DSP Assignment 3
IIR Filter

Miss Ramon Suwanban (2495594S)
Mr Wikara Gunawan (2397833G)

due 7 December 2020 (3PM)

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.

Student Number: 2495594S Student Name: Miss Ramon Suwanban
Student Number: 2397833G Student Name: Mr Wikara Gunawan

1 Introduction

IIR filters are invented to replace FIR filter in applications that need fast computation time. IIR filters can be used to remove DC line/noise from various types of sensors. In this experiment, a web-based game is controlled with filtered readings from analogue reading of an accelerometer. Noises from the sensors are filtered to produce desired controlling signal and the DC line is removed for the uses of threshold conditions.

2 Method and Results

2.1 Experiment Documentations

Dinosaur_trial1 is a WebGL format game created by us using Unity and free assets from Unity assets store. The character controlling script was taken from this tutorial(https://youtu.be/dwcT-Dch0bA). The aim of the game is to control(jump) the character to avoid all the spikes and reach the goal. If the game is played without using Arduino, accelerometer and the python script provided, the spacebar key on the keyboard can also be used to control(jump) the character. Instructions on how to open the game in a web browser is given on GitHub. (https://github.com/Lilypads/IIR_application)

The **realtime_iir_main.py** was developed from demo file of pyFirmata2. All rights to the original owner.

2.1.1 Photo of the Setup

The setup consists of an Arduino UNO, ADXL335 Accelerometer and a USB Cable. Jumper wires are connected from the X, Y, and Z axis readings from the Accelerometer.

