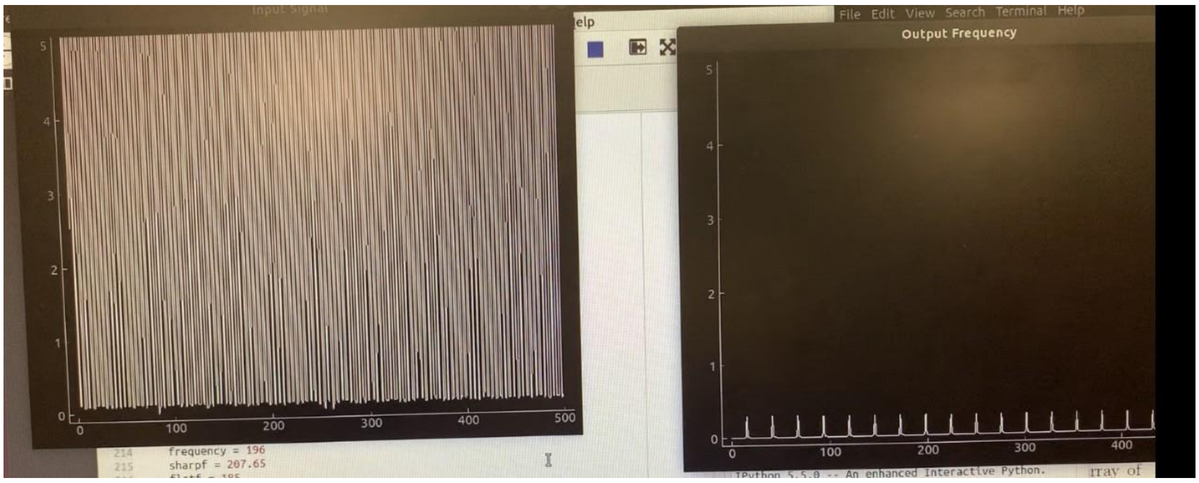
Digital Signal
Processing

Assignment 3: IIR Filters

For this assignment, it was decided to make a violin tuner. Using infinite impulse response, *IIR*, filters, the audio signal of each string of a violin was measured in realtime in order to determine the predominant frequency. To use our tuner, the user types the note of the string they would like to tune into the terminal. Using saved frequency values, the tuner then identifies the frequency of the note required and filters the input audio signal through a bandpass IIR filter to only pass when the frequency of the signal is close to the note they are trying to find. The output signal of the bandpass, is then loaded into a buffer. The fast Fourier transform (FFT) is taken of the buffer at regular intervals and this spectrum is then plotted on the screen continuously. As a result, the output plot shows continuous frequency peaks which increase in amplitude as the input signal moves closer to the required frequency.


Input Signal
Output Frequency
File Edit View Search Terminal Help
0
1
2
3
4
5
100
200
300
400
500
IPython 5.5.0 -- An enhanced Interactive Python.

```
frequency = 196
sharpf = 207.65
```

Figure 1- Realtime Input and Output Signal Plots

Analogue Circuit for Measurement

To process audio signals, the USB-DUX had to communicate with the outside world. To achieve this, an analogue circuit was developed which accepts an auxiliary input, amplifies that signal, and passes it onto the USB-DUX for analogue to digital conversion and processing.

