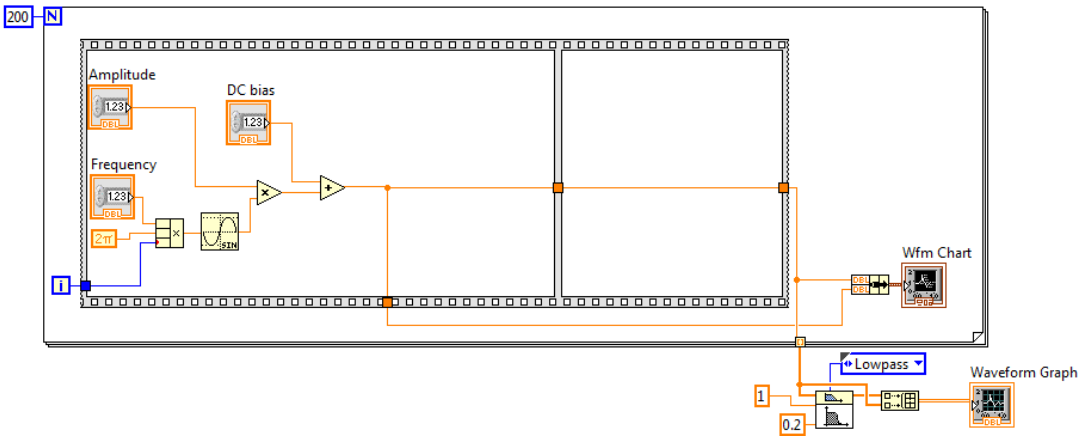

Figure 3.5: Practising without the USB 6008.

Tutorial 4: Saving acquired data

Learning outcomes

By the end of this exercise, you should be able to write LabVIEW programs that:

- use Case structures;
- save data to a file for further processing.

Introduction

LabVIEW can be very useful to quickly develop a data acquisition system or to control a process by generating control signals, as we've seen in the previous tutorial. With LabVIEW, it is very easy to introduce graphs that show in real time the data acquired, which is important to give you a feeling of how the experiment is going. We have also seen some signal processing (specially FFT and filtering) which can be valuable to highlight the features of a phenomenon you are measuring. For example, you may be looking at a vibrating beam and you are interested in the effect that temperature has on the frequency. In this case, you can get LabVIEW to plot a spectrum of the signal, so that you can immediately see if anything is happening. Experience teaches that even a small amount of signal processing in real time can be invaluable for fault-finding.

On the other hand, you'll typically want to (and should) do the proper signal processing, analysis and graphing *offline*. That is, back in your office, with applications like MATLAB.

Generate a vector of pseudo-time points

In the previous tutorial we created a sinewave of a given frequency. You might have noticed that the real frequency generated is not what you had set in the panel. Indeed, when you set 0.01, it was not 0.01 Hz, otherwise it would have taken almost two minutes to generate and acquire one period. We did not add any timing, so that we let the program run as fast as it could. In the next tutorial, we'll use fast data acquisition and we'll use the true time. For the moment, we just want to add a pseudo-time column to the data we save to disk.

To do this, remember that our code is generating this signal:

$$V_{out} = V_{bias} + A \sin(2\pi f i)$$

where i is simply the loop index, which counts 0, 1, 2, … This is our "time". Add another For Loop, with the same count as the first loop (as we always want these numbers to be the same, it is good practice to have only one constant and wire that one to both loops, as in Figure). The loop is quite empty: we're simply using the indexing of the tunnel to build an array.

Write Delimited Spreadsheet.vi

Now select Programming→File I/O→Write Delimited Spreadsheet.vi We want to have time and data next to each other, so we combine the two arrays into a matrix (which is a 2D array): Programming→Array→Build Array. The first