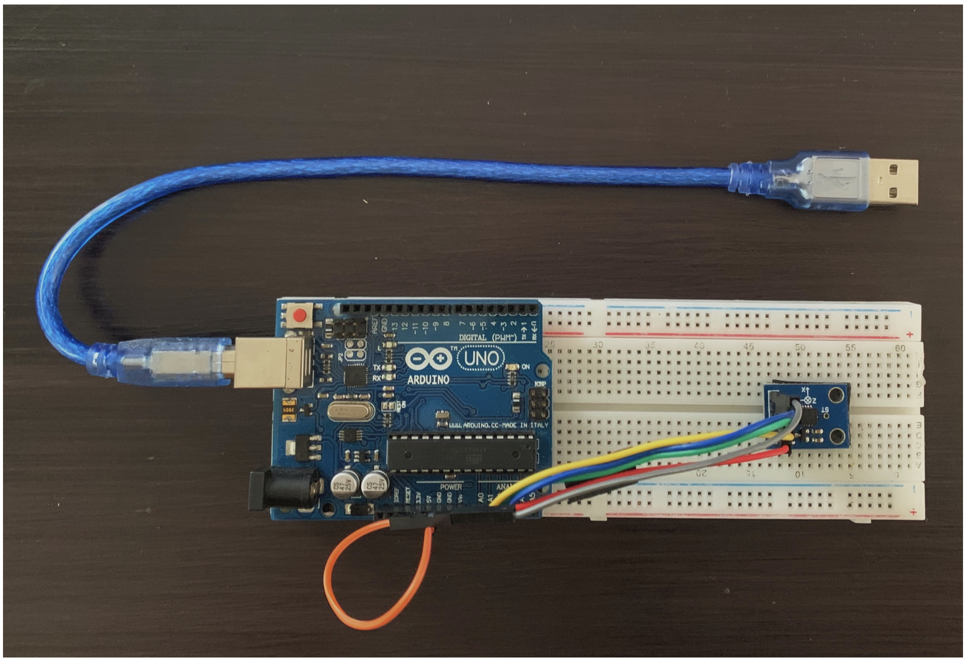

Figure 1: Sensor Configuration Setup

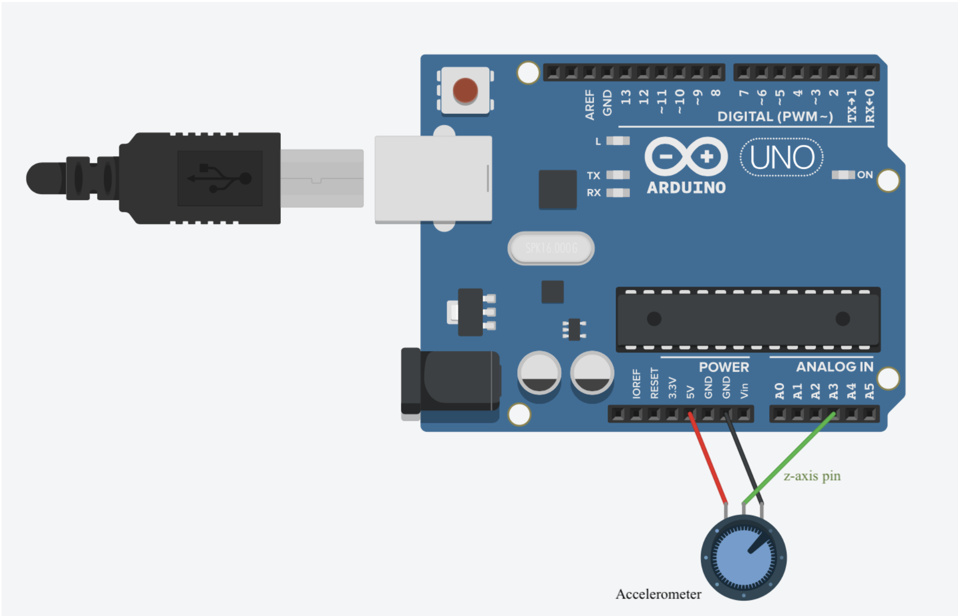

Figure 2: Wire Diagram

From figure 2, we need z-axis pin to connect to analog pin 3 of Arduino to comply with **realtime_iir_main.py** for jumping in **Dinosaur_trial1** game. x and y axis pins can be connected to any other analog pins, but they weren't used for **Dinosaur_trial1** game. If were to, player can configure the code to change analog pin 3 to other pins for wire configuration at the following lines of code.

```
# Register the callback which adds the data to the animated plot
board.analog[3].register_callback(callBack)        # analog pin a3 of Arduino connects to z-axis of the sensor
# Enable the callback
board.analog[3].enable_reporting()
```