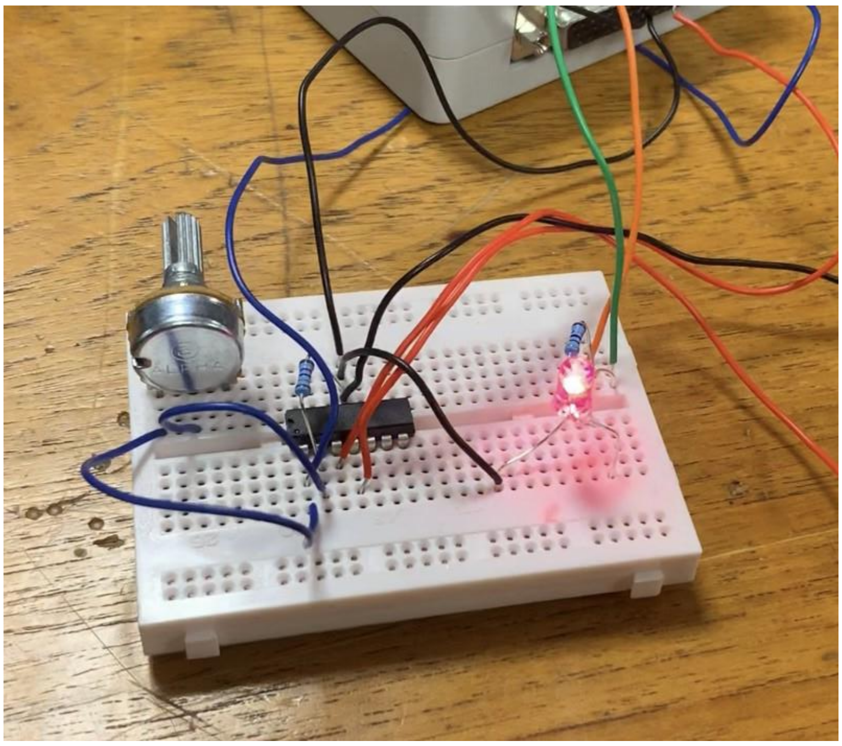


Figure 2- Analogue Circuit

A 3.5mm mono jack was connected to the noninverting input of an op amp, the inverting input of which was connected to ground via a 13kΩ resistor. A 470kΩ potentiometer provided negative feedback, the amount of which, and therefore the gain, being altered by adjusting the potentiometer's resistance. A potentiometer was chosen over a fixed resistor to allow for quick volume adjustments, compensating for various strengths of input signals from different sources.

This setup allowed for various means of getting audio signals to the USB-DUX, such as the output from an electric violin, a microphone, or a function generator. It was decided to include a mono jack rather than just a microphone to prevent microphone distortion where possible, which allowed for greater confidence when calibrating the filters with high quality input frequencies for reference.

Rather than connecting a mono jack to the USB-DUX directly, it was decided to first amplify any audio signals using an operational amplifier, to maximize the USB-DUX's limited resolution.

A simple schematic of the circuit is shown below. The specific operational amplifier used was a LM3245N, which was chosen due to its good frequency response for audio applications and as it was already to hand.

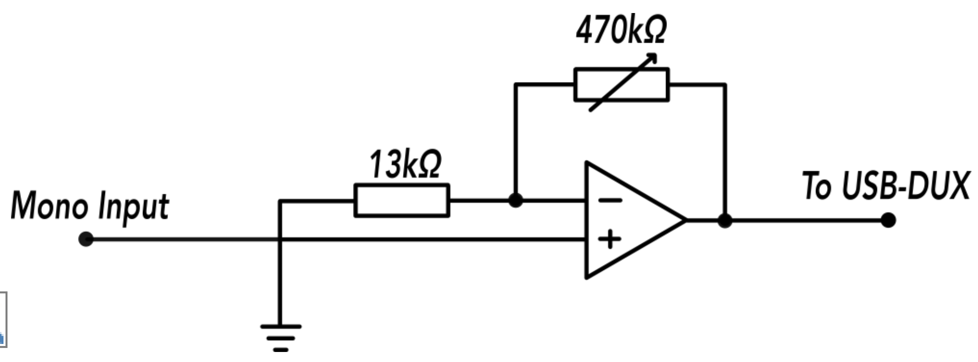


Figure 3 - Amplifier Circuit

Hardware for Visual Indication

To indicate if a string was correctly tuned, a tri-colour LED would light up. Helping a musician tune their instrument, if the string resonated at the exact frequency it aimed to, the LED would light up red. If instead, the frequency was slightly off, the LED would display blue light. Finally, if the string sometimes hit the correct frequency but not always, the LED would appear as magenta, indicating that the string was very close to being perfectly tuned.

The circuit that allowed for this is as follows.

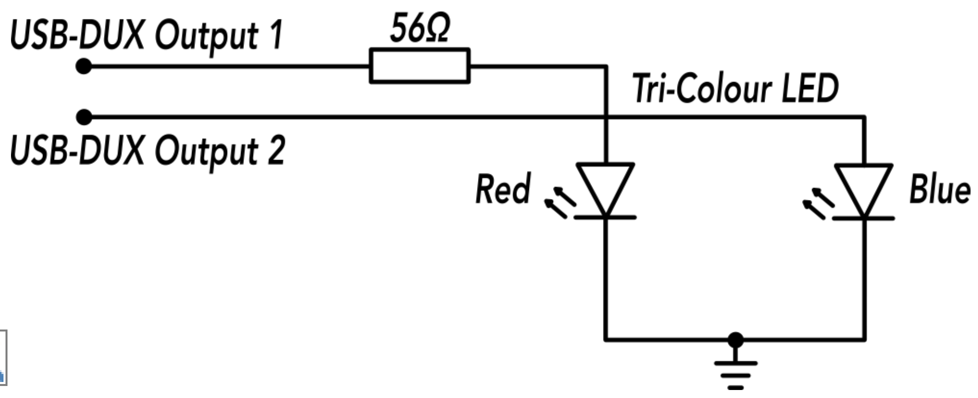


Figure 4 – Visual Indicator Circuit

Filtering

To filter for the correct frequency, the input signal was passed through a bandpass filter. The SOS coefficients were found by the Butterworth filter function from the module 'scipy.signal' The high and low frequencies for the bandpass were chosen to be +/- 11 from the selected frequency, found to be the frequencies within the note range, before the note changed (e.g. from A to A#). These calculated numbers were then normalised as the Butterworth filter only takes numbers between zero and one as its arguments.

```
wide= np.array([(frequency-11)*norm, (frequency+11)*norm])
sos_wide = sig.butter(10, wide, btype='bandpass', output='sos')
```

The calculated SOS coefficient array was then passed to the IIR_filter class to create the filter.