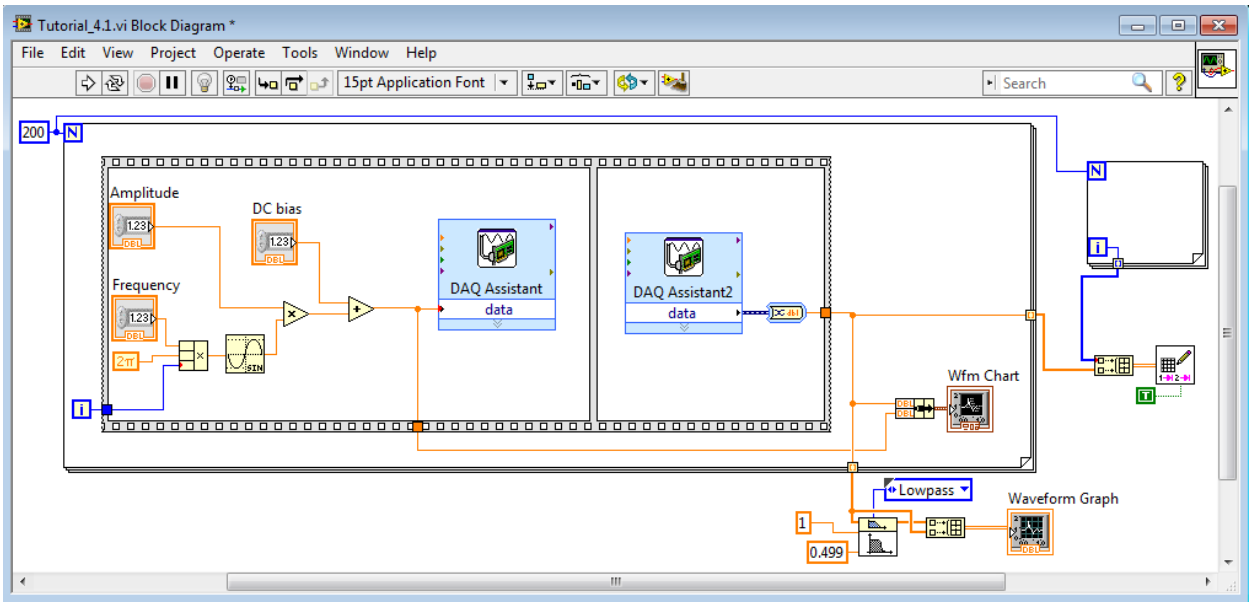

Figure 4.1: Saving data to a spreadsheet file

input is the “time”, the second is the data. The output of this block goes into the 2D data input of the Write Delimited Spreadsheet (it also has a 1D data input if you're saving a simple, 1D array). To complete, right-click on the lower edge of the Write VI, where it says *transpose?*, Create a constant and then click on it to toggle it from False to True. If we don't do this, LabVIEW will save two rows of data, instead of two columns. You program should look like in Figure 4.1.

When you run the program, at the end of the data acquisition it should ask you where to save the file (you should use extension .txt for your file). By default, LabVIEW uses a tab character to separate columns: MATLAB is very happy with such files and will recognise them and read them in without any problem.

Fine tuning the output file

Use a text editor (like notepad++) to open the file you created. Notice that there are only 3 digits after the decimal point. These are not enough in this case as the signal is so low: we have lost information. Go over the top of the Write Delimited Spreadsheet VI till you see a connector called *format*. This specifies how the decimal numbers are translated to text. By default this is **%.3f**, which means “floating point number with 3 digits after the point”. To ensure we have a good number of significant digits, create this string constant: **%_4g** (see Figure 4.2)

With this format, numbers may be written in exponential notation, but MATLAB is still happy with them.

At this point, review the difference between “precision”, in the sense of digits after the point, and significant digits.

Do you really want to save? Add a «Save?» switch.

Our program just saves data without asking if we want to. It's true experimental data are valuable and you don't want to waste them, but sometimes you're just trying out something. So it would be nice to have a way to decide, before running the program, if we want to save data or only display them.

To achieve this, we'll use a new construct: the Case Structure, which is similar to the If Then you may have met in other computer languages. Right-click the diagram, select Programming→Structures→Case Structure and draw it around the blocks which we added to save to file. On the left border, you'll notice a small rectangle with a green «?» inside. Right-click it and select Create Control. An icon representing a button will appear, and a switch also appears in the Front Panel. Click the arrows on the top border of the Case Structure to verify that when the input is True we are saving to file and when it's False we're doing nothing (the frame is empty!). By right-clicking near the top indicator you can change several things, for example you can easily swap the True and False cases. Take a look.

Run the program to check it behaves as you want. Note that the case displayed in the Block Diagram has no relevance to the behaviour during execution: you use the switch on the Front Panel to save or not to save.