Listing 1: realtime_iir_main.py

2.1.2 Dataflow Diagrams

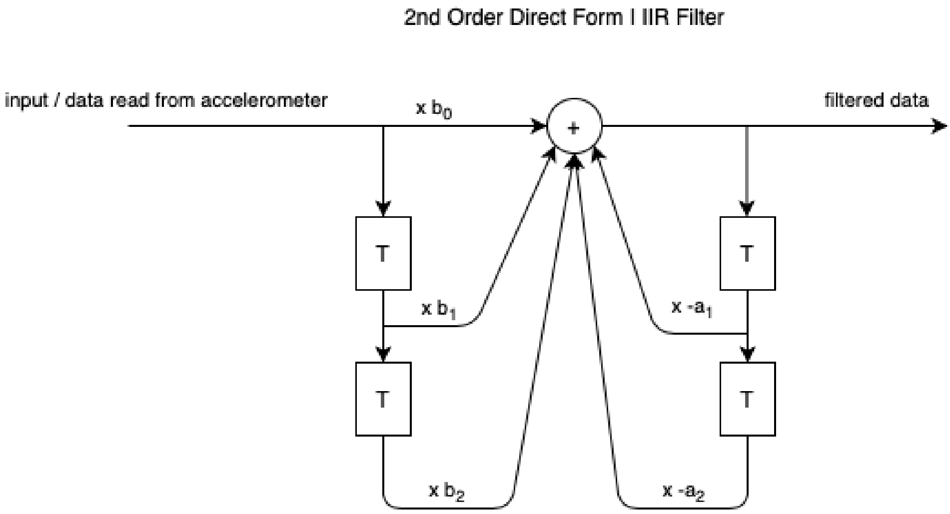

Figure 3: IIR 2nd Order Direct Form I Dataflow Diagram

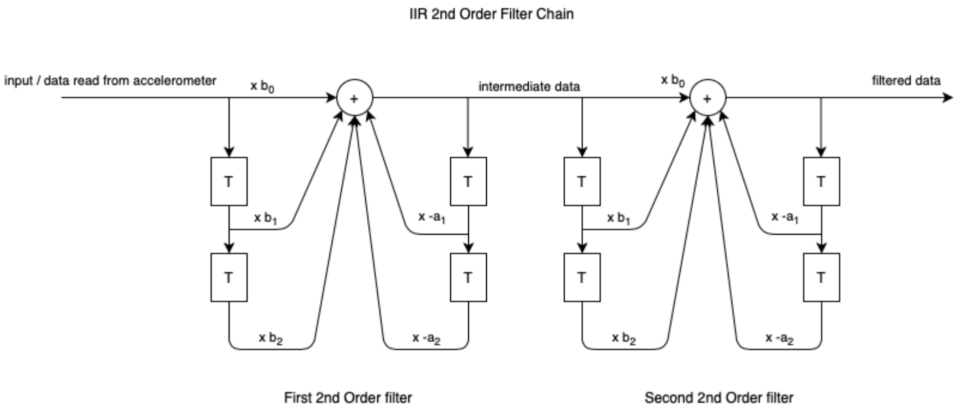

Figure 4: IIR 2nd Order Chain of 2 Dataflow Diagram

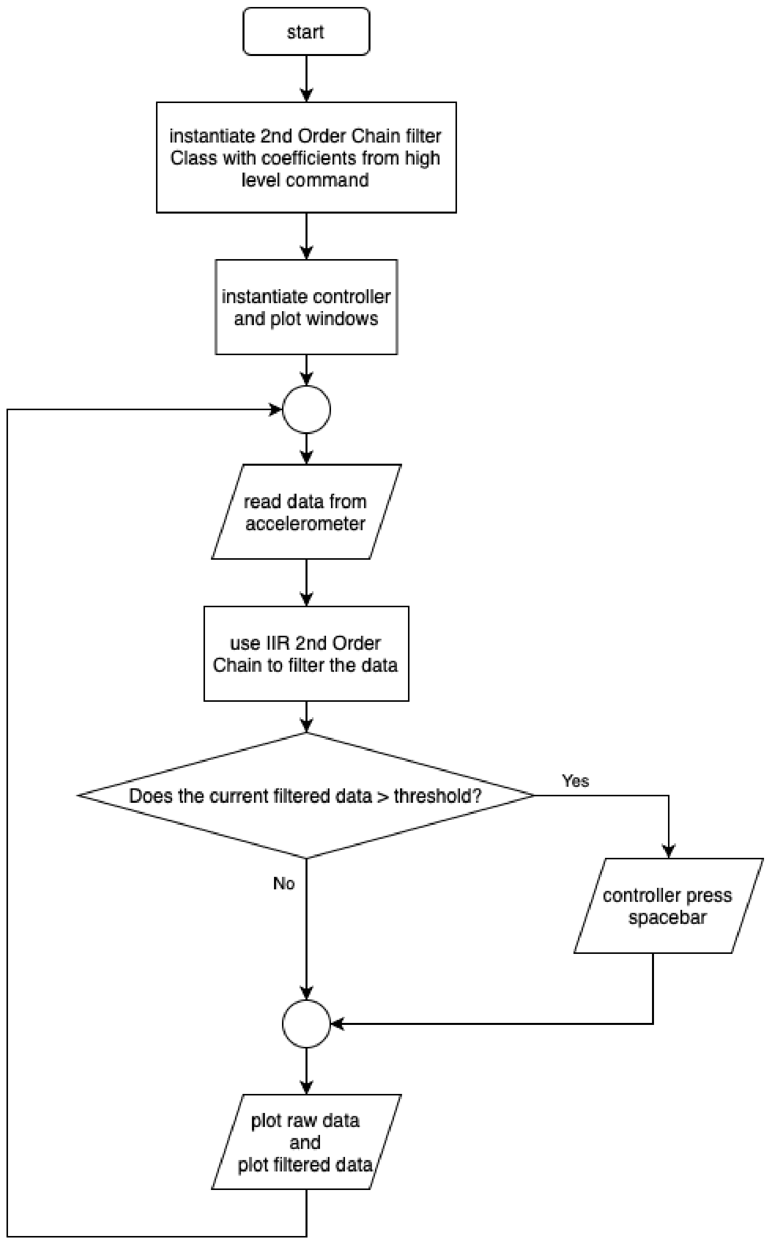

Figure 5: IIR Application Dataflow Diagram

2.1.3 YouTube and GitHub Resource

The following GitHub repository and youtube video link could be found below.
https://github.com/Lilypads/IIR_application
https://youtu.be/T_xc3NxIb94

2.2 Result Presentation

We plot the readings on graphs created by **RealtimePlotWindow** class, which is provided from demo file of pyFirmata2, as graphs would be best to represent changes in acceleration.