In order to create a chain of 2nd order IIR filters, the class IIR_filter was designed. It takes the SOS coefficient array as an input, and then passes each row to a different 2nd order IIR array consequentially. It also passes the output signal of the first array to be the input signal of the following array continuously until it reaches the end of the filter chain, as shown in Figure 5.

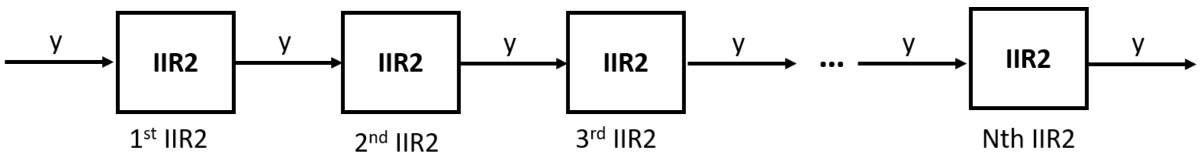


Figure 5- Chain of IIR filters

The filter design of the second order SOS filter was designed in the class IIR2_filter by following the flow diagram shown in Figure 6. The values b0, b1, b2, a1, and a2 were taken from a row of the SOS filter array.

```
def dofilter(self,x):
    #accumulator for the IIR part
    input_acc = x
    input_acc = input_acc - (self.a1*self.buffer1)
    input_acc = input_acc - (self.a2*self.buffer2)

    #accumulator for the FIR part
    output_acc = input_acc*self.b0
    output_acc = output_acc + (self.b1*self.buffer1)
    output_acc = output_acc + (self.b2*self.buffer2)

    self.buffer2 = self.buffer1
    self.buffer1 = input_acc

    return output_acc
```

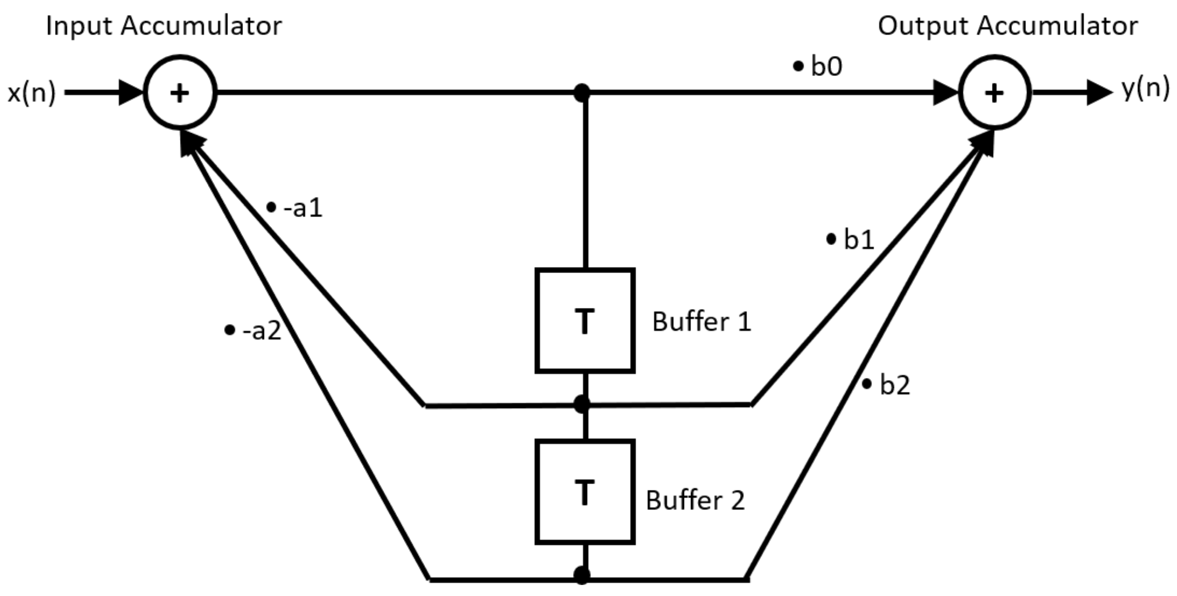


Figure 6- Flow diagram of 2nd order IIR Filter

After the filter was created, it was tested for all four frequencies to check that the correct range was being passed. A chirp signal was used as an input. The filter responses are plotted below.