Tired of typing in filenames? Auto-naming of files

Another thing that happens often in the lab, is that you want to run the program many times, every time you change something in your apparatus. And then, of course, you want to save all these data, so you end up typing in many file names. It is therefore convenient to have an auto-naming code within your program.

Make some space by moving the Case Structures to the right (remember that your code should not only work, but also be as tidy as possible). Right-click on the Diagram and select: Programming→String→Concatenate Strings; drop this new icon in the space you made. Right-click the top left corner and select Create→Control. Rename it as *Prefix*. A string control has appeared in the Panel, move it close to the switch and type *Tutorial4* inside it. Now, let's add a

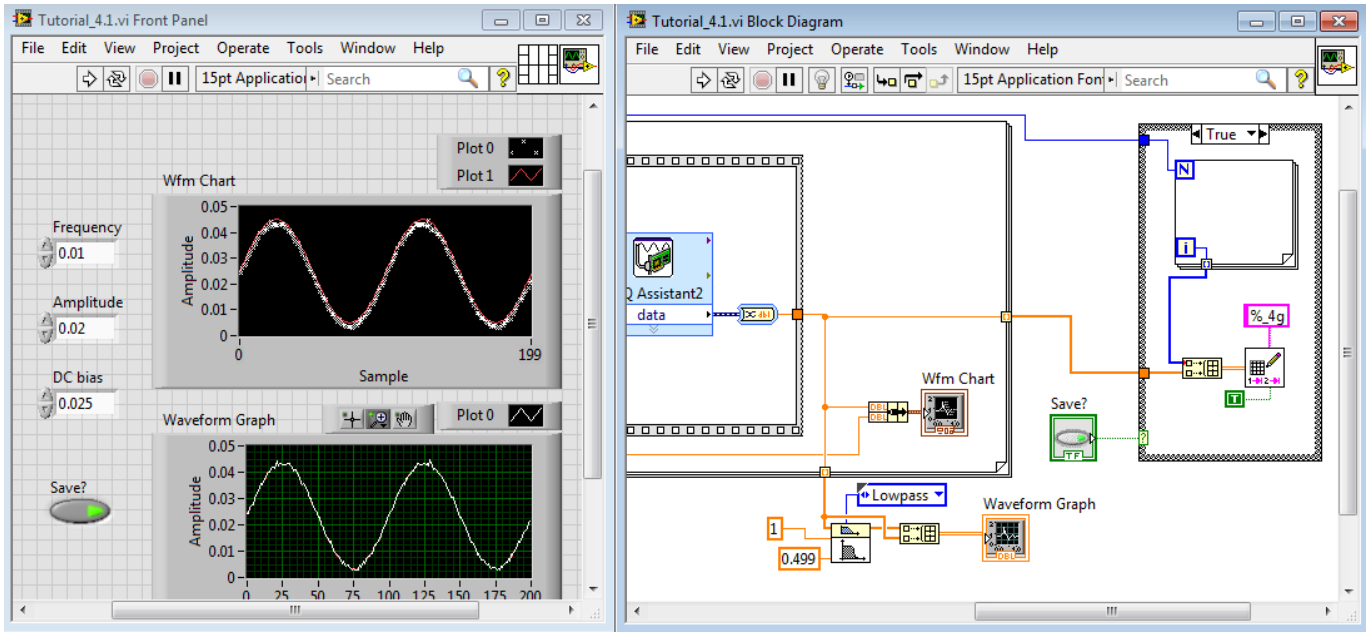

Figure 4.2: Add a switch to select save or not save

timestamp to the filename. Right-click the Diagram and select Programming→Timing→Format Date/Time String. Right-click top left to select Create→Constant and type in `_%Y%m%d_%H%M%S.txt` . This is a formatting string that tells LabVIEW how we want to represent the current time (%Y means full year, %m month, etc.), we are also using it to add an extension to the filename and some separators. You can customise it, but this is approximately the ISO standard and is very good to help you sort files by date/time.

We're almost there: we now have a filename, but LabVIEW needs an absolute path to know where we want to put the file. So add Programming→File I/O→File Constants→Default Data Directory. Now we concatenate this with the filename we produced before, using Programming→File I/O→Build Path. And finally we connect this output to the filename input of the Write Delimited Spreadsheet VI, as in figure. Run the program to test it.

