DSP Assignment 3, Digital Signal Processing: IIR filters

Arric Hamilton (2310737H) & Ishan Patel (2298968P)

Aims & Objectives:

- Design and create two IIR filter classes in python which takes data sampled from an Arduino and filters it in real time to solve a real-world problem.
- Investigate the sampling functionality of the Arduino and explain any discrepancies caused due to sampling.
- Determine the coefficients for the filter analytically or with high level functions and implement into the IIR classes which take the coefficients in the constructor.
- Compare the results from the IIR filter to the original data and present the filtered in a meaningful way.

Real-time Measurement Problem

A "problem" requiring real time measurement is identified as the automation of the popular procrastination tool known as "Chrome Dinosaur Game", developed by Google as part of their Chrome browser's graphical user interface for when users are unable to connect to the internet. Without internet users can instead play a variation of the classical dino run game.

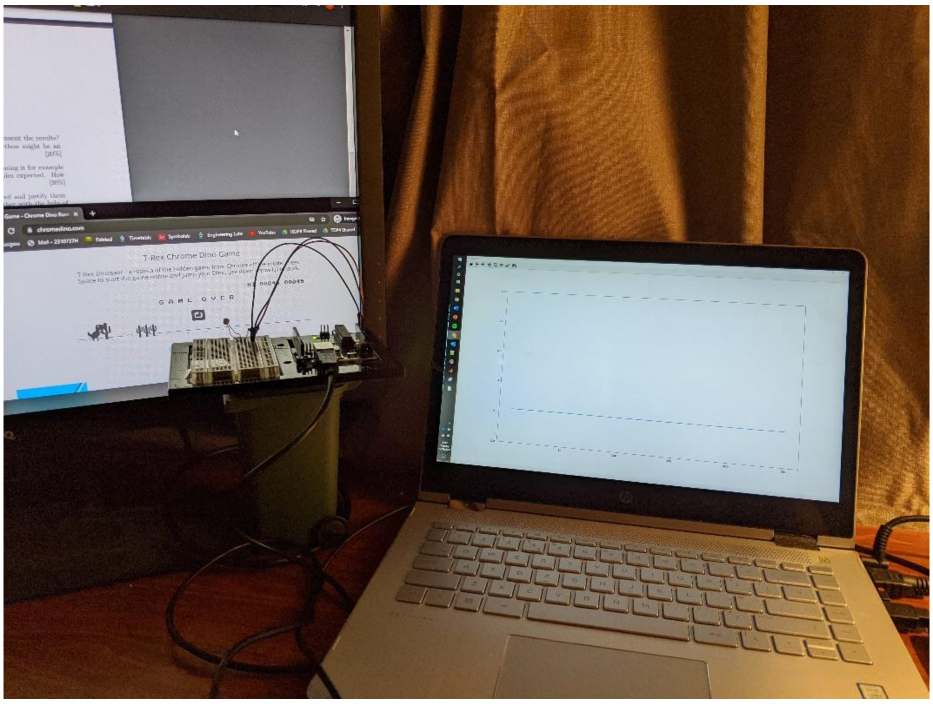
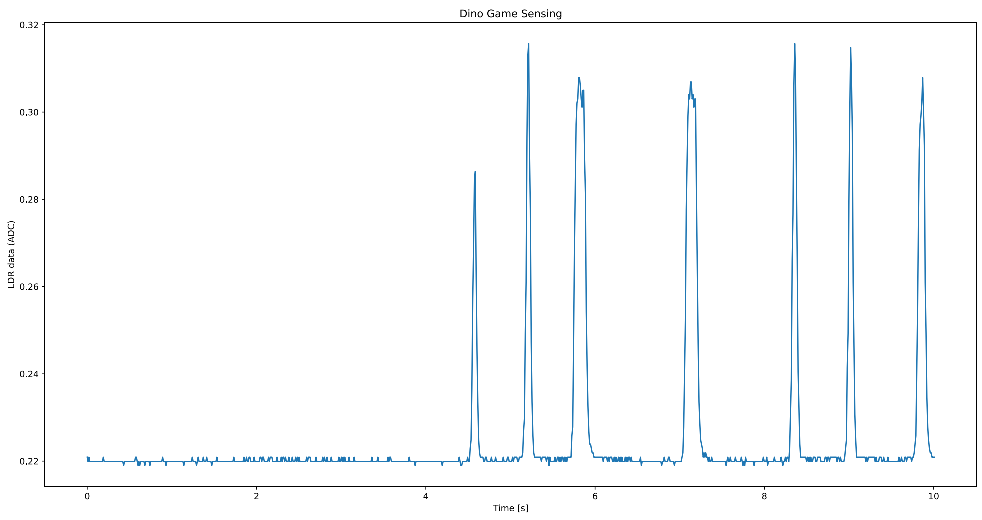


Figure 1 – Photo of setup for real time measurement

The idea behind the problem is to be able to detect when an obstacle appears which the user will have to jump over. This can be realised using an LDR (Photoresistor) attached to an Arduino. The sampling rate of the Arduino is set as 100Hz, the max available using the pyFirmata2 sketch uploaded to the device. However, it should be noted that in a worst-case scenario the LDR has a rise time of ~50ms which means the max sample rate would be around 20Hz. In order to set the correct sampling rate, the components in the frequency spectrum for the LDR data needs to be analysed. A random sample of the game (from restart) is taken for 10 seconds at 100Hz sampling.



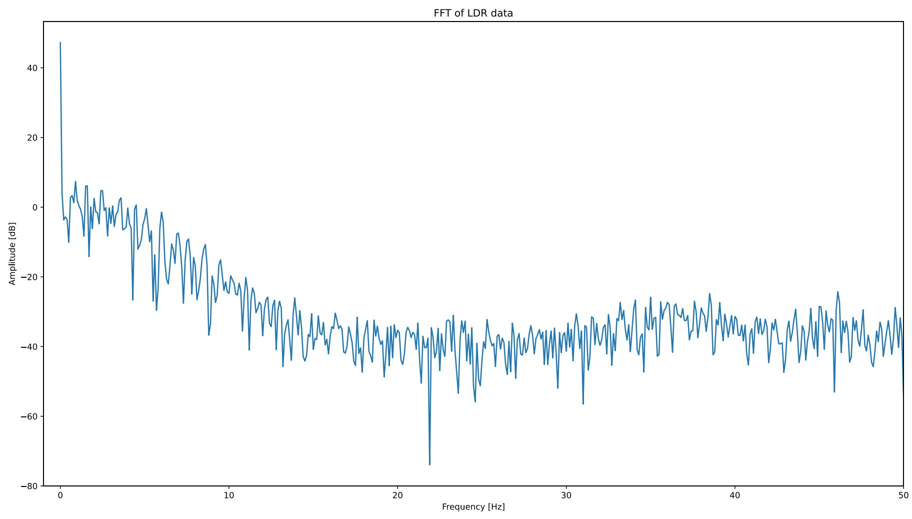


Figure 2 – Time domain representation (top) and FFT frequency spectrum (bottom) up to fs/2

From the above data, it is evident that there is a significant DC component with the rest of the data containing frequencies below 10Hz. This identifies two issues:

- DC Removal
- Noise removal above 10Hz

A large amount of noise can be introduced to the system by accidently bumping the power cable, as the Arduino is powered from USB which has a 5% tolerance on its powerline –

something that will be discussed later. This can be seen by small spikes in the ADC measurement in figure 2.

There exists very little noise above 10Hz and therefore, in order to eliminate any future sources of noise, the sampling rate can be set as 20Hz (10Hz signals can be sampled accurately as per Nyquist theorem which states that the bandwidth of a signal B<Fs/2). The result is a frequency spectrum which has some USB powerline noise removed – still far less than ideal.