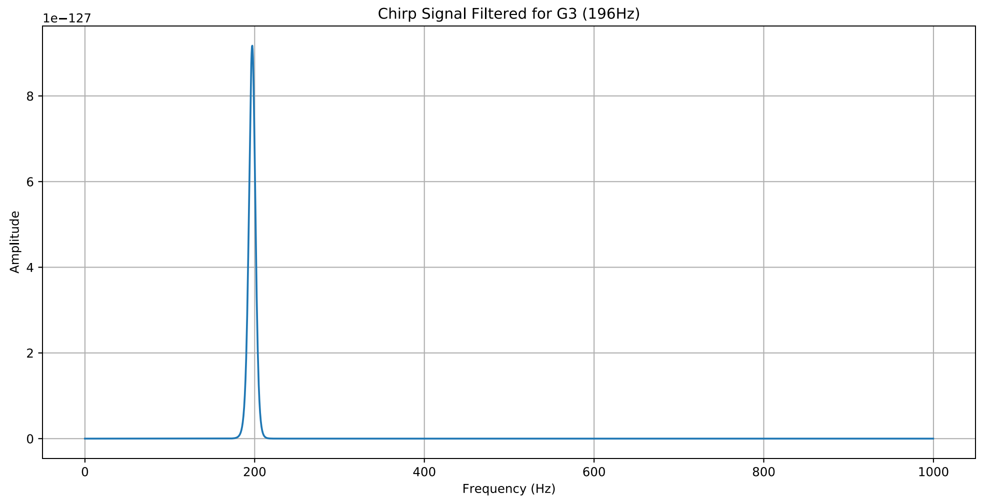


Figure 7- Response of G Bandpass filter

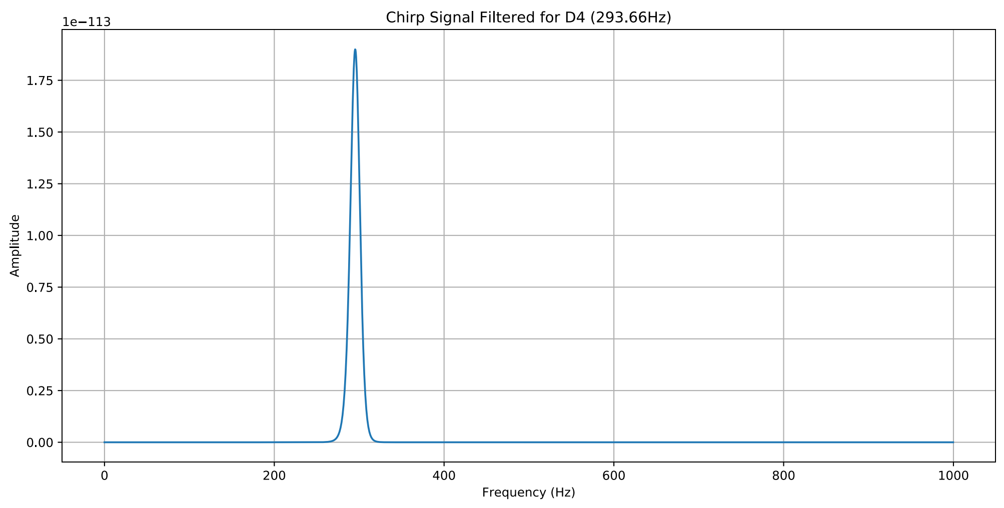


Figure 8- Response of D Bandpass filter

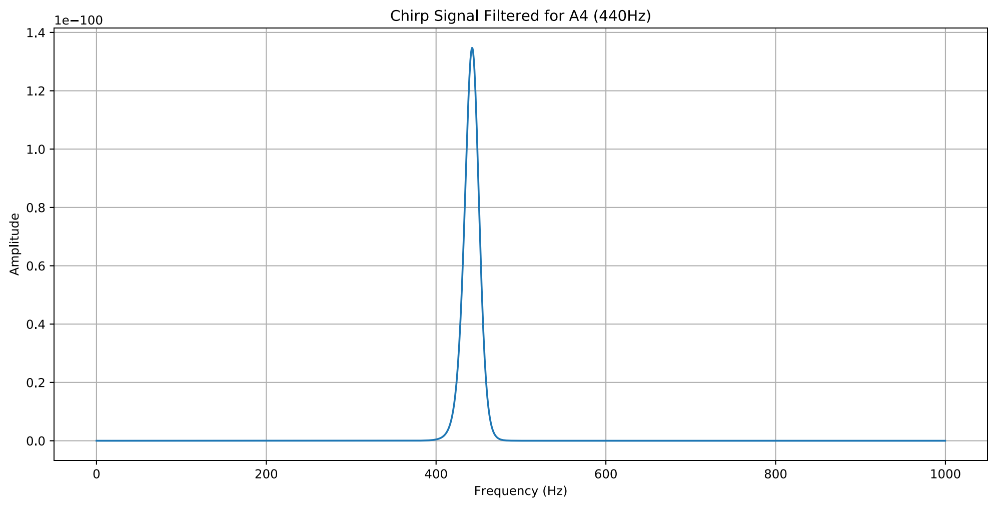


Figure 9- Response of A Bandpass filter

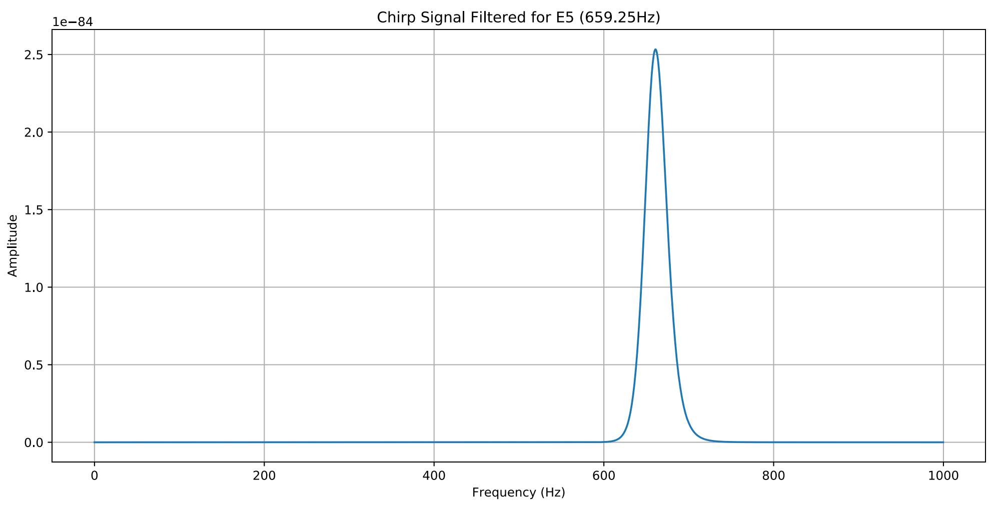


Figure 10- Response of E Bandpass filter

Code

The Python programme is used to sample, filter and plot the input audio signal. The layout of the software is shown in Figures 11 and 12.