Where is my file? We have used the Default Data Directory, so that's where your file is. To find out or change this directory, use the main menu in the Diagram to select Tools→Options→Paths and then Default Data Directory from the drop-down on the right. You need to restart LabVIEW for changes to take effect.

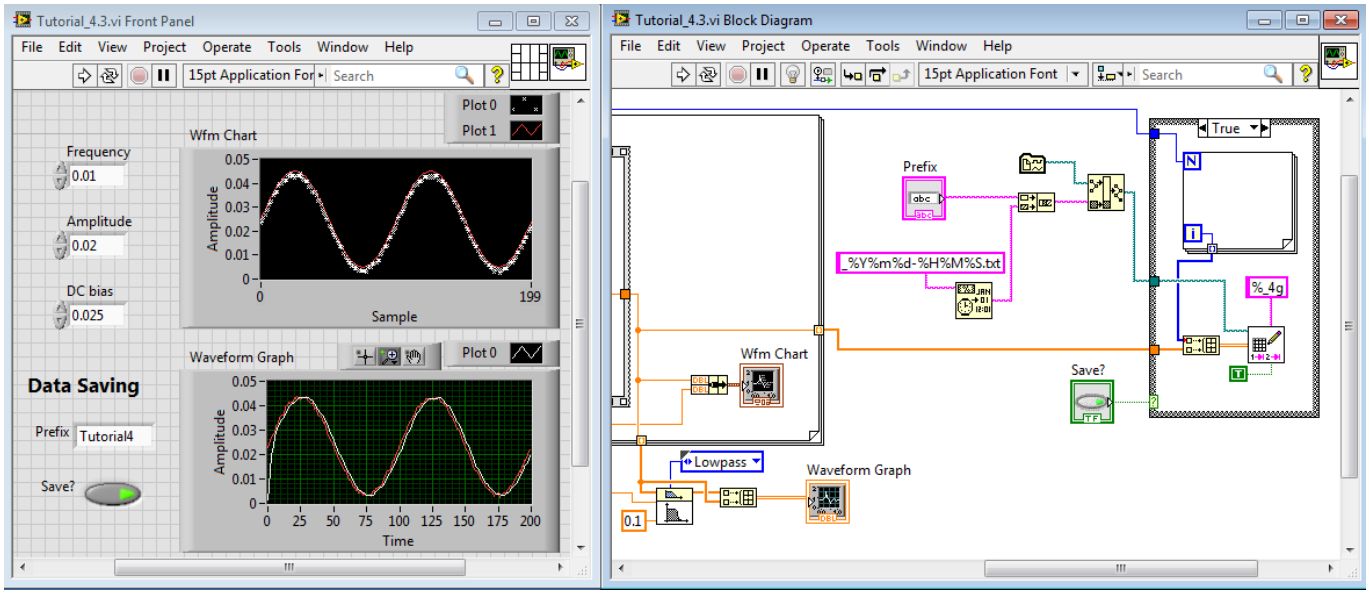

Figure 4.3: Automatically naming and saving data files

Exercises (if you can't finish these today, complete them later in the Robinson Library, see Figure 3.5)

1. Modify the program so that the user is asked for the absolute path where to save the files, rather than using the Default Data Directory. This is a very desirable feature of your program, as you may want to keep data from different projects in separate folders.
2. Add an indicator to the panel that shows the filename to which data were saved. This is also extremely desirable, as it shows the filename that you will want to record in your logbook, otherwise you'll have a long list of files and … can you remember what were the settings of the apparatus when you ran the acquisition at 15:34:54 of that day?

Tutorial 5: Acquiring Sensor Data

Learning outcomes

By the end of this exercise, you should be able to:

- connect an unearthed sensor to the DAQ;
- acquire signals from a real sensor

Introduction

In this tutorial we'll prepare a LabVIEW program to acquire the signal from a sensor. Sensors can be externally powered, in which case you need to make sure the ground of their supply is the same as the supply of your acquisition card (the PC in our case). Other sensors are self-powered, in the sense that they generate a signal without need for external power. We are going to use one such sensor, based on the piezoelectric effect.