Figure 6 and figure 7 show the readings of the accelerometer before and after filtering respectively. Figure 6 shows an offset of about 0.38, which indicates DC line interference. Additionally, there are unwanted high frequency peaks in the readings which are considered noise. Even when holding the accelerometer still, it would create these peaks. Figure 7 shows the filtered readings which is not subject to DC line and high frequency noise.

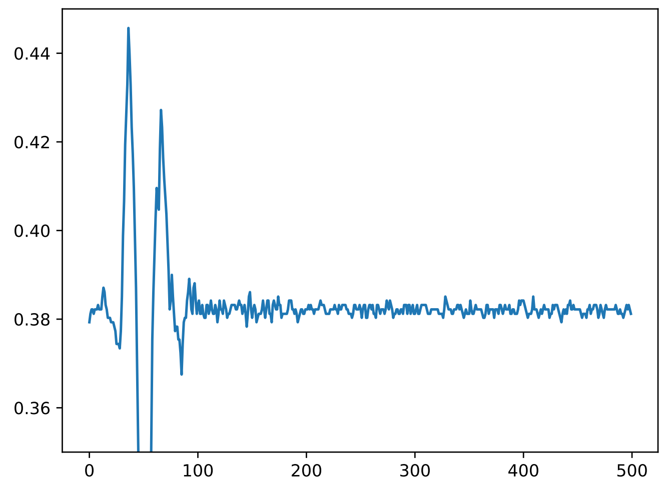

Figure 6: Readings of Accelerometer to Positive Z-Axis Before Filtering

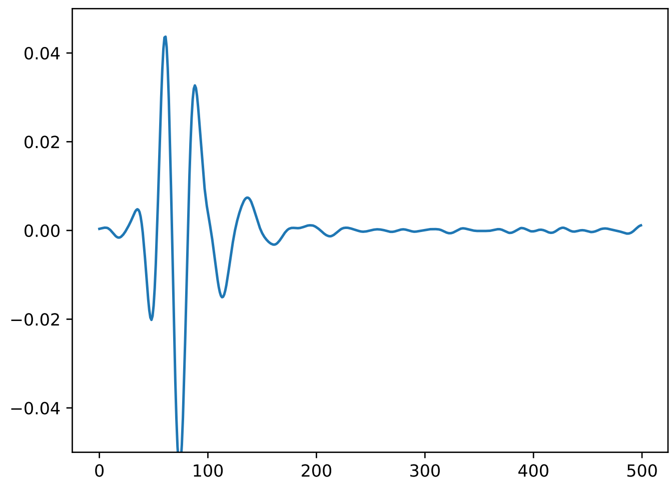

Figure 7: Readings of Accelerometer to Positive Z-Axis After Filtering

2.3 Sampling Rate Check and Jitter Check

```python
class callbacks:
    def __init__(self):

        # Get the Ardunio board.
        self.board = Arduino('COM3')
        self.count = 0
        self.samplingRate = 100


    def initial(self):
        # Set the sampling rate in the Arduino
        self.board.samplingOn(1000 / self.samplingRate)
        # Register the callback which adds the data to the countsample
        self.board.analog[3].register_callback(self.countsample)
        # Enable the callback
        self.board.analog[3].enable_reporting()
        # Sleep for 10 Seconds and let the countsample take samples
        delay = 10
        time.sleep(delay)

        print("Number of Sample:", self.count)
        print("Sampling rate:", self.count/delay)

    def countsample(self, data):
        # Increment every sample taken
        self.count +=1

# Declare callbacks class as call_backs
call_backs = callbacks()
# Start the Initial function from the class initial which run the true
    callback inside the class
call_backs.initial()
```

Listing 2: Sampling Rate Check.py

To check the sampling rate, an increment counter is temporary attached to the main program to check the amount of sample taken under the time.sleep() function for 10 seconds. To attach it with the pyFirmata2 callbacks, a class is created. It consists of board settings and a callback function to count the sample for 10 seconds. The result of the sampled counts results to 1000 samples per 10 seconds, which is 100 Hz. The class callbacks() is initialised ant the function initial is called.