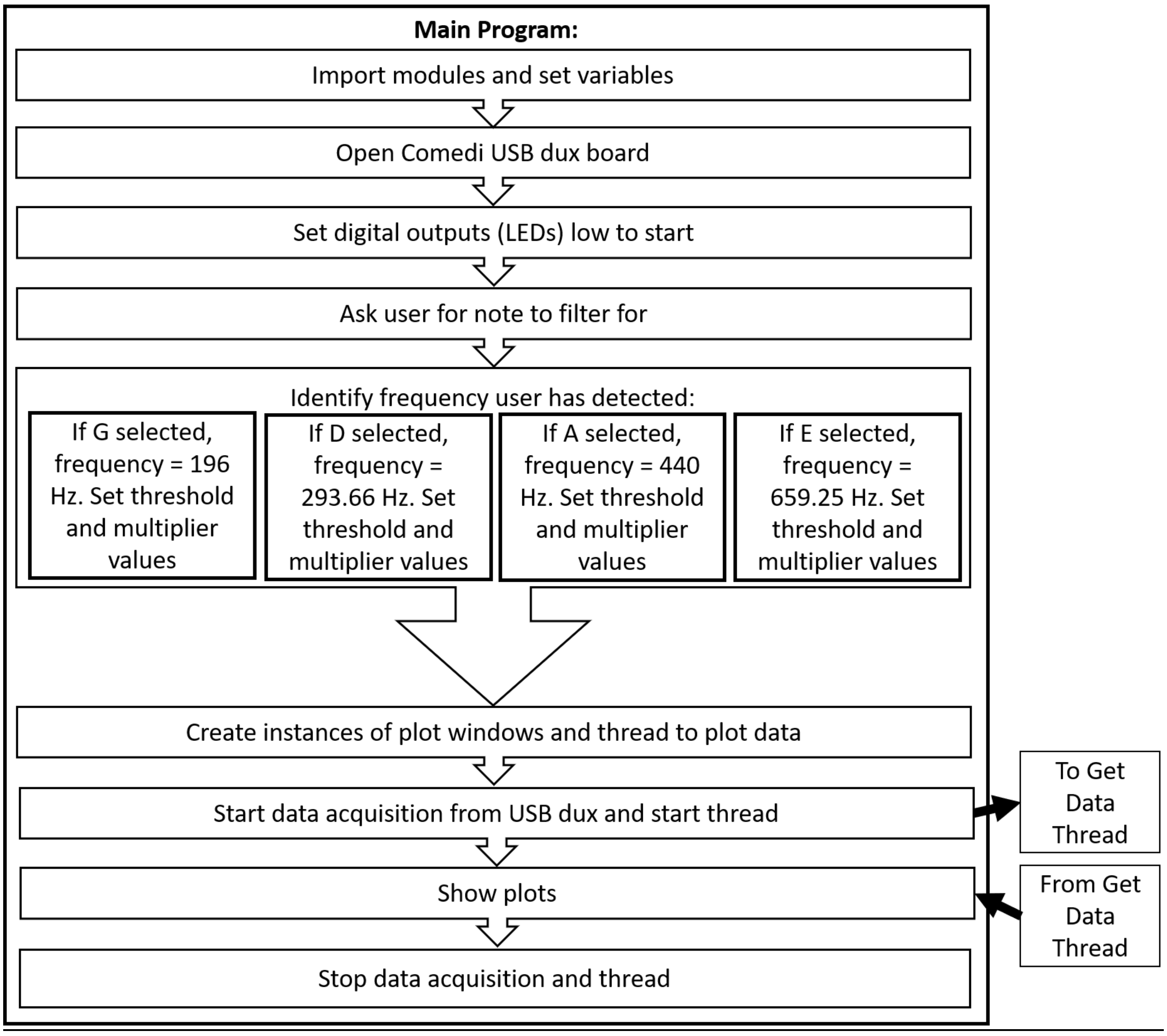

Main Program:
Import modules and set variables
Open Comedi USB dux board
Set digital outputs (LEDs) low to start
Ask user for note to filter for
Identify frequency user has detected:
If G selected, frequency = 196 Hz. Set threshold and multiplier values
If D selected, frequency = 293.66 Hz. Set threshold and multiplier values
If A selected, frequency = 440 Hz. Set threshold and multiplier values
If E selected, frequency = 659.25 Hz. Set threshold and multiplier values
Create instances of plot windows and thread to plot data
Start data acquisition from USB dux and start thread
To Get Data Thread
Show plots
From Get Data Thread
Stop data acquisition and thread