Connecting the hardware (refer to Introduction Lecture's slides in BB for photo-sequence)

The most important points to remember are:

- the sensor is a piezoelectric cantilevered beam. It is moderately delicate, so treat it with care and avoid bending it too much
- carefully press the pins of the sensor into the breadboard until the thin sections are fully inserted. Use the central area of the board, not the power lines at the sides
- insert two wires in the holes next to the sensor pins, staying in the same line (e.g. if one sensor pin is in hole e5, the wire can go in c5)
- without over-tightening the screws in the terminal, connect the two wires from the breadboard to inputs 2 and 3 on the USB-6008, corresponding to AI0 and AI4. Connect the wire coming from the + on the cantilever to AI0 (this is not important).

Acquire data from the sensor as supplied

Once you have connected the USB-6008 to the PC, create a new VI and place a DAQ Assistant in the Block Diagram. Leave it configured for Differential Acquisition; Acquisition Mode should be N Samples and we'll read 5k samples at 1 kS/s (1000 Samples per second), hence the acquisition will last 5 seconds.

Create a Waveform Graph and connect the data from the DAQ to it. We'll then calculate the spectral content of the acquired signal, as we are most interested in the natural frequency of the cantilever. Check the Diagram in **Figure 5.1**, you should recognise all elements from previous tutorials. There is no need to create manually the conversion from DDT to Array of Doubles as LabVIEW knows what is needed: just connect the two points and LV will insert the converter. Note that we have wired 0.001 in the dt input, as we are acquiring at 1 kS/s so samples are separated by 1 ms (the USB 6008 can operate at up to 10 kS/s if only one channel is used).

Check that everything is connected as described. Launch the program and in those 5 seconds of acquisition, *pluck* the cantilever: deflect it downwards with the tip of the screwdriver and let go by moving the screwdriver backwards or sideways. Note that you must not re-accompany the cantilever upwards, it must be free to vibrate. It may take you a few tries to get a good signal; note that you are more likely to get a nice clean response if you don't deflect it too much (which would induce non-linear behaviour). Two to four millimeters is probably good. With some chance, you should get curves similar to those in the figure (after zooming in).

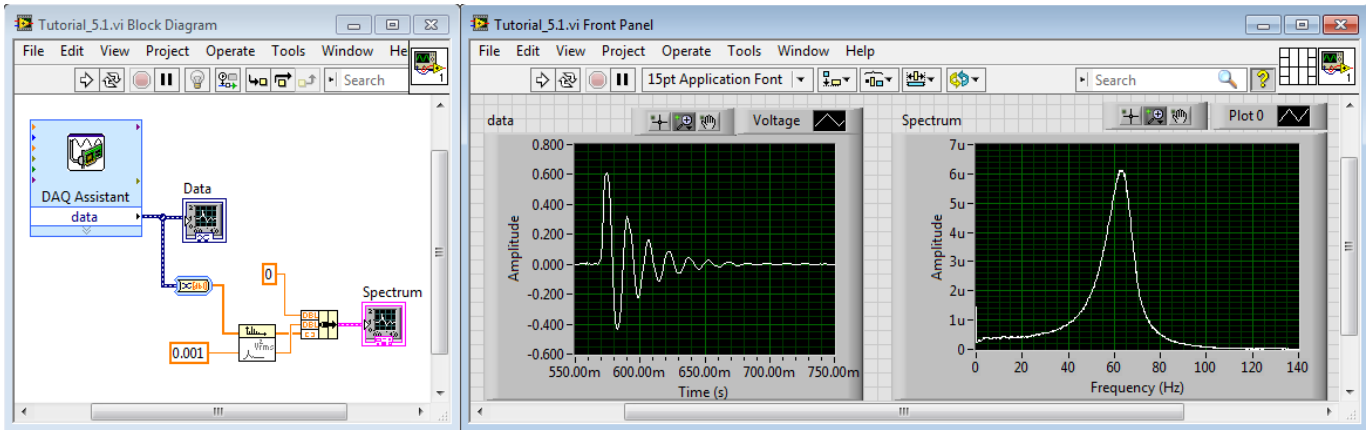

Figure 5.1: Acquiring the signal from the vibrating cantilever.

The cantilever is approximately behaving as a single degree of freedom oscillator. This is helped by the relatively large mass crimped at its tip, so that the distributed mass of the beam itself is less important and higher modes less likely. There are also significant non-linearities (of geometric and material origin), but we can ignore them, especially if you deflect the cantilever only a little. So, when you push down the tip, you load the spring which is the beam itself; when you release, the restoring force accelerates the tip mass upwards. And then there is the usual exchange of energy between mass (kinetic e.) and spring (elastic potential e.). Damping is provided by internal "friction" in the materials, air and the conversion to electricity. With some calibration, you could determine the effective mass at the tip. This has already been done for you and you can accept that the mass is **m = 310 mg**. You'll need this value later on.