To check the jitter, we could input the sensor with a sinusoidal wave of the same frequency as the sampling rate. Since sampling rate and the input signal is the same frequency, the reading should become a DC line. (Because the same part of a sinusoidal is sampled for every single sinusoidal wave) If the readings do not look exactly like a DC line, then jitter would have occurred and the value read is slightly shifted to left of right of where it should actually have been causing the reading to not look like a DC line. An easy way to do this is to use 50 Hz noise as the input(by putting a finger on the sensor) and change the sampling rate in the program to 50 Hz and observe the readings.

2.4 Required Filter Response

The readings from accelerometer is subject to DC line. To use a threshold to control movement in a game, the DC line must first be eliminated. The sensitivity of the sensor is too high with higher frequencies(just trying to hold the accelerometer steadily already creates waves), so we filtered out the higher frequency to lower the sensitivity and make it applicable for human players.

2.5 SOS Coefficients Generation

In the **filter_coefficient.py**, we design our IIR coefficients using python high level command: butterworth and chebyshev type II from **scipy.signal** library. We experimented couple of orders and cutoff frequencies to reach our desired controller response. The final decision of our filter frequency response is shown in figure 8 and figure 9.

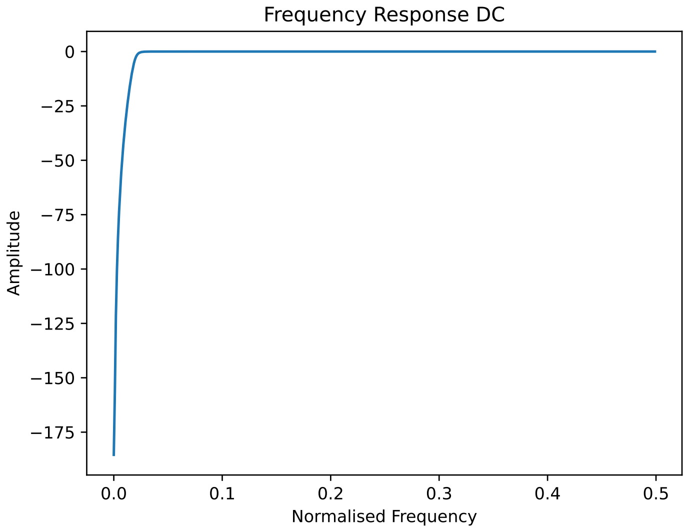

Figure 8: Frequency Response of DC filter (Butterworth)

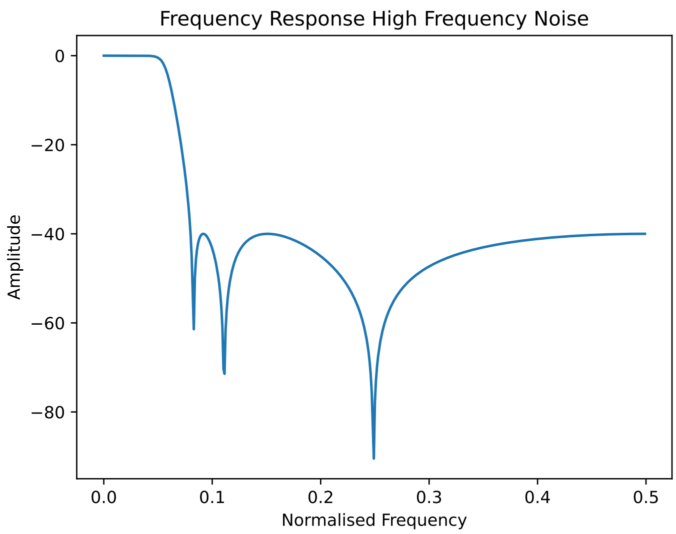

Figure 9: Frequency Response of Low Pass filter (ChebyShevII)

