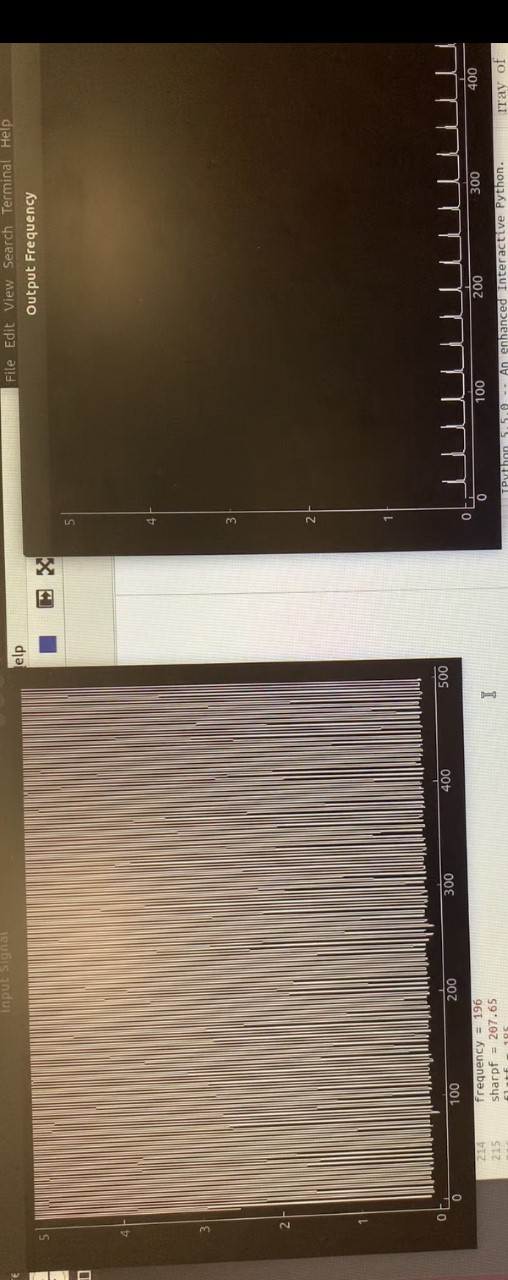
**Assignment 3: IIR Filters**

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'd 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.

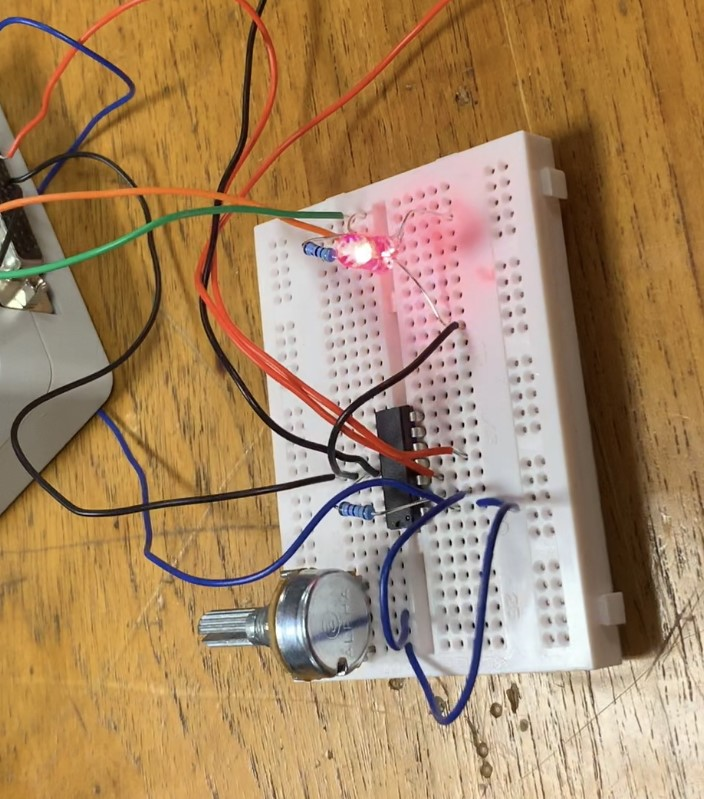


**Figure 1- Realtime Input and Output Signal Plots**

It was also decided to incorporate a tricolor LED into the design to help with tuning. As the user adjusts the tuning of their instrument, the LED changes colour to help the musician make adjustments. For example, if the A string resonates exactly in tune at 440Hz, the tri-colour LED will illuminate in red, indicating that the pitch is spot on. If the pitch isn't quite so perfect, the LED will illuminate in blue, and if the there's crossover between the perfect note and near ideal, the LED will light up in magenta.