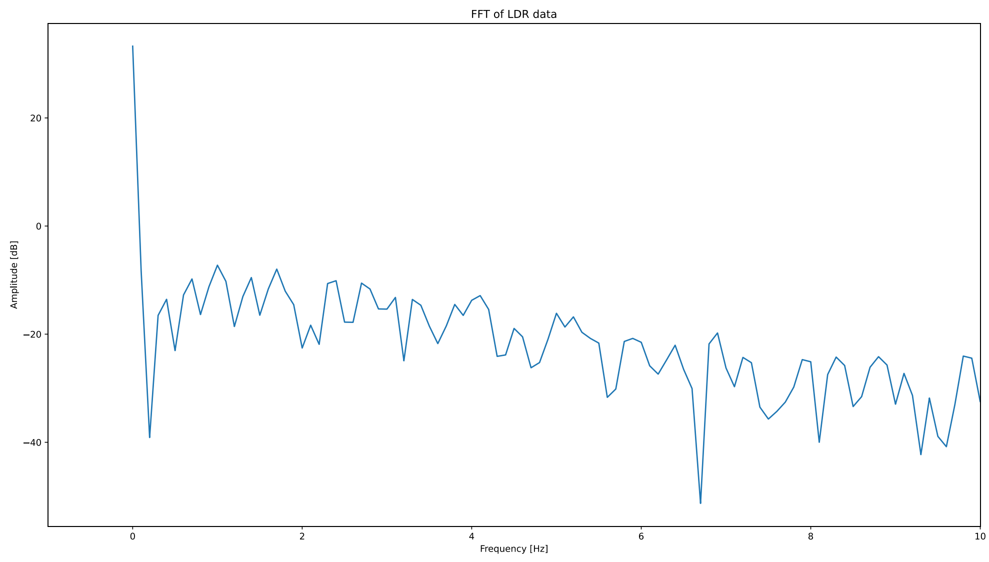
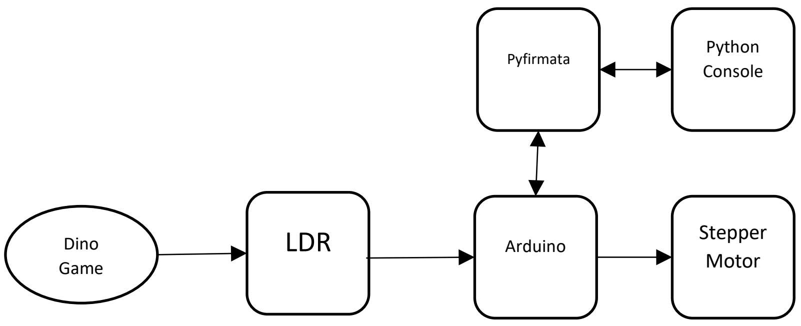


Figure 3 – FFT of ADC data with 20Hz sampling frequency

The overall goal of the experiment is to be able to detect when an obstacle appears at increasing frequency and eventually be able to trigger a stepper motor to press the spacebar to jump. The stepper motor solution is perhaps more ideal however due to budget constraints this is not achievable and a spacebar solution is used. The game only ends after 17 million years of playtime (in reference to the extinction of the T-Rex) which could unfortunately not be simulated either. Further info on Chrome Dino is available here.

The generic block diagram of the system is:



The exact wiring up of the Arduino and LDR is shown in figure 4 and is also viewable in the YouTube video of experimental setup: https://youtu.be/INpnwhm0LSk

Note: R1=10kΩ

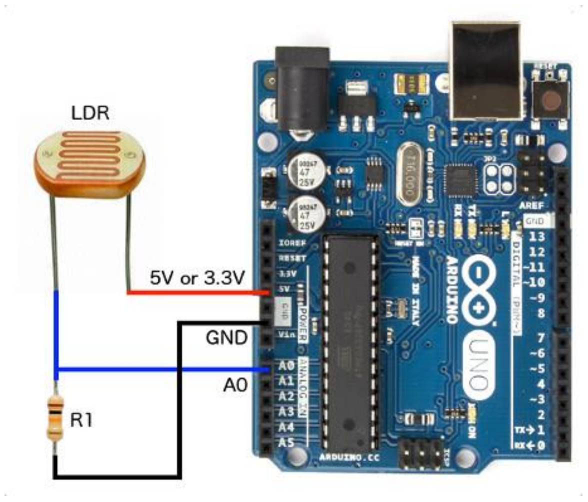


Figure 4 – LDR and Arduino Schematic

Clock Jitter

To check for jitter in sampling, a basic analog input printer is run for 10 seconds at 100Hz. The expected number of samples is thus 1000 however the program counter tallies the total samples at a different number almost every time, with the result typically 1000±2. These results suggest that clock jitter is taking place during the sampling callback function as the ADC runs from a clock divider of the main 16MHz clock on the Arduino. The likelihood is that the jitter is caused from a variety of sources, such as thermal noise, power supply/ground variation or even noise coupled from other parts of the device.

The analog printer code is available in Appendix A and is adapted from Bernd Porr's pyFirmata2 example scripts. The code sets the main part of the program to sleep so that the interference from the CPU operation is minimized.

The Arduino is a fairly basic "beginner" microcontroller and will suffer from ground bounce no matter what due to the unoptimized layout – its not intended for super accurate analog applications. Another valid point in relation to power line noise is the fact that the board is powered from USB 5V (VBUS). The host device is USB specification (in this case a laptop) has, by requirement, a 5% tolerance meaning that the voltage supplied to the Arduino could be as low as 4.75V or as high as 5.25V, a tolerance that is not helpful for powering a board that is trying to perform accurate measurements.

One way to measure or check how much the ADC sampling port suffers from clock jitter for the expected number of samples is by determining the error from a standard sine wave input. Therefore, the change in voltage with respect to sample time jitter (dt) is:

$$\frac{dV}{dt} = \omega V_0 \cos(\omega t)$$

Assuming a worst-case scenario of $\cos(\omega t) = 1$ then the result is as below:

$$\frac{dV}{V_0} = 2\pi f dt \approx 6.28\frac{dt}{T}$$

It is evident that small changes in the voltage are proportional to the frequency multiplied by amount of jitter (given by dt).

Unfortunately, the Arduino does not sport a DAC to produce a sine wave input to measure this. It could hypothetically be performed using a PWM square wave however the Arduino timer registers are set to default values and modifying these to set PWM frequencies would also affect timing of other parts of the device. An example of how it would be carried is shown:

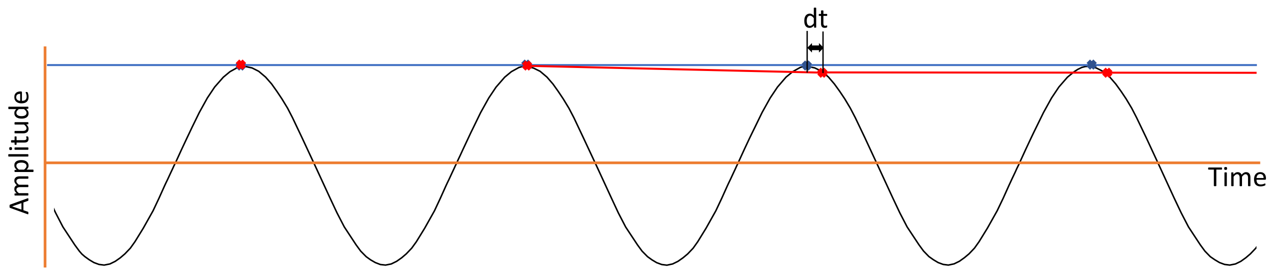


Figure 5 – Sine Wave (black) sampled at set sampling rate to produce a DC waveform (blue) and with jitter (red)

The blue line shows how the signal would appear if sampled at the same rate as the input waveform. However, clock jitter may result in the signal being sampled at a small time delay (dt) from the anticipated timestamp, appearing still as DC but slightly smaller in amplitude.

Coefficients

It was initially thought a bandpass filter for the region 0.1-10Hz would be the optimal solution, however as seen in figure 3, there exists very little noise above 10Hz and therefore a high pass filter with a cutoff frequency of 0.1Hz would suffice to remove the DC aspect of the waveform produced and leave the rest of the spectrum unaffected so that the increasing frequency of obstacles in the game can be left open (although it will never exceed 10Hz). Using the following high-level Butterworth filter command, the coefficients are plotted. Butterworth is chosen over Chebyshev due to its monotonic frequency response.

```
fs=20
f1=0.1/fs #normalized
b,a=sig.butter(2,f1*2,btype='high')
```