2.6 IIR filter Class

We used Direct Form I 2nd order IIR filter to implement **IIR2Filter** class. There is only 1 accumulator and 2 delay lines in the Direct Form I 2nd order IIR filter. The **coefficients** given to initiate this class should be an array of 6 elements containing coefficient **b0, b1, b2, a0, a1, a2** respectively. The coefficients from our high level python command, using sos, come as sets of 6 elements arrays. In the **IIRFilter** class, we input one of the arrays at a time to instantiate the filters chain.

```
class IIR2Filter:
    def __init__(self,coefficients):    # need array of 6 elements input
        self.b0 = coefficients[0]
        self.b1 = coefficients[1]
        self.b2 = coefficients[2]
        self.a1 = coefficients[4]
        self.a2 = coefficients[5]
        self.acc = 0;        # accumulator
        self.b_x1 = 0;       # input x delay buffer
        self.b_x2 = 0;
        self.b_y1 = 0;       # output y delay buffer
        self.b_y2 = 0;

    def dofilter(self,x):
        self.acc = (x*self.b0) + (self.b_x1*self.b1) + (self.b_x2*self.b2) - (self.b_y1*self.a1) - (self.b_y2*self.a2)
        self.b_x2 = self.b_x1      # update delay lines
        self.b_x1 = x
        self.b_y2 = self.b_y1
        self.b_y1 = self.acc
        return self.acc
```

Listing 3: IIR_filter.py

2.7 Instantiate and 2nd Order Chain filter Class

We instantiate the filters chain in the **__init__** function using for-loop. We then implemented the **dofilter** function to put the input data through all the chain filters and returns the result.

```
class IIRFilter:
    def __init__(self,coefficients_array):
        self.filters = []
        for i in range(len(coefficients_array)):
            self.filters.append(IIR2Filter(coefficients_array[i])) # instantiate 2nd order filters

    def dofilter(self,u):
        for i in range(len(self.filters)):
           u = self.filters[i].dofilter(u)
        return u
```

Listing 4: IIR_filter.py

In the **realtime_iir_main.py** we instantiate the filter chain classes with our experimented coefficients from **filter_coefficient.py**.

```
# sampling rate: 100Hz
samplingRate = 100

# DC line filter
fc = 2      # cutoff frequency
sosDC = sig.butter(6,fc/samplingRate*2,btype='high',output='sos')

# deleting high frequency noise
fc = 8      # cutoff frequency
```

```python
sosLP = sig.cheby2(6,40,fc/samplingRate*2,btype='low',output='sos')

# instantiate the 2nd order chain class
filterDC = IIRFilter(sosDC)
filterLP = IIRFilter(sosLP)
```

Listing 5: realtime_iir_main.py

In the **callBack** function, we used the instantiated chain filter classes to filter the data read from accelerometer. We plot it on a graph and use threshold as the condition to output game controller action.

```python
# called for every new sample which has arrived from the Arduino
def callBack(data):

    # send the sample to the plotwindbow
    # add any filtering here:
        # data = self.myfilter.dofilter(data)

    output = filterDC.dofilter(data)
    output = filterLP.dofilter(output)

    # send jump key operation to control the game
    if output >= 0.04:
        keyboard.press(Key.space)
        keyboard.release(Key.space)
        #print("SpaceJump")

    realtimePlotWindow.addData(data,)
    realtimePlotWindow2.addData(output)
```

Listing 6: realtime_iir_main.py

3 Discussion

The implementation of IIR Filters to remove DC line and high frequency noise has been successful. The control signal from using butterworth and chebyshevII filters result to an error-free jump movement on the game. Since the **Dinosaur_trial1** game is programmed to not allow jumping in mid-air, the segment of the signal that still is above the threshold after it initially reached the threshold doesn't affect in multiple jumps. However, further implementation can be done to the callback function to delete that segment completely to make sure it only jump once in other games.

4 Appendix

4.1 IIR_filter.py

```python
class IIR2Filter:
    def __init__(self,coefficients):    # need array of 6 elements input
        self.b0 = coefficients[0]
        self.b1 = coefficients[1]
        self.b2 = coefficients[2]
        self.a1 = coefficients[4]
        self.a2 = coefficients[5]
        self.acc = 0;        # accumulator
        self.b_x1 = 0;       # input x delay buffer
        self.b_x2 = 0;
        self.b_y1 = 0;       # output y delay buffer
        self.b_y2 = 0;