Analogue Circuit

*2. Design a simple analogue circuit for your measurement. This could be as simple as two/three components on breadboard or more complex with an instrumentation / operational ampliﬁer. Generally the aim is to be simple but eﬀective. Great ideas can also work with very little hardware. Every team has a budget of max 5 GBP and can order via the electronics store on level 7 (see link on moodle) [20%]*



**Figure 2- Analogue Circuit**

We're using a microphone to convert the audio signal produced into an electrical one, which we then process. To improve its resolution, we used a USB-DUX's 5V output and built an analogue circuit to boost the microphone's signal. This amplified signal is then fed into Python for processing, via the USB-DUX.

The USB-DUX is also responsible for driving the tri-colour LED, with one of its output pins controlling the red leg, and another controlling the blue.

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 normalized 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.

A close up of a logo

Description automatically generated

**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*

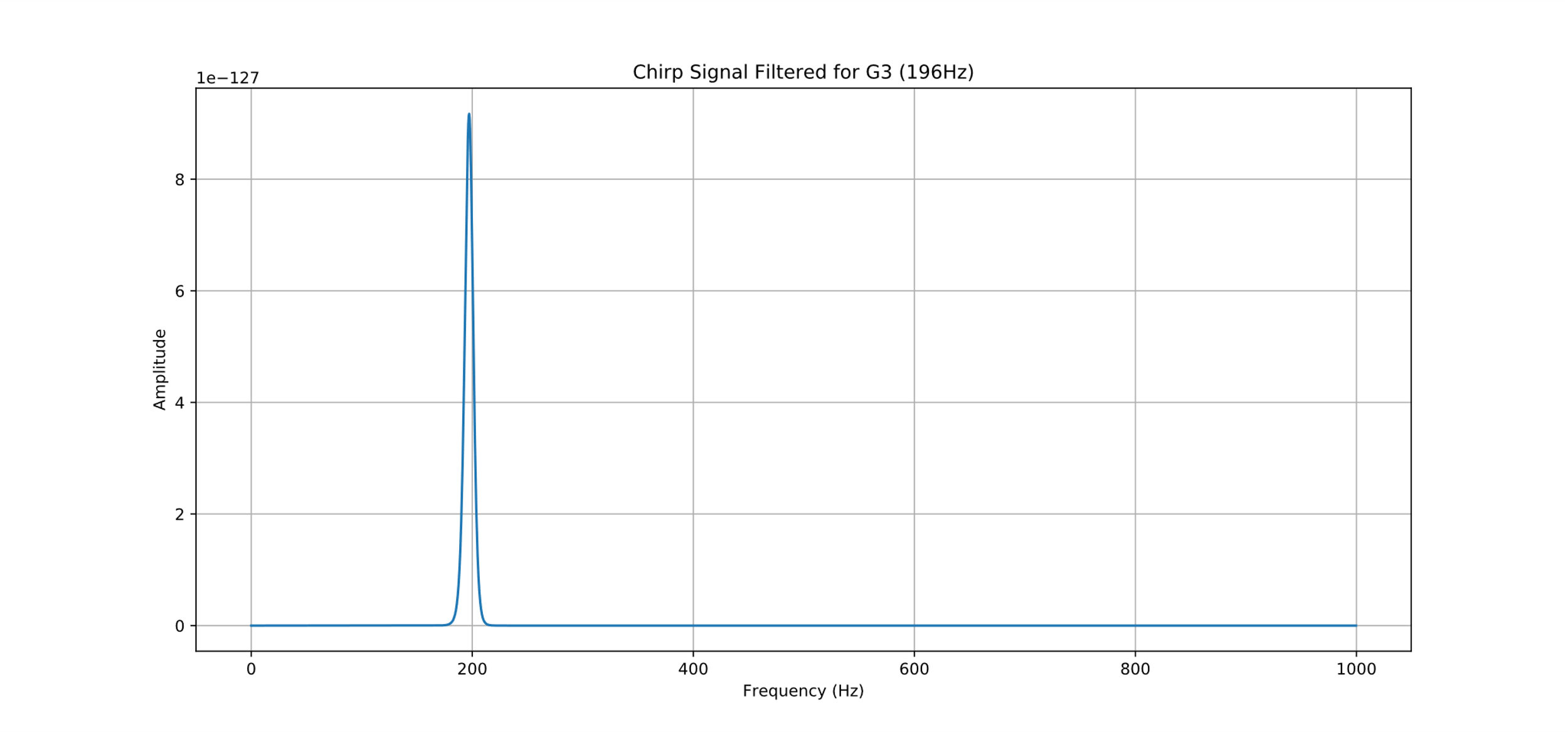
*return output\_acc*

A close up of a logo

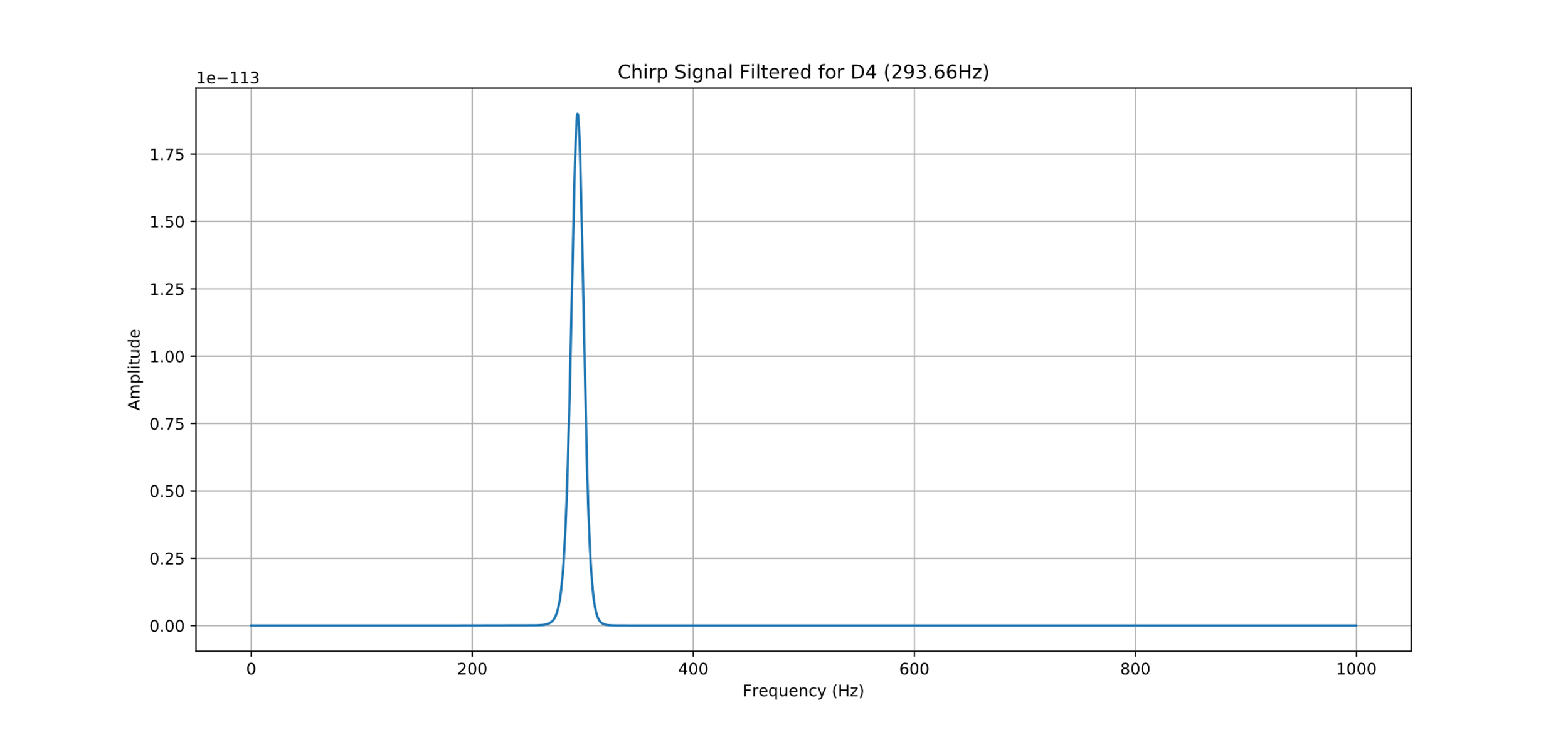
Description automatically generated

**Figure 6- Flow diagram of 2nd order IIR Filter**

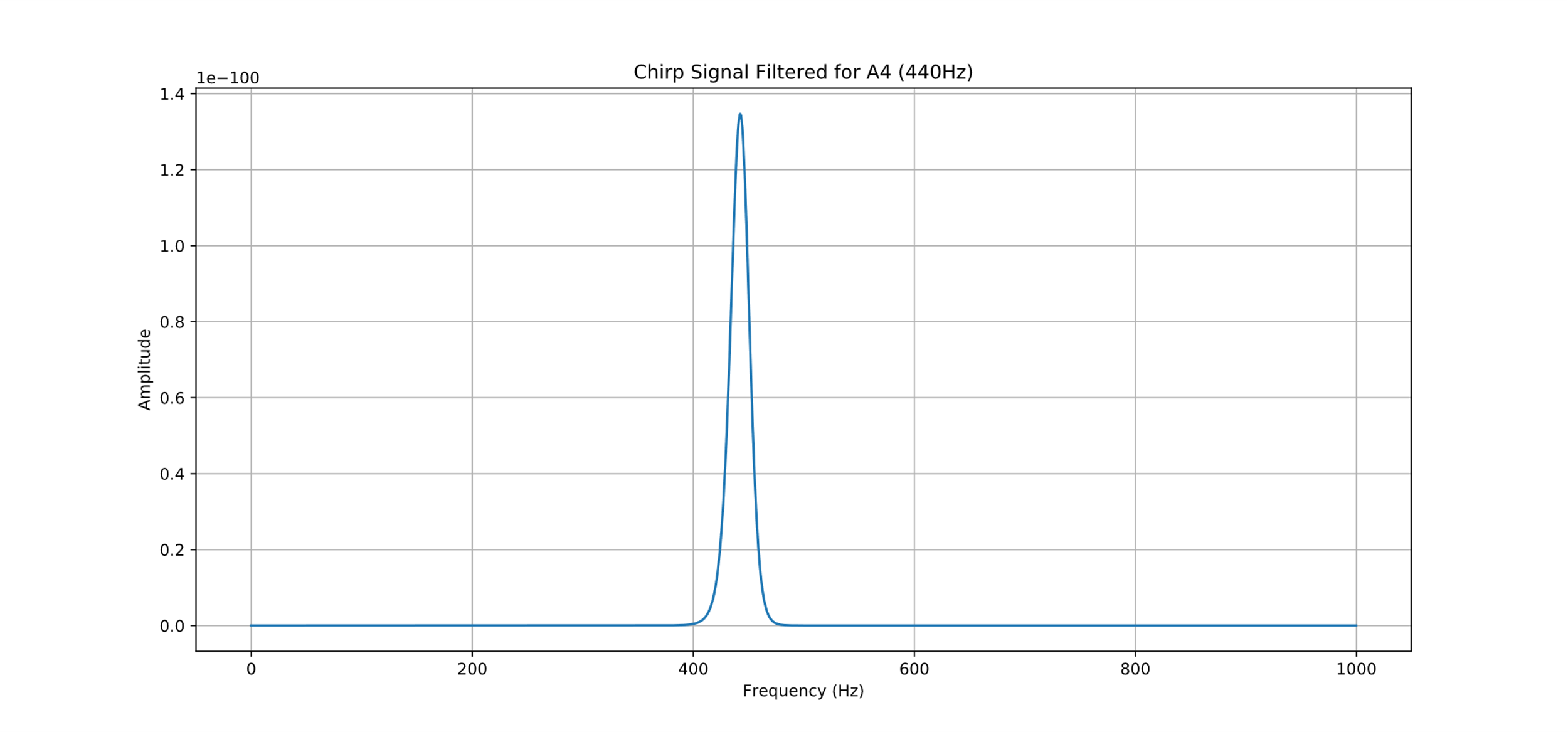
After the filter was created, it was tested for all four frequencies to check that the correct range was being passed. A chirp signal was used as an input. The filter responses are plotted below.