Figure 6 shows two variations of the high-level command. Orange shows N=2 and green is N=4. It is evident from this that the second order does not significantly damp the frequency components as compared to the 4th order filter. The blue line shows the analytical solution coefficients. The second order coefficients are:

a1=-1.9556	a2= 0.9565	
b0=0.9780	b1= -1.9560	b2=b0=0.9780

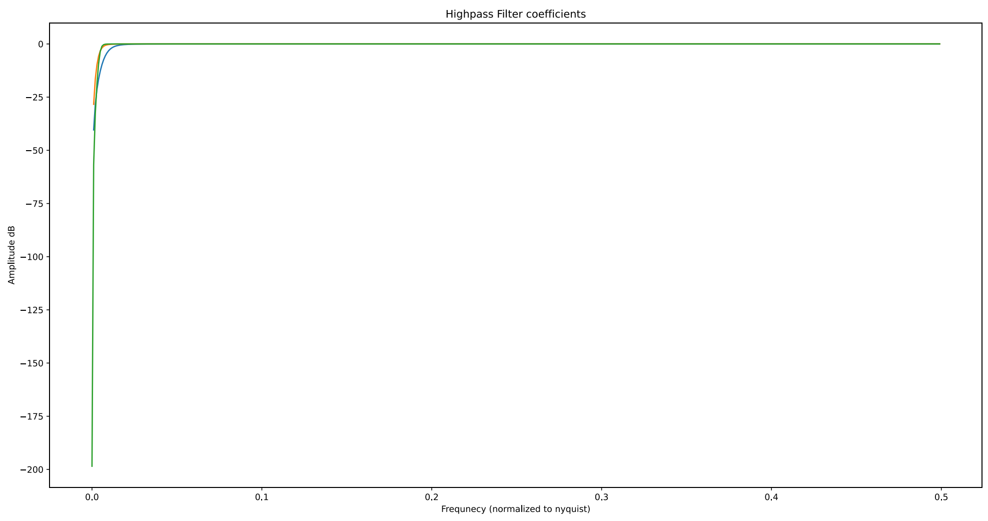
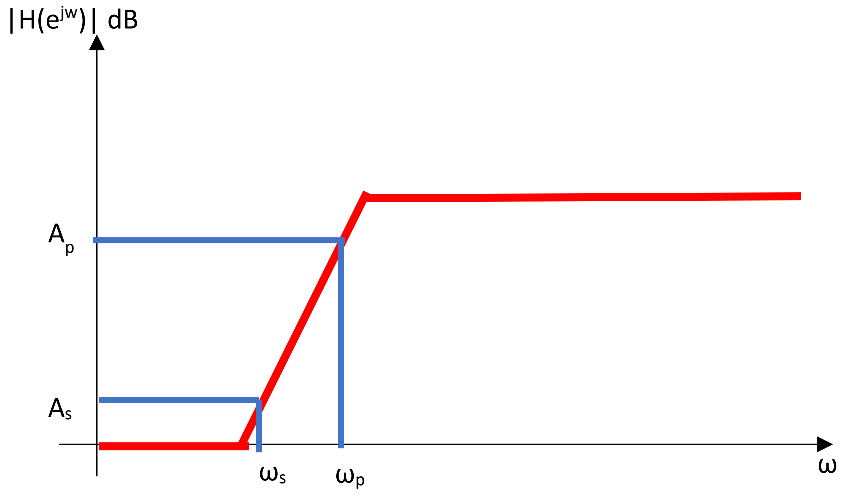


Figure 6 – Highpass response from high level command

As it is trivial to calculate the bilinear transform for orders higher than two, often a chain of 2nd order filters is implemented. In order to verify these coefficients, a second order high pass is calculated analytically.

The first step to design the filter analytically is choose the desired pass and stopband edge frequencies. In this case the filters are normalized in the range -π<ω<π so that the passband edge frequency (ω_p) is $\frac{2*0.1\pi}{f_S} = \frac{\pi}{100}$ and stopband $\omega_S = \frac{2*0.05\pi}{f_S} = \frac{\pi}{200}$. The desired passband attenuation (Ap) is -3dB and stopband attenuation (As) -15dB. This is highlighted below:



The next step is calculating the prewarp (analog cutoff) frequencies which are determined as follows, with 2/T accounting for the effect of the sampling rate ($T=1/f_s$):

$$\Omega = \frac{2}{T}\tan\left(\frac{\omega}{2}\right)$$

Therefore, $\Omega_p = 2.5166$ and $\Omega_p = 1.2571$. In order to design the highpass filter, a low pass filter prototype is used, with the transfer function $\mathrm{H(s)} = \frac{1}{\prod_{k=0}^{N-1}(s-s_k)}$ where N is the order required. It is already decided that N=2 however this can further be validated against the design parameters with the equation:

$$N = \frac{log_{10}\left(\frac{10^{\frac{|Ap|}{10}}-1}{10^{\frac{|As|}{10}}-1}\right)}{2log_{10}(\frac{\Omega_S}{\Omega_p})} = 2.48 \approx 2 \text{ as required.}$$

Thusly $\mathrm{H(s)} = \frac{1}{s^2+\sqrt{2}s+1}$. To convert this to a highpass filter, $\mathrm{s} \to \frac{\Omega_c}{s}$, where Ω_c is the mapped critical frequency. The final result for the highpass filter transfer function is therefore:

$H(s) = \frac{s^2}{s^2+\sqrt{2}\Omega_c s+\Omega_c{}^2}$ which for simplification can become $\frac{s^2}{s^2+Cs+D}$.

This result can be plotted in the complex plane using python. As the poles lie clearly in the left-hand side of the plane, the system is stable. The code for this is available in Appendix E.

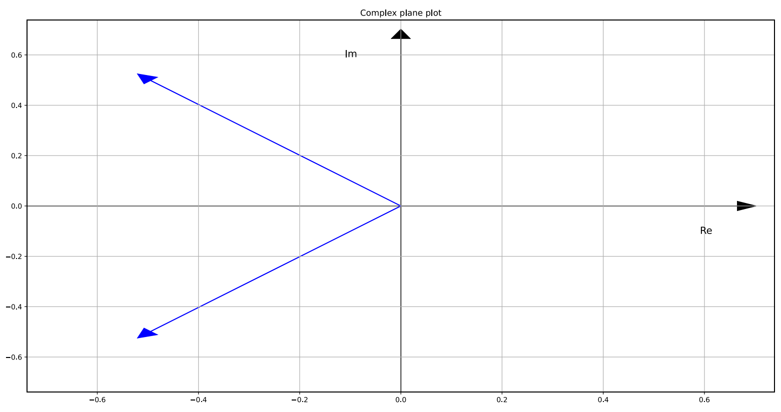


Figure 7 – Complex plane plot of system poles

To perform the bilinear transform, $s \to \frac{1-z^{-1}}{1+z^{-1}}$ which yields the result:

$$H(z) = \frac{1-2z^{-1}+z^{-2}}{(1+C+D)+(2D-2)z^{-1}+(1-C+D)z^{-2}} = \frac{1-2z^{-1}+z^{-2}}{(1+\sqrt{2}\Omega_c+\Omega_c{}^2)+(2\Omega_c{}^2-2)z^{-1}+(1-\sqrt{2}\Omega_c+\Omega_c{}^2)z^{-2}}$$

This is further divided to ensure a0=1 and gives the equation:

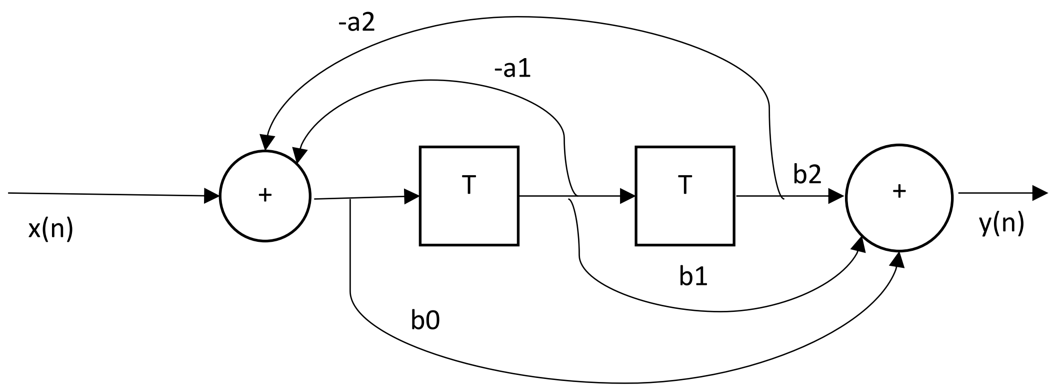
$$\boxed{H(z) = \frac{\frac{1}{1+\sqrt{2}\Omega_c+\Omega_c{}^2} - \frac{2}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}z^{-1} + \frac{1}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}z^{-2}}{1+\frac{2\Omega_c{}^2-2}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}z^{-1}+\frac{1-\sqrt{2}\Omega_c+\Omega_c{}^2}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}z^{-2}}}$$

The following coefficients can be calculated as:

a1=$\frac{2\Omega_c{}^2-2}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}$=-1.9115 a2=$\frac{1-\sqrt{2}\Omega_c+\Omega_c{}^2}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}$= 0.9152

b0=$\frac{1}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}$=0.9567 b1=$\frac{-2}{1+\sqrt{2}\Omega_c+\Omega_c{}^2}$= -1.9134 b2=b0=0.9567

In comparison to the 2nd order coefficients calculated within the signal.butter command, they are fairly similar with better damping characteristics as shown in figure 6, but are not quite as sharp at cutoff. The dataflow diagram for the system is given below:



IIR Filter

The first class in order to design a 2nd order IIR filter takes coefficients in the constructor with no arrays for the buffer to optimize the implementation. Since the direct form I topology is only suitable for integer operations, and the ADC port normalises the input such that $0 \leq V_{A0} \leq 1$, a direct form II topology is used which makes use of two accumulators and a single delay line. This is shown in figure 8.