Calculate the spring constant

You have obtained nice plots. Let's get some information from them. Look carefully at your data. Zoom in onto the peak in the Spectrum and determine as best as you can (and as best as it makes sense with the experimental uncertainties present) the frequency at which the peak occurs. This will be the (damped) natural frequency. Use the formula for a simple harmonic oscillator (we neglect altogether damping in this tutorials):

$\omega = \sqrt{k/m}$ to calculate the corresponding stiffness of the beam (remember to work in SI and that ω is in rad/s).

You should get something like 50 N/m.

Make note of the peak positions (with mass now and without mass later, during the assignment work), so you can do the calculations of part 3 (see later) whatever happens.

Measuring mass

The *m* and *k* parameters we derived above are decided by the material and construction of your cantilever. Within the approximations we are using, if we add more mass (*M*) to the tip, we're not changing the stiffness of the beam. So, the new frequency will simply be given by:

$$\omega_{loaded} = \sqrt{\frac{k}{(m+M)}}$$

so we can expect that if we add a mass, the peak we've just detected in the spectrum will shift to the left (i.e. occur at lower frequencies).

This principle is at the basis of the quartz balance and is also used in advanced bio-sensors. In the latter, we are measuring the tiny mass of a bunch of molecules and extra tricks are also needed to ensure you minimize the effects of temperature, residual stresses, etc.

Module assignment: Determination of unknown mass

Aim of the assignment
Use the resonance frequency shift of a cantilever to determine an unknown mass

Objectives:
- develop a program that acquires the time-domain signal from the cantilever, performs suitable signal processing, plots appropriate graphs and saves the data to file
- prepare a plot in MATLAB to show the frequency shift
- use the frequency shift to calculate the mass. Discuss the results.

Read all instructions in full before starting to work.
Submit your program *(the .vi file) via Blackboard (MEC8043 → Assessment → LabVIEW Program) before the official end of your last session. Test your program to make sure it works.*
Submit your report *(printed) to the General Office on the first teaching Monday of the New Year. The report includes three parts for a total of 20 marks. Use the template from Blackboard. Use the Snipping tool or IrfanView to get screenshots of Front Panel and Block Diagram; ensure that text is readable when printed (use PNG not JPEG).*

1. Program (Front Panel and Block Diagram) {10 marks}
Start from the program in Tutorial 5; modify it to satisfy these requirements:
- reconfigure the DAQ Assistant to optimise the input range (report your settings in the submission sheet).
- data are saved to a spreadsheet file; a box allows the user to set a prefix to the filename; a timestamp is included; the resulting filename is visible on the front panel
- the program is able to save time domain and frequency domain data (spectrum); the user can activate or deactivate each of these savings independently. Appropriate filenames are generated (for time domain and spectrum), without user intervention.

The code must be clear and tidy, the corresponding front panel should be pleasant to the eye and easy to use (choose meaningful labels for controls and indicators, perhaps add short instructions and decorations if suitable as well as brief comments to the code).

Important: in your submission the front panel must show the acquired data in the graphs, hence the screenshots must be taken during your last session. The program in your report must match exactly the one you submitted as .vi. ***No one*** *will be allowed to use the DAQ after their last session.*

2. Spectrum graph {5 marks}
Plot on the same graph the two spectra (with and without mass) as calculated and saved by LabVIEW. This is a relatively simple component of the submission, but we want you to pay attention to a correct, informative and neat graphical representation of experimental data. Clearly identify the two curves (don't use just colours, which fail if in black/white!); label axes appropriately; choose appropriate axes limits; make sure text is readable and data points are visible.

3. Calculation and Discussion {5 marks}
You must:
- use the formula and data given in Tutorial 5 to estimate the unknown mass
- report the acquisition parameters, by filling in the appropriate box on the template
- briefly (only the first 100 words will be marked!) discuss your results considering the main sources of uncertainties and how you could improve the measurement system (sensitivity, resolution, reproducibility). Use your experience of plucking the beam several times (with and without additional mass) to try to determine what factors affect the measured frequency.