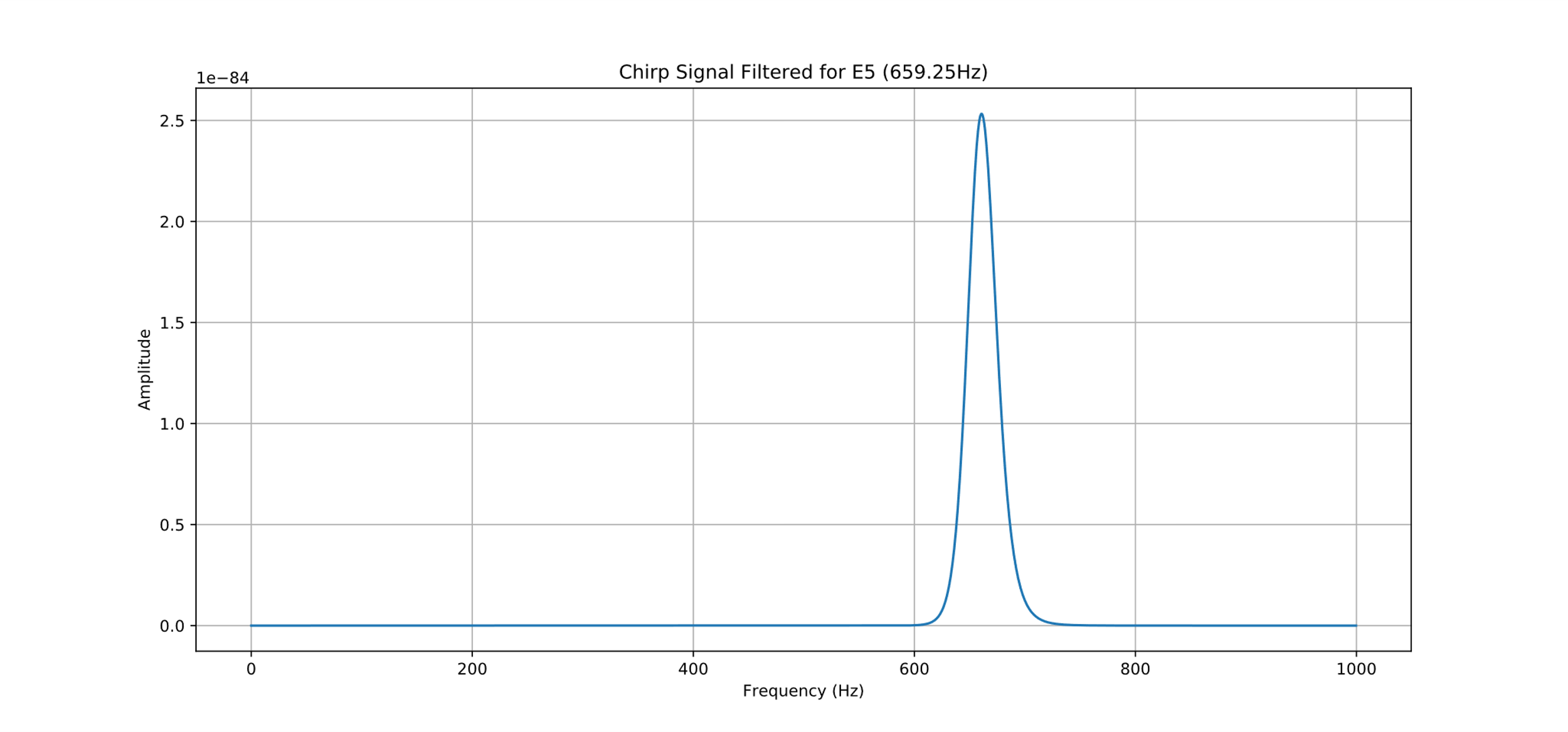
**Figure 7- Response of G Bandpass filter**



**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.

A close up of a logo

Description automatically generated

**Figure 11- Main Programme**

A close up of a logo

Description automatically generated

**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 little 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 program 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 might be wanted for tuning.

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 real time 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

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