```

```python
    def dofilter(self,x):
        self.acc = (x*self.b0) + (self.b_x1*self.b1) + (self.b_x2*self.b2) - (self.b_y1*self.a1) - (self.b_y2*self.a2)
        self.b_x2 = self.b_x1       # update delay lines
        self.b_x1 = x
        self.b_y2 = self.b_y1
        self.b_y1 = self.acc
        return self.acc

class IIRFilter:
    def __init__(self,coefficients_array):
        self.filters = []
        for i in range(len(coefficients_array)):
            self.filters.append(IIR2Filter(coefficients_array[i])) # instantiate 2nd order filters

    def dofilter(self,u):
        for i in range(len(self.filters)):
           u = self.filters[i].dofilter(u)
        return u
```

Listing 7: IIR_filter.py

4.2 filter_coefficient.py

```python
import scipy.signal as sig
import matplotlib.pyplot as pyplot
import numpy as np

fs = 100     # sampling frequency
fc = 2       # cutoff frequency

# DC line filter
b,a = sig.butter(6,fc/fs*2,btype='high')
sos = sig.butter(6,fc/fs*2,btype='high',output='sos')

w,h = sig.freqz(b,a)
pyplot.figure(1)
pyplot.plot(w/np.pi/2,20*np.log10(np.abs(h)))
pyplot.title('Frequency Response DC')
pyplot.xlabel('Normalised Frequency')
pyplot.ylabel('Amplitude')

# deleting high frequency noise
fc = 8       # cutoff frequency
b,a = sig.cheby2(6,40,fc/fs*2,btype='low')
sos = sig.cheby2(6,40,fc/fs*2,btype='low',output='sos')

w,h = sig.freqz(b,a)
pyplot.figure(2)
pyplot.plot(w/np.pi/2,20*np.log10(np.abs(h)))
pyplot.title('Frequency Response High Frequency Noise')
pyplot.xlabel('Normalised Frequency')
pyplot.ylabel('Amplitude')
```

Listing 8: filter_coefficient.py

4.3 realtime_iir_main.py

```python
from pyfirmata2 import Arduino
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IIR_filter import IIRFilter
from pynput.keyboard import Key, Controller # Add keyboard controller
```

```python
import scipy.signal as sig

# Realtime oscilloscope at a sampling rate of 50Hz
# It displays analog channel 0.
# You can plot multiple chnannels just by instantiating
# more RealtimePlotWindow instances and registering
# callbacks from the other channels.


#PORT = Arduino.AUTODETECT
# PORT = '/dev/ttyUSB0'

# Creates a scrolling data display
class RealtimePlotWindow:

    def __init__(self, ylim1,ylim2):
        # create a plot window
        self.fig, self.ax = plt.subplots()
        # that's our plotbuffer
        self.plotbuffer = np.zeros(500)
        # create an empty line
        self.line, = self.ax.plot(self.plotbuffer)
        # axis
        self.ax.set_ylim(ylim1,ylim2)
        # That's our ringbuffer which accumluates the samples
        # It's emptied every time when the plot window below
        # does a repaint
        self.ringbuffer = []
        # add any initialisation code here (filters etc)
        # 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 9
        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
# To plot more channels just create more instances and add callback handlers below
realtimePlotWindow = RealtimePlotWindow(0.35,0.45)      #input custom plot ylim
realtimePlotWindow2 = RealtimePlotWindow(-0.05,0.05)

# sampling rate: 100Hz
samplingRate = 100

# DC line filter
fc = 2      # cutoff frequency
sosDC = sig.butter(6,fc/samplingRate*2,btype='high',output='sos')

# deleting high frequency noise
fc = 8      # cutoff frequency
sosLP = sig.cheby2(6,40,fc/samplingRate*2,btype='low',output='sos')

# instantiate the 2nd order chain class
```

```python
filterDC = IIRFilter(sosDC)
filterLP = IIRFilter(sosLP)

# instantiate keyboard controller
keyboard = Controller()

# called for every new sample which has arrived from the Arduino
def callBack(data):

    # send the sample to the plotwindbow
    # add any filtering here:
        # data = self.myfilter.dofilter(data)

    output = filterDC.dofilter(data)
    output = filterLP.dofilter(output)

    # send jump key operation to control the game
    if output >= 0.04:
        keyboard.press(Key.space)
        keyboard.release(Key.space)
        #print("SpaceJump")

    realtimePlotWindow.addData(data,)
    realtimePlotWindow2.addData(output)

# Get the Ardunio board.
board = Arduino('COM3')


# Set the sampling rate in the Arduino
board.samplingOn(1000 / samplingRate)

# Register the callback which adds the data to the animated plot
board.analog[3].register_callback(callBack)         # analog pin a3 of
     Arduino connects to z-axis of the sensor
# Enable the callback
board.analog[3].enable_reporting()


# show the plot and start the animation
plt.show()

# needs to be called to close the serial port
# board.exit()

print("finished")
```