The class constructor takes in two parameters, f (cutoff frequency- must be normalized) and fs (sampling freq) in order to set the coefficients. The calculations that take place in the code are simply that derived in the coefficient section of this report. The implementation in python is available in appendix B.

In order to perform the IIR operation, the input accumulator deals with the "IIR" coefficients and the output accumulator the FIR coefficients. The order this is performed is not of consideration due the linear nature of the filter. The code for the IIR section of the filter is:

```
        self.in_acc = data
        self.in_acc = self.in_acc - (self.a1*self.buf1)
        self.in_acc = self.in_acc - (self.a2*self.buf2)
```

Similarly, the FIR part is implemented as:

```
        self.out_acc = self.in_acc * self.b0
        self.out_acc = self.out_acc + (self.b1*self.buf1)
        self.out_acc = self.out_acc + (self.b2*self.buf2)
```

After the coefficient multiplication, the rest of the doFilter function simply delays the data to the next buffer e.g. buf2=buf1. The general topology is shown below:

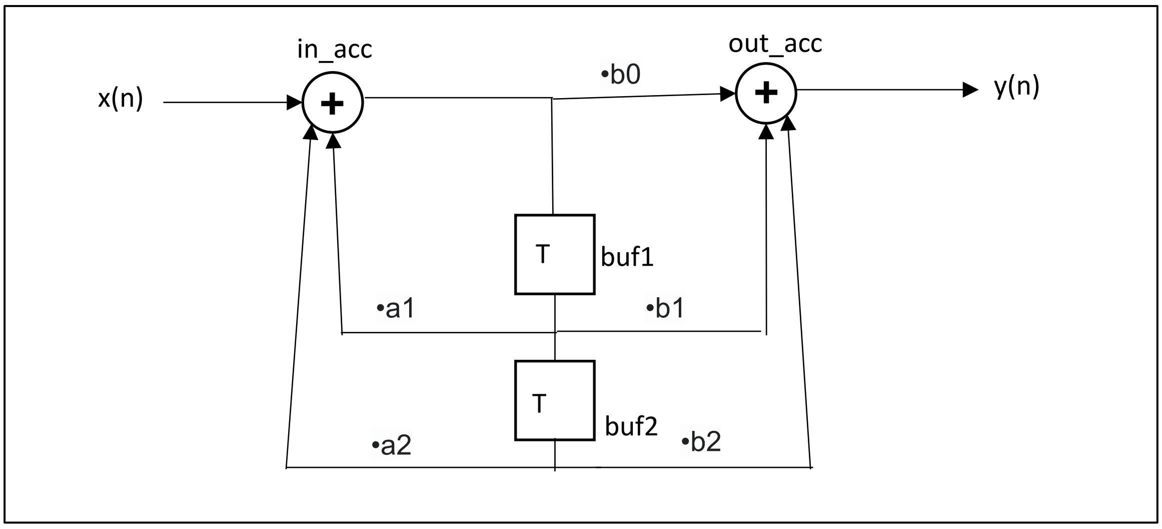
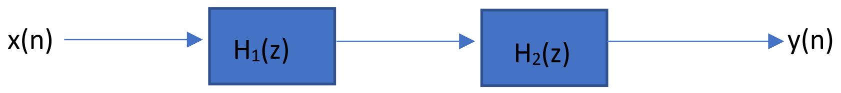


Figure 8 – Dataflow diagram for direct form II topology

IIR Chain

As seen in the previous section, a 4th order Butterworth highpass provides considerably more damping for less than the cutoff frequency. The most straightforward implementation of a higher order filter is chaining 2nd order IIR filters as demonstrated:



This eliminates the need to perform trivial bilinear transforms as each analogue transfer function produces a complex conjugate pole pair. The dataflow diagram for the new system is realised below in figure 9. Note that it is simply two of the same biquad sections together. The coefficients shown are evaluated from the SOS output version of the signal.butter command.

If a 6th order was desired, another 2nd order filter can be chained on, and using the signal.butter command to generate the necessary SOS coefficients.

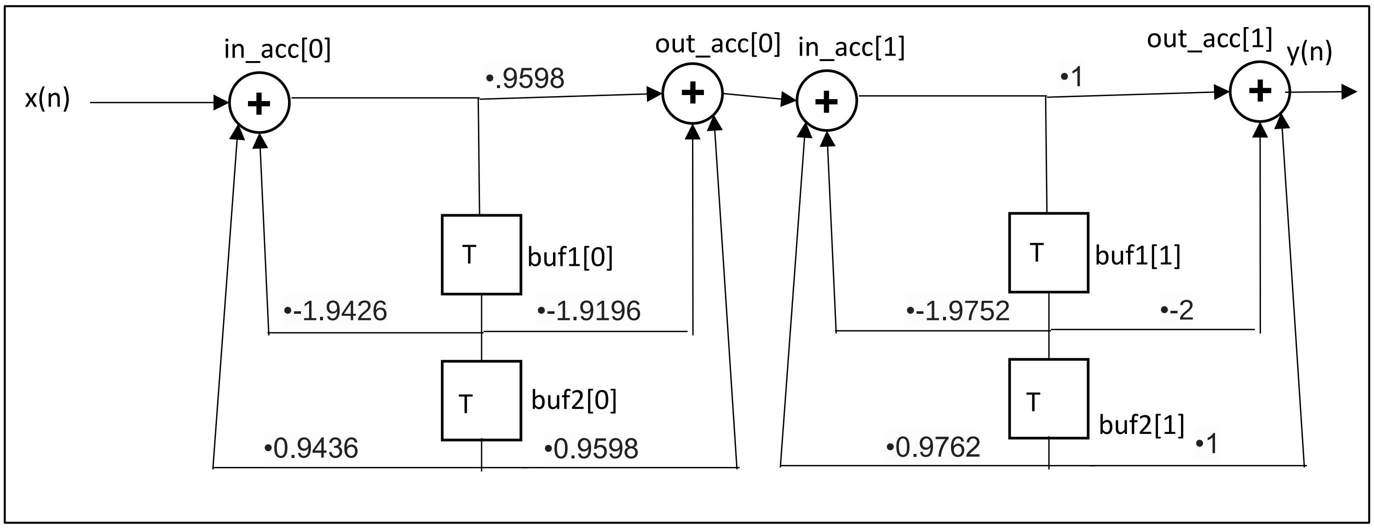


Figure 9 – Chained IIR 2nd Order Filter for direct form II topology

The implementation of this in python is available in Appendix B. The operation is as follows:

```
for i in range(len(self.chain)):
            data=self.chain[i].doFilter(data)  #return  data  for  next  input
            …
```

The for loop checks for how many iterations to run through according to the number of chained filters required (which is set based on the order of SOS coeffs). The code then runs through the usual IIR filter and returns the output to chain it to the input of the next one. The SOS output of the butter command will generate an array of size [2][6] for a fourth order filter, indicated by the first index of the array – e.g. a 6th order would be of size [3][6]. Setting the for loop to run for 2 cycles requires the use of 4 buffers and 4 accumulators.

Results

The first result to analyse is the II2Filter using the coefficients calculated earlier. The code utilized to perform the following tests is included in appendix C. It is heavily based on the AnalogPrinter demo created by Bernd Porr and modifies the print callback to capture data and find peaks as well as including the number of samples – previously used in clock jitter testing.

The other major inclusion is a new coeffHP function which, as the name suggests, determines the highpass coefficients to be fed to the IIR2Filter when it is not used as a chain. The function is essentially a python representation of the equation H(z) for a highpass filter. It takes in f (normalized cutoff) and fs (sampling freq) to calculate the prewarped frequencies which is further used to generate an array of the coefficients as required.

An instance of IIR2Filter is created with these coeffs and the main program starts the data acquisition for a period of time in which the program "sleeps" – in this case 30 seconds. The final step of the main program involves resizing the captured data to include only the data that was acquired, as it is initially set to a generic predefined size.