Figure 11- Main Programme

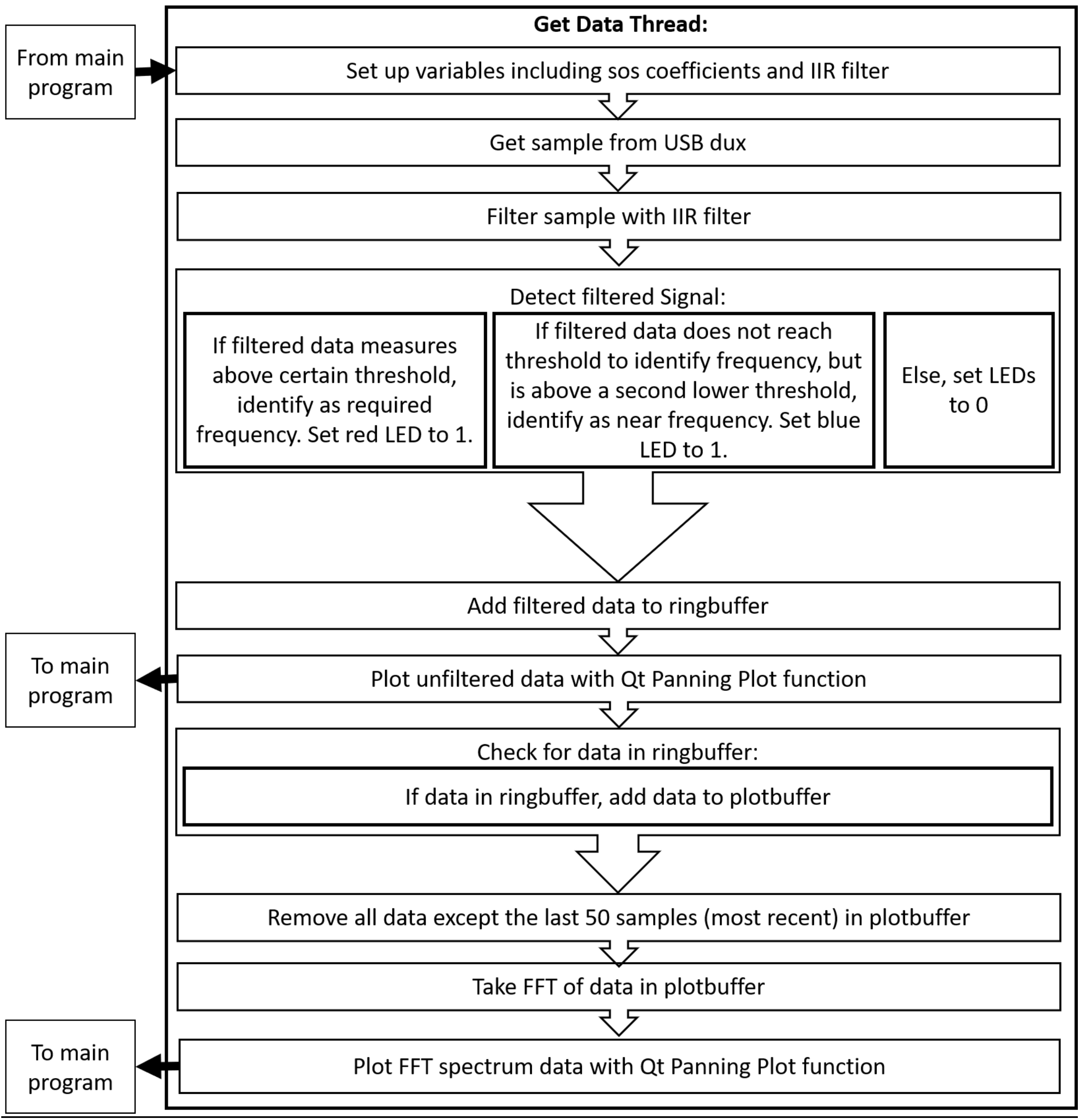

Get Data Thread:
From main program
Set up variables including sos coefficients and IIR filter
Get sample from USB dux
Filter sample with IIR filter
Detect filtered Signal:
If filtered data measures above certain threshold, identify as required frequency. Set red LED to 1.
If filtered data does not reach threshold to identify frequency, but is above a second lower threshold, identify as near frequency. Set blue LED to 1.
Else, set LEDs to 0
Add filtered data to ringbuffer
To main program
Plot unfiltered data with Qt Panning Plot function
Check for data in ringbuffer:
If data in ringbuffer, add data to plotbuffer
Remove all data except the last 50 samples (most recent) in plotbuffer
Take FFT of data in plotbuffer
To main program
Plot FFT spectrum data with Qt Panning Plot function

Figure 12- Get Data Thread Function

Results

The tuner successfully filtered the incoming signal, with an output signal only occurring when the signal was in the correct frequency band. The closer the signal was to the desired frequency, the higher the output peak. In this way the frequency was able to be measured. However in practical terms, i.e. lighting the LED depending on these values, it could be a little unreliable to tell if the instrument was in tune as the strength of the output signal varied somewhat between strings so each one needed to be multiplied by a different number and required some 'tuning' of its own. A threshold value to turn on the LED was set for all the frequencies, but it probably would have been more sensible to set individual threshold values due to the variation in signal strengths.

The filters were calibrated by using reliably produced sinusoidal waves created through a trusted function generator app. For each note, a sinusoidal wave of ideal frequency was fed into our amplifier circuit through an auxiliary cable. The filters' response to this perfect pitch was analysed. Then, by increasing and decreasing the frequency of the sinusoidal waves around the ideal, within the app, the filters were able to be fine-tuned to an acceptable tolerance.