Listing 9: realtime_iir_main.py

4.4 realtime_iir_main.py with sampling rate check

```python

from pyfirmata2 import Arduino
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IIR_filter import IIRFilter
from pynput.keyboard import Key, Controller # Add keyboard controller
import scipy.signal as sig
import time

# Realtime oscilloscope at a sampling rate of 50Hz
# It displays analog channel 0.
# You can plot multiple chnannels just by instantiating
# more RealtimePlotWindow instances and registering
# callbacks from the other channels.
```

```

#PORT = Arduino.AUTODETECT
# PORT = '/dev/ttyUSB0'

# Creates a scrolling data display
class RealtimePlotWindow:

    def __init__(self, ylim1,ylim2):
        # create a plot window
        self.fig, self.ax = plt.subplots()
        # that's our plotbuffer
        self.plotbuffer = np.zeros(500)
        # create an empty line
        self.line, = self.ax.plot(self.plotbuffer)
        # axis
        self.ax.set_ylim(ylim1,ylim2)
        # That's our ringbuffer which accumluates the samples
        # It's emptied every time when the plot window below
        # does a repaint
        self.ringbuffer = []
        # add any initialisation code here (filters etc)
        # 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 9
        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
# To plot more channels just create more instances and add callback
    handlers below
realtimePlotWindow = RealtimePlotWindow(0.35,0.45)
realtimePlotWindow2 = RealtimePlotWindow(-0.05,0.05)

# sampling rate: 100Hz
samplingRate = 100

# DC line filter
fc = 2      # cutoff frequency
sosDC = sig.butter(6,fc/samplingRate*2,btype='high',output='sos')

# deleting high frequency noise
fc = 8      # cutoff frequency
sosLP = sig.cheby2(6,40,fc/samplingRate*2,btype='low',output='sos')

# instantiate the 2nd order chain class
filterDC = IIRFilter(sosDC)
filterLP = IIRFilter(sosLP)

# instantiate keyboard controller
keyboard = Controller()

# called for every new sample which has arrived from the Arduino

class callbacks:
    def __init__(self):
```

```python

        # initialise timer, count and samplingRate
        self.timer = 0
        self.count = 0
        self.samplingRate = 100
        # Get the Ardunio board. It is not always COM3
        self.board = Arduino('COM3')

    def initial(self):
        # Set the sampling rate in the Arduino
        self.board.samplingOn(1000 / self.samplingRate)

        # Register the callback which adds the data to the animated plot
        #self.board.analog[3].register_callback(self.countsample)
        self.board.analog[3].register_callback(self.callBack)# analog pin a3 of Arduino attach to callback countsample
        # Enable the callback
        self.board.analog[3].enable_reporting()

        # Sleep for 10 Seconds and let the countsample take samples
        delay = 10
        time.sleep(delay)

        print("Number of Sample:", self.count)
        print("Sampling rate:", self.count/delay)

    def countsample(self, data):
        # Increment every sample taken
        self.count +=1

    def callBack(self, data):
        global count
        global tsec
        # send the sample to the plotwindbow
        # add any filtering here:
        # data = self.myfilter.dofilter(data)

        output = filterDC.dofilter(data)
        output = filterLP.dofilter(output)

        # Set jump threshold
        if output >= 0.04:
            keyboard.press(Key.space)
            keyboard.release(Key.space)
            #print("SpaceJump")

        # Plot data and output
        realtimePlotWindow.addData(data)
        realtimePlotWindow2.addData(output)

# Declare callbacks class as call_backs
call_backs = callbacks()
# Start the Initial function from the class initial which run the true callback inside the class
call_backs.initial()

# show the plot and start the animation
plt.show()

# needs to be called to close the serial port
# board.exit()

print("finished")
```

Listing 10: realtime_iir_main.py with sampling rate check