The results from the IIR2Filter using the calculated coefficients are as follows:

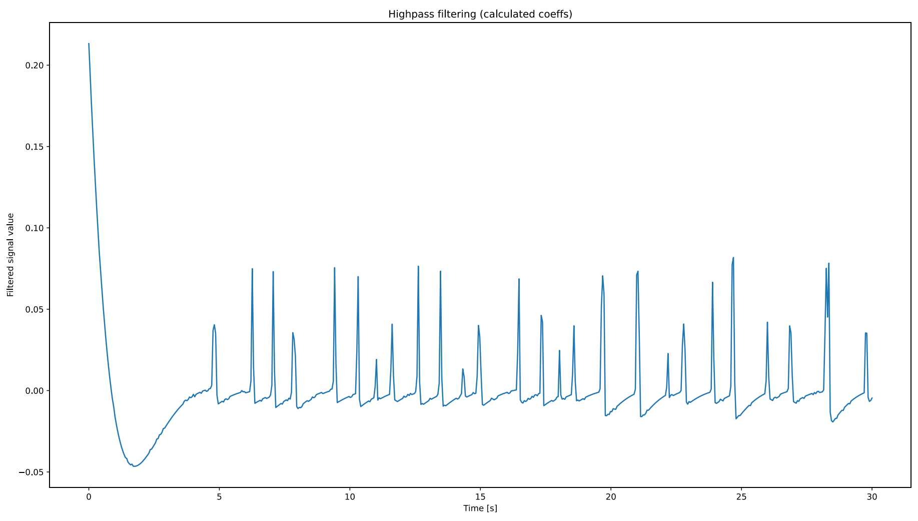


Figure 10 – 2nd order Highpass filtering of chrome dino obstacle detection

As evident in figure 10, the DC component of the signal has clearly been removed. The resulting output is very clean (albeit with very small ripples) after the initial DC normalisation before 5seconds. Fortunately, when the game starts there is a 5 second "grace period" where no obstacles occur. This provides ample opportunity for the IIR filter to get to work and remove the DC before the resulting obstacles start impacting the signal. There is a clear correlation between the size of the obstacle, with smaller cacti generating the weaker pulses such as the one at 5s, and larger cacti generating greater pulses e.g. 24s.

Figure 11 shows the FFT spectrum where it is evident there is a small degree of noise introduced to the signal.

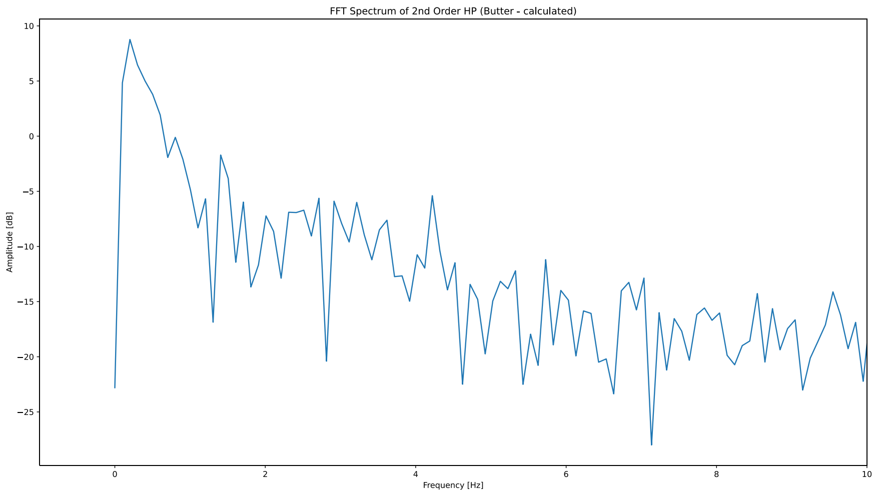
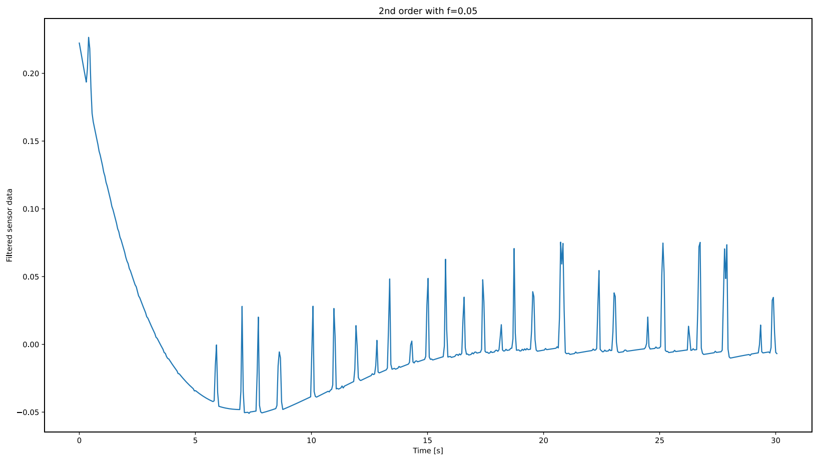


Figure 11 – FFT spectrum of 2nd order calculated butter IIR filtering

Such noise is undesirable however it is less critical than the time taken to remove DC. After experimenting with the cutoff frequency and trying 0.05Hz for DC removal, as the dino game evidently has very low frequency components, the time taken in this outweighs the effect of the time. A heavier damped spectrum is produced but at the cost of time. It was decided that 0.1Hz would be kept.



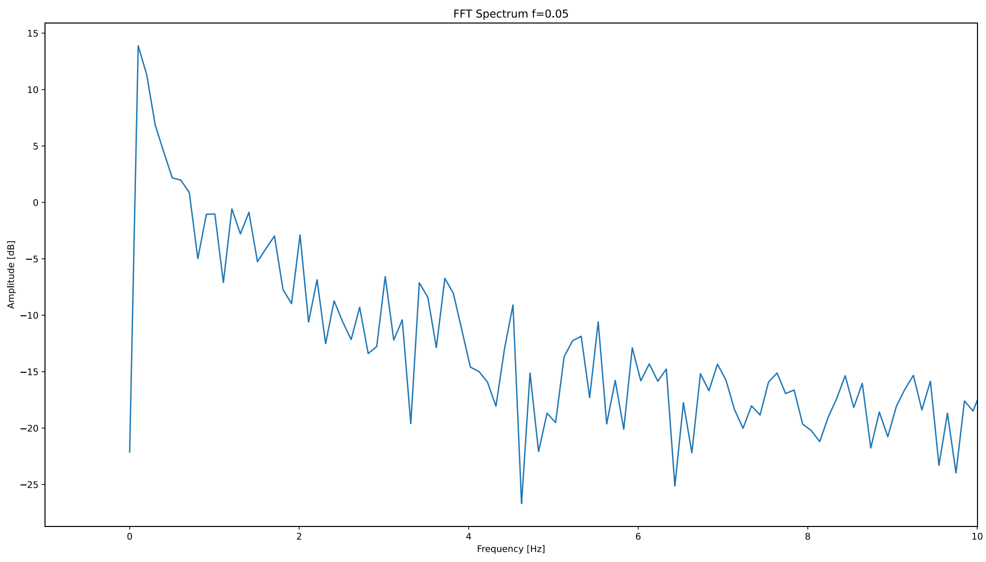


Figure 12 – Time domain (top) and FFT spectrum (bottom) of 2nd order with f=0.05

There is a few pulses with amplitude in between the smaller and larger pulses, and this is due to the obstacle being a collection of the cacti, as shown in figure 13. The actual size of the pulse in terms of obstacle detection does not play a significant part in determining the required response as all pulses require the same space bar press to jump. The only obstacle that does not require a jump is the high-flying pterodactyl – this is due to its location on the screen away from the sensor as it is not (nor needed to be) detected.

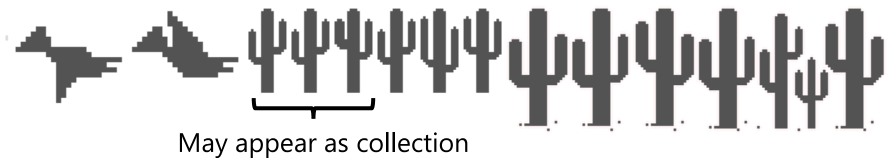


Figure 13 – Obstacles faced in Chrome Dino run game

As mentioned beforehand, unfortunately there is no inclusion of a stepper motor. However, the time at which the spacebar needs to be pressed to jump is captured using the code in the call back function:

```
if (output>0.01 and self.timestamp-self.oldt>0.25 and self.timestamp>5):
#peak detection
            self.jump+=1
            wsh.SendKeys(" ") #OR include write to stepper motor here!
            self.oldt=self.timestamp
```

The detection function as highlighted above could write to a stepper motor on the Arduino outputs. In the current implementation, the output is checked against a threshold (0.01), makes sure the difference between successive peaks is not too close together, and ensures that the data during the initial DC removal period of ~5s does not impact detection. If so, the spacebar is sent using pywin32 module. In figure 10, the self.jump correctly counts 26 peaks.