After testing the filter with sinusoidal waves, it was then tested with a violin, using a bought tuner to determine how close the violin note was to the desired frequency to check the LEDs were lighting at the correct time. The tuner was found to be success although a little difficult to use as occasional lags in the programme meant that the LEDs sometimes reacted a quite slowly. To solve this problem, it would probably have been a good idea to move some of the plotting and updating that occurs in the 'Get Data Thread' into a separate function in order to possibly include some sleep time when collecting samples. The programme also seemed to lag more when the LEDS were incorporated so perhaps the design could be modified to find a way around this.

There are several further ways in which the programme could be developed and improved. For this experiment we decided to design the tuner for the violin as for tuning we only need to consider four frequencies, the violin strings. However, it could also easily be expanded to determine a broader range of frequencies, not only those specific to the violin in order detect any note that may wish to be tuned.

Furthermore, for tuning it would be helpful to be able to determine if a note was too sharp or too flat. Potentially this could be done by passing the output of the IIR filter through digital high and lowpass filters to determine which side of the desired frequency the detected signal has fallen on.

If there was more time to work on this project it might also be interesting to create a GUI interface to show which note has been detected, or perhaps to display this information on an LCD screen.

Overall, our device successfully solves a real-life problem by filtering a measurement in realtime and allows for greater expandability and circuit refinement in the future.

Appendix

realtime_iir_main.py:

```
import threading
import sys
import pyqtgraph as pg
from pyqtgraph.Qt import QtCore, QtGui
import numpy as np
import pyusbdux as c
import iirclass as iir
import scipy.signal as sig

#set up variables
```