The next test is using the 2nd and 4th order coefficients are generated by the signal.butter command. The second order result is shown below:

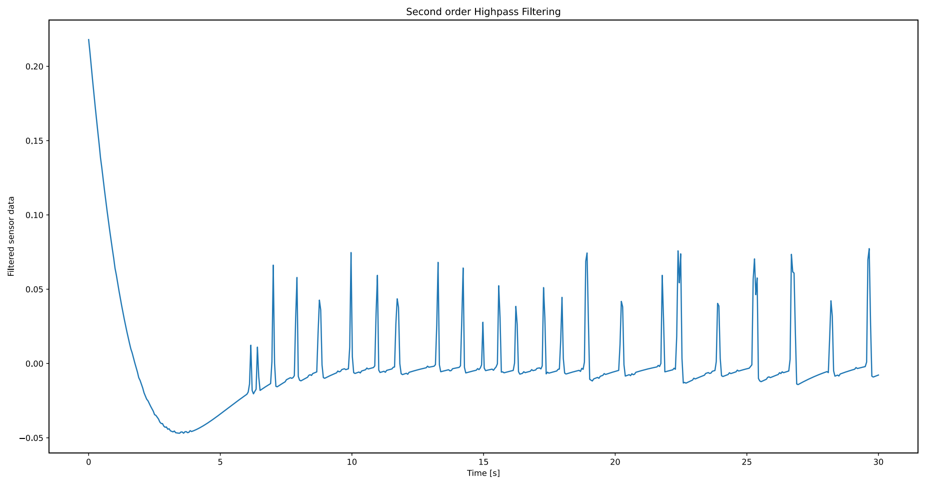


Figure 14 – 2nd order butter command filtering

As expected, the result is similar to that of the calculated coefficients although it suffers from a longer delay at the start before DC is fully removed. This is near problematic as the initial 2 pulses are lower in amplitude than they would be usually, however the threshold is still met and the jump correctly counts 24 peaks. The signal itself is a little smoother as from figure 6, it does not have as harsh damping for frequencies less than cutoff. There is however still a relative degree of noise, but less than the calculated coefficient result, in the FFT spectrum:

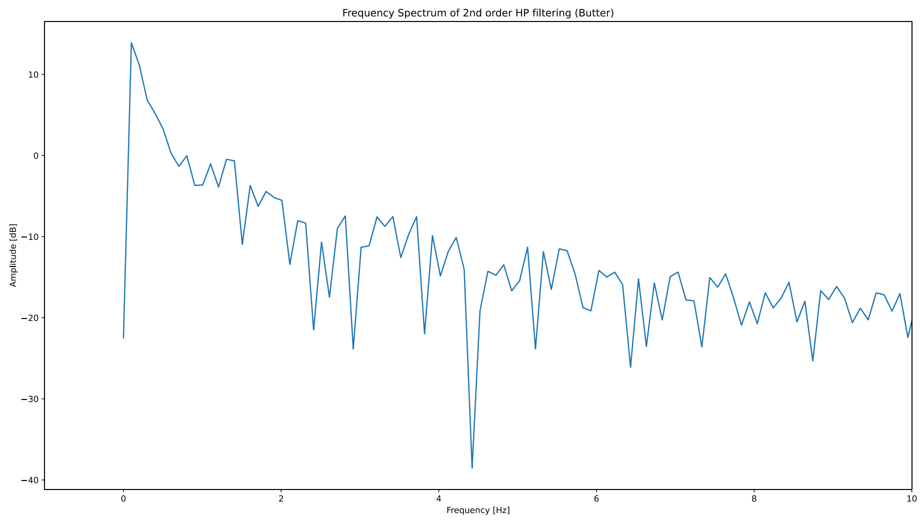


Figure 15 – FFT spectrum of 2nd order butter IIR filter

The 4th order is tested as in figure 16. This is now problematic. The greater damping provided by the 4th order means it takes longer to settle to the zero DC state. Where it was 5 seconds before, it is now realistically more than 10 seconds which is not practical at all for this application. The graph shows 26 peaks, yet the jump storage suggests there is 33 which is evidently wrong and would mess up the timing at the initialisation of the game. The FFT of this is available in figure 17.

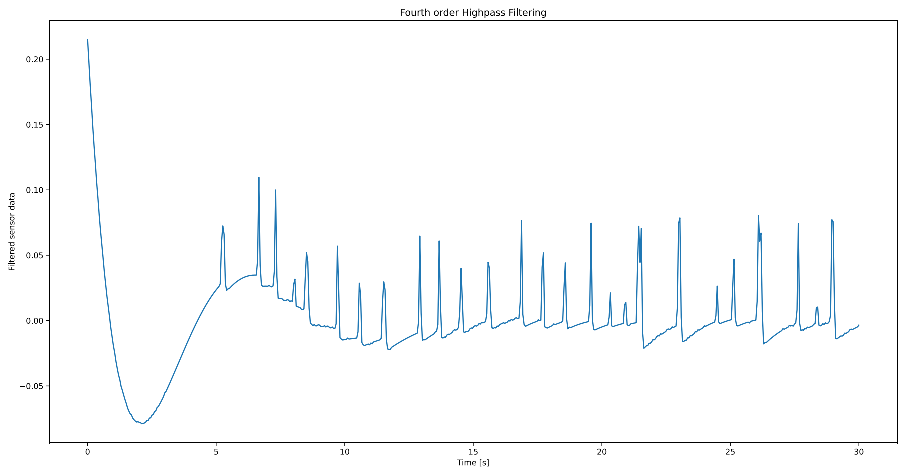


Figure 16 – 4th order butter command filtering

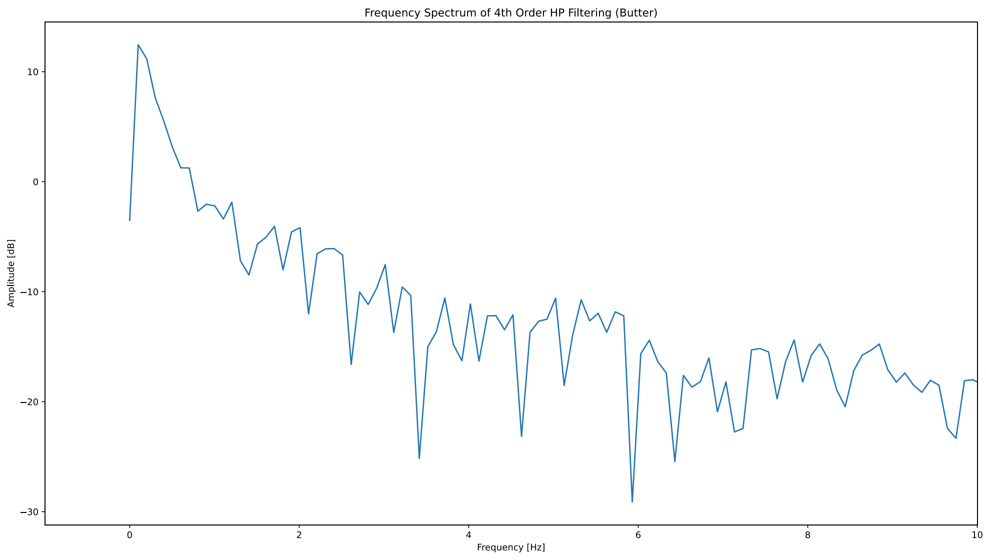


Figure 17– FFT spectrum of 4th order IIR filtering (up to fs/2)

The spectrum has slightly less noise as the overall amplitudes are less, however it is still not suitable as the dino game has low frequency components which cannot be as harshly damped as in the 4th order filter.

The conclusion from these tests is that the 2nd order generated coefficients work best to provide the smallest time delay to reach a true zero DC level as a trade-off against noise.

The real time adaptation of this program is available in appendix D. It is functionally the same as the analogPrinter script but instead incorporates two real time plot to display the data and filtered output as well as includes a “jump” class to detect for when the dinosaur should jump.

The video playlist of this is available at: [YouTube link]

The main use of the program is to trigger the space bar via a stepper motor and as such the use of any particular graph to display the output is unnecessary. The only use a graph would have is to clearly highlight at what time intervals the jump occurs. This is demonstrated by the following code addition to the analogPrinter code (and jump is changed to an array):

```
self.jump[self.i]=1
…
time = np.linspace(0., len(analogPrinter.data)/fs, len(analogPrinter.data))
analogPrinter.jump=np.resize(analogPrinter.jump,len(time))
plt.plot(time,analogPrinter.jump)
plt.title('Jumps against time')
plt.ylabel('Jump')
plt.xlabel('Time [s]')
plt.show()
```

This will print the jumps at the according time intervals. Figure 18 shows the precise nature of this action. Note that the orange trace shows the filtered data:

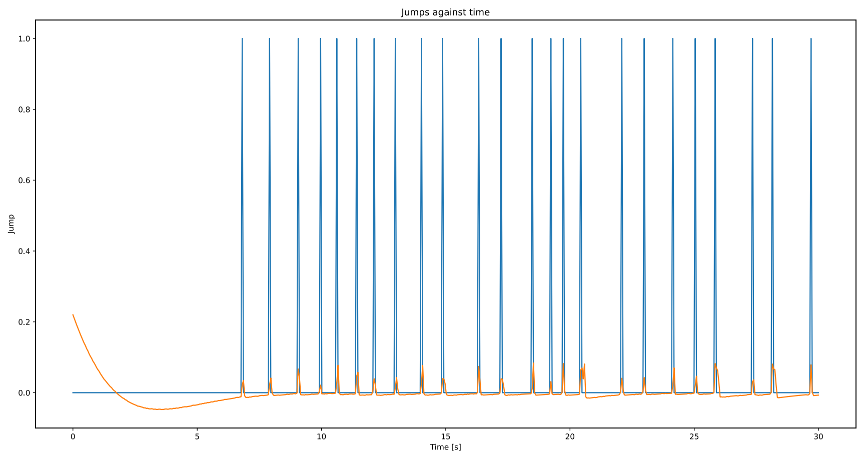


Figure 18 – Jumps occurring in relation to the filtered signal

Conclusion

In conclusion, the DC filtering action of the IIR highpass implemented works well. The DC removal aspect means the code should work in any light environment as any constant light source will be removed. The pure signal after the offset is removed will remain the same thus allowing the threshold set in the code to remain constant. The best performance for damping is a 4th order filter but the time it takes to reach a zero DC level is of greater importance and thus a 2nd order filter should be used – preferably the calculated highpass Butterworth filter.

Future considerations for this project would be the fact the change in light intensity (whilst unaffected by constant daylight/lightbulbs etc) is given by a monitor – the brightness of which will vary depending on the manufacturer and model. In this use case, the model is a BenQ RL2455 which sports a brightness of 250 cd/m^2.

Realistically speaking, the ADC input is accurate enough to correctly determine the peaks albeit with an offset included – the threshold would need a small adjustment. It is perhaps fortunate that the dino game has a 5s delay at the start before obstacles appear as this allows time to remove the DC, otherwise the threshold would encounter some difficulty in this time period.

The program is bound to fail eventually if the frequency increases however the implementation here can be taken as a baseline which would be improved upon if a stepper motor was used instead of the space bar using pywin32 which adds additional overheads.

Appendix A

```python
from pyfirmata2 import Arduino
import time

PORT = Arduino.AUTODETECT
# PORT = '/dev/ttyACM0'

# It uses a callback operation so that timing is precise and
# the main program can just go to sleep.


class AnalogPrinter:

    def __init__(self):
        self.samplingRate = 100
        self.timestamp = 0
        self.board = Arduino(PORT)
        self.samples=0

    def start(self):
        self.board.analog[0].register_callback(self.myPrintCallback)
        self.board.samplingOn(1000 / self.samplingRate)
        self.board.analog[0].enable_reporting()

    def myPrintCallback(self, data):
        print("%f,%f" % (self.timestamp, data))
        self.timestamp += (1 / self.samplingRate)
        self.samples+=1 #increases no of samples per callback

    def stop(self):
        self.board.samplingOff()
        self.board.exit()

print("Let's print data from Arduino's analogue pins for 10secs.")

# Let's create an instance
analogPrinter = AnalogPrinter()

# and start DAQ
analogPrinter.start()

time.sleep(10)

# let's stop it
analogPrinter.stop()

print("finished with n samples:")
print(analogPrinter.samples)
```

Appendix B

```
# -*- coding: utf-8 -*-
"""
Created on Thurs Nov 5 10:35:22 2020

@author: Arric Hamilton & Ishan Patel
"""
class IIR2Filter:
    def __init__(self,_coeffs):
        """
        Sets coeffs for the 2nd order filter

        Parameters
        ----------
        coeffs : [0:3] must be FIR coeffs and [3:6] IIR coeffs
        """
        #sets coeffs
                self.a0,self.a1,self.a2=_coeffs[3:6]
        self.b0,self.b1,self.b2=_coeffs[0:3]

        self.in_acc = 0
        self.out_acc = 0
        self.buf1 = 0
        self.buf2= 0
        self.output = 0

    def doFilter(self, data):
        #IIR part
        self.in_acc = data
        self.in_acc = self.in_acc - (self.a1*self.buf1)
        self.in_acc = self.in_acc - (self.a2*self.buf2)

        #FIR part
        self.out_acc = self.in_acc * self.b0
        self.out_acc = self.out_acc + (self.b1*self.buf1)
        self.out_acc = self.out_acc + (self.b2*self.buf2)

        self.buf2=self.buf1 #shifts buffers along
        self.buf1=self.in_acc

        self.output=self.out_acc
        return self.output

class IIRFilter:
    def __init__(self,_coeffs):
        #sets no. of filters required
        self.chain=[]
        #sets the sos array coeffs to correct filter
        for i in range(len(_coeffs)):
            self.chain.append(IIR2Filter(_coeffs[i]))

    def doFilter(self, data):
        for i in range(len(self.chain)):
            data=self.chain[i].doFilter(data) #return data output for next
input
        return data
```

Appendix C

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 18 11:38:44 2020

@author: Arric Hamilton (2310737H) & Ishan Patel (2298968P)
The original pyFirmata was written by Tino de Bruijn. The realtime sampling
/ callback has been added by Bernd Porr.
"""
from pyfirmata2 import Arduino
import time
import numpy as np
PORT = Arduino.AUTODETECT
import iir_filter as IIR
import scipy.signal as sig
import matplotlib.pyplot as plt
import win32com.client as comclt
wsh= comclt.Dispatch("WScript.Shell")
wsh.AppActivate("Chrome") # select another application

# It uses a callback operation so that timing is precise and
# the main program can just go to sleep.

class AnalogPrinter:
    def __init__(self):
        # sampling rate: 20Hz
        self.samplingRate = 20
        self.timestamp = 0
        self.board = Arduino(PORT)
        self.samples=0
        self.i=0
        self.jump=np.zeros(1000)
        self.oldt=0
        self.data=np.zeros(1000)

    def start(self):
        self.board.analog[0].register_callback(self.myPrintCallback)
#callback everytime data captured
        self.board.samplingOn(1000 / self.samplingRate)
        self.board.analog[0].enable_reporting()

    def myPrintCallback(self, data):
        #prints sampled data and stores filtered samples
        output=yn.doFilter(data) #IIR Filter
        print("%f,%f" % (self.timestamp, data))
        self.timestamp += (1 / self.samplingRate)
        self.samples+=1 #incremented every callback
        self.data[self.i]=output #stores data for analysis
        if (output>0.01 and self.timestamp-self.oldt>0.25 and
self.timestamp>5): #peak detection
            self.jump[self.i]=1
            wsh.SendKeys(" ") # send the keys you want
            #we could write to a stepper motor here!
            self.oldt=self.timestamp
        self.i+=1

    def stop(self):
        self.board.samplingOff()
        self.board.exit()
```

```python
def coeffHP(f,fs):
    """
    Calculates the highpass coefficients for the IIR class

    Parameters
    ----------
    f : Highpass cutoff - MUST be normalized to fs
    fs : Sampling frequency

    """
    w=2*np.pi*f/fs #digital frequency scaled to pi
    ohm=2*fs*np.tan(w/2) #prewarped freq
    C=1.414*ohm #sqrt2*prewarped freq
    D=ohm**2 #prewarped freq^2
    coeffs=np.zeros(6)