Digital Signal
Processing

```
ringbuffer = []
fs =1345

#normalise
norm = 2/fs

################################################################################

#Set up Qt Panning Plot class
app = QtGui.QApplication(sys.argv)
running = True


channel_of_window1 = 0
channel_of_window2 = 0

class QtPanningPlot:

    #set up Panning plot
    def __init__(self,title):
        self.win = pg.GraphicsLayoutWidget()
        self.win.setWindowTitle(title)
        self.plt = self.win.addPlot()
        self.plt.setYRange(0,5)
        self.plt.setXRange(0,500)
        self.curve = self.plt.plot()
        self.data = []
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.update)
        self.timer.start(100)
        self.layout = QtGui.QGridLayout()
        self.win.setLayout(self.layout)
        self.win.show()

    def update(self):
        self.data=self.data[-500:]
        if self.data:
            self.curve.setData(np.hstack(self.data))

    def addData(self,d):
        self.data.append(d)

################################################################################

#create function to get, filter and plot data
def getDataThread(qtPanningPlot1, qtPanningPlot2, ringbuffer, frequency, n):

    ################################################
    #set up variables

    #normalise (Wn for butterworth filter is in half-cycles / sample)
```

```
 norm = 2/fs
 plotbuffer = []

 #calculate coefficients
 wide= np.array([(frequency-11)*norm, (frequency+11)*norm])

 #sos
 sos_wide = sig.butter(10, wide, btype='bandpass', output='sos')

 #access master IIR filter with coefficients
 master_wide = iir.IIR_filter(sos_wide)

 ################################################

 while running:
     # loop as fast as we can to empty the kernel buffer
     while c.hasSampleAvailable():
         sample = c.getSampleFromBuffer()
         v1 = 10**2*sample[channel_of_window1]
         #filter data
         v2 = master_wide.dofilter(v1)
         #detect strength of peak
         detect=((th*m*10*abs(v2)))
         print(detect)
         #if in range
         if detect > n:
             #digital outputs
             #3 = red light
             #2 = blue light
             c.digital_out(3,1)
             c.digital_out(2,0)

         #if it is close
         elif n > detect > 2:
             c.digital_out(3,0)
             c.digital_out(2,1)

         #if no signal is detected
         else:
             c.digital_out(3,0)
             c.digital_out(2,0)

         #add filtered data to ringbuffer
         ringbuffer= np.append(ringbuffer,v2)
         #plot incoming signal
         qtPanningPlot1.addData(v1)
         #check if there is data in the ringbuffer
         #if data is found then add it to plotbuffer and reset ringbuffer
         if not ringbuffer == []:
             result = ringbuffer
             ringbuffer = []
```

```
            plotbuffer=np.append(plotbuffer,result)

          #only keep the most recent 50 samples of data
          plotbuffer=plotbuffer[-50:]
          #calculate the spectrum
          spectrum = np.fft.rfft(plotbuffer)
          # absolute value
          spectrum2 = m*np.absolute(spectrum)/len(spectrum)
          #plot spectrum
          qtPanningPlot2.addData(spectrum2)

###############################################################################

#main programme
# open comedi
c.open()

#set digital outputs low to start
c.digital_out(2,0)
c.digital_out(3,0)

#print user interaction
print("Violin Tuner")
print("Type note to tune: G, D, A or E")
frequency = input()

#set variables based on user input
if frequency == 'G':
    frequency = 196
    m= 10**107
    th = 1
    n = 12
if frequency == 'D':
    frequency = 293.66
    m= 10**105
    th = 5
    n = 12
if frequency == 'A':
    frequency = 440
    m = 10**116
    th = 7
    n = 12
if frequency == 'E':
    frequency = 659.25
    m= 10**105
    th = 1
    n = 12

print("The frequency of that note is {}Hz" .format(frequency))

##############################################
```

```
#create two instances of plot windows
qtPanningPlot1 = QtPanningPlot("Input Signal")
qtPanningPlot2 = QtPanningPlot("Output Frequency")

#create a thread which gets the data from the USB-DUX
t = threading.Thread(target=getDataThread,args=(qtPanningPlot1,
qtPanningPlot2,ringbuffer,frequency, n))

###############################################

# start data acquisition
c.start(8,1345)

# start the thread getting the data
t.start()

# showing all the windows
app.exec_()

# no more data from the USB-DUX
c.stop()

# Signal the Thread to stop
running = False

# Waiting for the thread to stop
t.join()

c.close()

print("finished")
```

iirclass.py

```
import numpy as np

#2nd order IIR filter
class IIR2_filter:
    def __init__(self, sos):
        self.b0= sos[0]
        self.b1= sos[1]
        self.b2= sos[2]
        self.a1= sos[4]
        self.a2= sos[5]
        self.buffer1 = 0
        self.buffer2 = 0

    def dofilter(self,x):
        #accumulator for the IIR part
        input_acc = x
        input_acc = input_acc - (self.a1*self.buffer1)
        input_acc = input_acc - (self.a2*self.buffer2)

        #accumulator for the FIR part
        output_acc = input_acc*self.b0
        output_acc = output_acc + (self.b1*self.buffer1)
        output_acc = output_acc + (self.b2*self.buffer2)

        self.buffer2 = self.buffer1
        self.buffer1 = input_acc

        return output_acc


class IIR_filter:
    def __init__(self, SOS):
        self.sos = SOS
        self.slaves = []
        self.data = np.zeros(500)
        #access IIR2 filter with coefficients sos
        self.order = len(SOS)
        for i in range(self.order):
            self.slaves.append(IIR2_filter(SOS[0,:]))

    def dofilter(self, x):
        y = x
        #create n instances of slave class
        for i in range(self.order):
            y = self.slaves[i].dofilter(y)
        z = y
        return z
```