    #calculates coeffs - see report for derivation
    coeffs[3]=1.0
    coeffs[4]=(2*D-2)/(1+C+D)
    coeffs[5]=(1-C+D)/(1+C+D)
    coeffs[0]=1/(1+C+D)
    coeffs[1]=-2/(1+C+D)
    coeffs[2]=coeffs[0]
    return coeffs

analogPrinter = AnalogPrinter()

fs=analogPrinter.samplingRate
f1=0.1/fs #normalized

#coeffs_chain=sig.butter(4,f1*2,btype='high',output='sos') #chain filter
coeffs
coeffs_2=coeffHP(f1,fs)
#yn=IIR.IIRFilter(coeffs_chain)
yn=IIR.IIR2Filter(coeffs_2)

print("Let's print data from Arduino's analogue pins for 30secs.")
analogPrinter.start() #data acquisition
time.sleep(30) # acquires for 30s
analogPrinter.stop() #stops DAQ and closes SERCOM
analogPrinter.data=np.resize(analogPrinter.data,analogPrinter.samples)
#resizes to include only the captured points
print("finished with n samples:")
print(analogPrinter.samples)

#UNCOMMENT TO PERFORM JUMP COUNTER
"""
time = np.linspace(0., len(analogPrinter.data)/fs, len(analogPrinter.data))
analogPrinter.jump=np.resize(analogPrinter.jump,len(time))
plt.plot(time,analogPrinter.jump)
plt.title('Jumps against time')
plt.ylabel('Jump')
plt.xlabel('Time [s]')
plt.show()
"""
```

Appendix D

```
"""
Created on Wed Nov 18 11:38:44 2020

@author: Arric Hamilton (2310737H) & Ishan Patel (2298968P)
The original pyFirmata was written by Tino de Bruijn. The realtime sampling
/ callback has been added by Bernd Porr.
SEE LICENSE FILE
"""
from pyfirmata2 import Arduino
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import iir_filter as IIR

# Realtime oscilloscope at a sampling rate of 20Hz
# It displays analog channel 0.
PORT = Arduino.AUTODETECT

# Creates a scrolling data display
class RealtimePlotWindow:
    def __init__(self,title,x,y,scale):
        # create a plot window
        self.fig, self.ax = plt.subplots()
        self.ax.set_title(title)
        self.ax.set_xlabel(x)
        self.ax.set_ylabel(y)
        # that's our plotbuffer
        self.plotbuffer = np.zeros(500)
        # create an empty line
        self.line, = self.ax.plot(self.plotbuffer)
        self.ax.set_ylim(scale[0], scale[1])
        # That's our ringbuffer which accumluates the samples
        # It's emptied every time when the plot window below
        # does a repaint
        self.ringbuffer = []
        # start the animation
        self.ani = animation.FuncAnimation(self.fig, self.update,
interval=100)

    # updates the plot
    def update(self, data):
        # add new data to the buffer
        self.plotbuffer = np.append(self.plotbuffer, self.ringbuffer)
        # only keep the 500 newest ones and discard the old ones
        self.plotbuffer = self.plotbuffer[-500:]
        self.ringbuffer = []
        # set the new 500 points of channel 0
        self.line.set_ydata(self.plotbuffer)
        return self.line,

    # appends data to the ringbuffer
    def addData(self, v):
        self.ringbuffer.append(v)

# Create an instance of an animated scrolling window with text label
parameters
realtimePlotWindowData = RealtimePlotWindow('Realtime LDR
Data','Samples','ADC sensor data',[0,.5])
```

```
realtimePlotWindowFilter = RealtimePlotWindow('Filtered LDR
Data','Samples','Filtered ADC sensor data',[-0.1,.3])

# called for every new sample which has arrived from the Arduino
def callBack(data):
    # send the sample to the plotwindow
    output=yn.doFilter(data)
    realtimePlotWindowFilter.addData(output)
    realtimePlotWindowData.addData(data)
    jump.check(output) #checks if a jump is needed

class Jump:
    def __init__(self,fs):
        self.samplingRate = fs
        self.timestamp = 0
        self.i=0
        self.jump=0
        self.oldt=0
        self.capture=np.zeros(1000)

    def check(self,output):
        self.timestamp += (1 / self.samplingRate)
        self.capture[self.i]=output #stores data for analysis
        if (output>0.01 and self.timestamp-self.oldt>0.25 and
self.timestamp>5): #peak detection
            self.jump+=1
            wsh.SendKeys(" ")
            #we could write to a stepper motor here!
            self.oldt=self.timestamp
        self.i+=1

def coeffHP(f,fs):
    """
    Calculates the highpass coefficients for the IIR class

    Parameters
    ----------
    f : Highpass cutoff - MUST be normalized to fs
    fs : Sampling frequency

    """
    w=2*np.pi*f/fs #digital frequency scaled to pi
    ohm=2*fs*np.tan(w/2) #prewarped freq
    C=1.414*ohm #sqrt2*prewarped freq
    D=ohm**2 #prewarped freq^2
    coeffs=np.zeros(6)

    #calculates coeffs
    coeffs[3]=1.0
    coeffs[4]=(2*D-2)/(1+C+D)
    coeffs[5]=(1-C+D)/(1+C+D)
    coeffs[0]=1/(1+C+D)
    coeffs[1]=-2/(1+C+D)
    coeffs[2]=coeffs[0]
    return coeffs

fs=20 #20Hz sampling rate
f1=0.1/fs
coeffs=coeffHP(f1,fs) #calculate coeffs
yn=IIR.IIR2Filter(coeffs) #initiate IIR class with coeffs
jump=Jump(fs)
```

```
# Get the Ardunio board.
board = Arduino(PORT)
# Set the sampling rate in the Arduino
board.samplingOn(1000 / fs)
# Register the callback which adds the data to the animated plot
board.analog[0].register_callback(callBack)
# Enable the callback
board.analog[0].enable_reporting()
board.analog[1].enable_reporting()
# show the plot and start the animation
plt.show()
```

Appendix E

```
"""
All credit due OSNOVE
https://github.com/osnove/other/blob/master/complex_plane.py
"""
import matplotlib.pyplot as plt
import numpy as np

def complex_plane2(z,axis_type=0):
    """Creates complex plane and shows complex numbers as vectors
(complexors)

    Parameters
    ----------
    z : array of complex values
        array of complex values to be shown
    axis_type : int
        three types of shapes of complex plane:
        0 : symple
        1 : with grid
        2 : moved axis to middle
    -----------------------------------
    # Example
    z=[20+10j,15,-10-10j,5+15j]
    complex_plane2(z,2) """

    w=max(np.abs(z))
    fig, ax = plt.subplots()

    if axis_type==0:
        plt.axis("off")
        plt.text(-0.15*w, 0.8*w, "Im", fontsize=14)
        plt.text( 0.8*w,-0.15*w, "Re", fontsize=14)
    elif axis_type==1:
        plt.axis("on")
        plt.grid()
        plt.text(-0.15*w, 0.8*w, "Im", fontsize=14)
        plt.text( 0.8*w,-0.15*w, "Re", fontsize=14)
    else:
         # Move left y-axis and bottim x-axis to centre, passing through
(0,0)
        ax.spines['left'].set_position('center')
        ax.spines['bottom'].set_position('center')

        # Eliminate upper and right axes
        ax.spines['right'].set_color('none')
        ax.spines['top'].set_color('none')
        # Show ticks in the left and lower axes only
        ax.xaxis.set_ticks_position('bottom')
        ax.yaxis.set_ticks_position('left')

        ax.set_xlabel('                                                    Re
[]')
        ax.set_ylabel('                                                    Im
[]')

    plt.xlim(-w,w)
    plt.ylim(-w,w)
    plt.arrow(0, -w, 0, 1.9*w, head_width=w/20, head_length=w/20, fc='k',
ec='k');
    plt.arrow(-w, 0, 1.9*w, 0, head_width=w/20, head_length=w/20, fc='k',
ec='k');

    for i in range(len(z)):
        fi_a=np.angle(z[i])
        x=z[i].real -abs(w)/20*np.cos(fi_a)
        y=z[i].imag-abs(w)/20*np.sin(fi_a)
        plt.arrow(0, 0, x, y, head_width=w/20, head_length=w/20, fc='b',
ec='b');
    plt.show()

z=[-0.5205 + 0.5245j,-0.5205 - 0.5245j] # array of complex values
complex_plane2(z,1) # function to be called
```

Declaration of Originality and Submission Information

I affirm that this submission is my own / the groups original work in accordance with the University of Glasgow Regulations and the School of Engineering Requirements.

1. Student Number: 2310737h   Student Name: Arric Hamilton

2. Student Number: 2298968p   Student Name: Ishan Patel
