# Digital Signal Processing

IIR Assignment

Accelerometer Base Fall Detector

Szymon Gula (2601553G@student.gla.ac.uk)

Allan Hernandez (2589702H@student.gla.ac.uk)

Dr Bernd Porr

Dr Nicholas Bailey

6 December 2020

**Ex 1**

**Fall detection system**

Falls are significantly dangerous, as they may cause serious injuries such as head trauma or hip fracture. Lonely living elderly and people with pre-existing medical disabilities or long-term health conditions are the most vulnerable group to falls. In the event of a fall, immediate assistance may be required to provide timely support and prevent further complications. However, in some cases, the affected person may not be able to get up and reach out for help. We designed a proof-of-concept of fall detection system application using an accelerometer controlled by an Arduino Mega and infinite impulse response (IIR) filtering to detect hard falls that in further development can support emergency call if needed.

The sensor uses a 3-axis measurement system to measure acceleration in the x, y, and z axes. Its output signals are analogue voltages proportional to the measured acceleration. For the proof-of-concept porpoise, the fall detection application identifies an abrupt change in acceleration as a fall. The resulting filtered signal removes noise and prevents inconsequent movements to trigger a fall event (false positive).

**YouTube Link:**

A presentation video with demos of our fall detection application can be seen at: https://youtu.be/YQ3CkbUhUGs

Our hardware set up for the project consists of an ADXL335 accelerometer sensor module and an Arduino Mega 2560 Rev3 board. To filter the analogue readings from the sensor in real time, we interfaced the Arduino board with Python through MIT licenced pyFirmata2 module by author Bernd Porr, Github repository, https://github.com/berndporr/pyFirmata2.

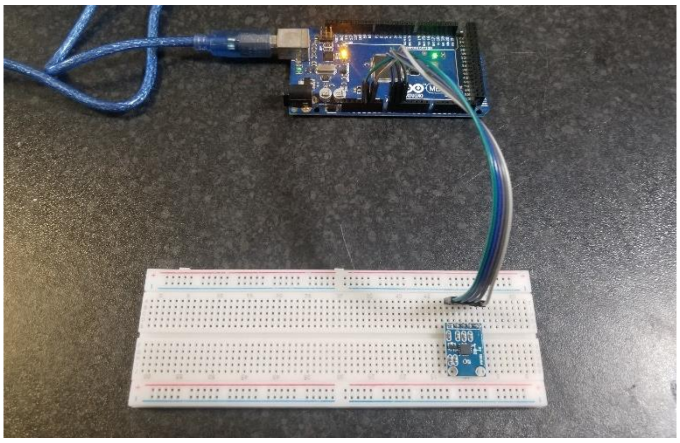

*Figure 1: Project setup - ADXL accelerometer sensor module and Arduino Mega 2560 Rev 3 (Own photograph, 2020)*

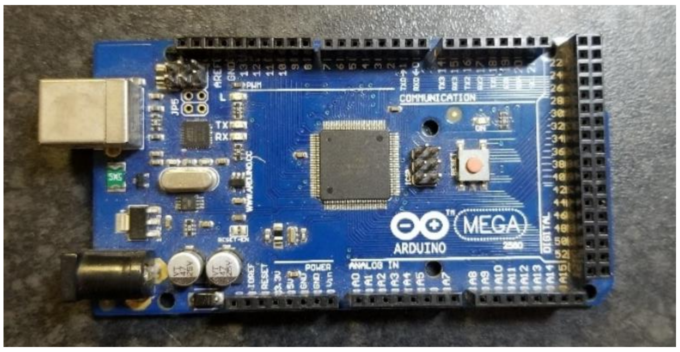

*Figure 2: Arduino Mega 2560 Rev3 (Own photograph, 2020)*

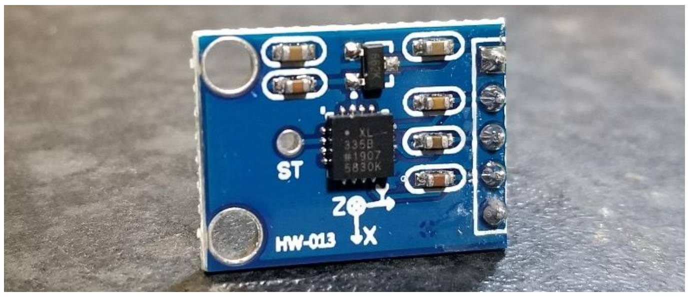

*Figure 2: ADXL355 Accelerometer sensor module (Own photograph, 2020)*

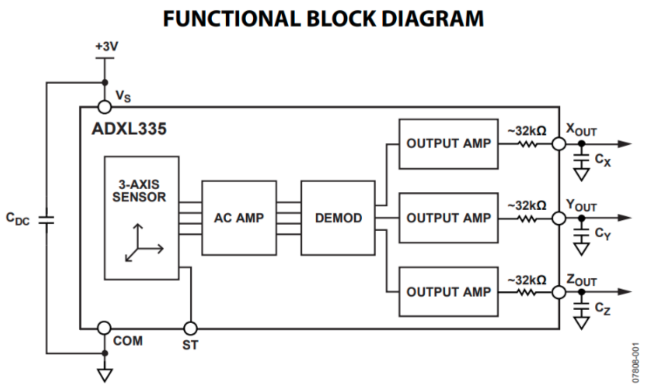

*Figure 3: Accelerometer functional block diagram. From Analog Devices, Inc., 2009, retrieved from:* [*https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf*](https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf)

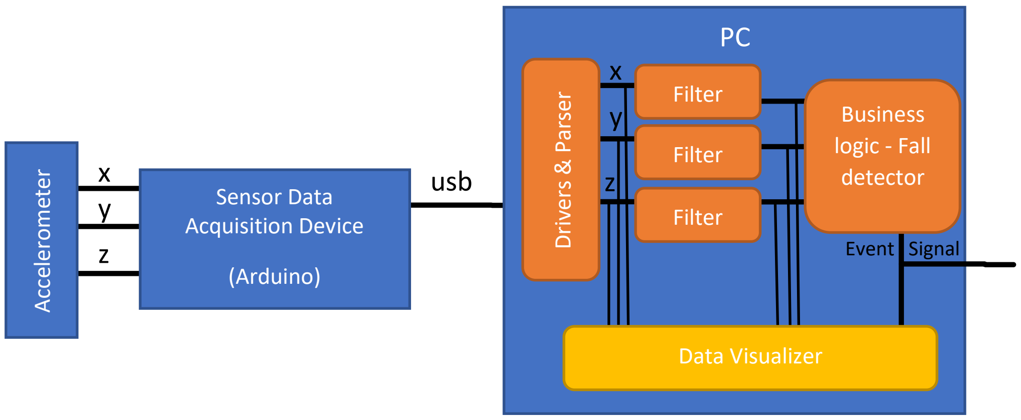

*Figure 4: System block design with presented signal flow*

We connected the sensor’s output pins for acceleration of the x, y, and z axes in the Arduino’s analogue input pins (A0, A1, and A2, respectively). Next, we enabled these pins for reading in Python based on the available pyFirmata2 realtime_two_channel_scope.py module example, which is the baseline for our code. To realize our project goals, we implemented a wrapper class ArduinoScope that acts as a proxy between Arduino interface and visualization class (QtPanningPlot). It is implemented as a context manager that supports object initialization, object recourses allocation and recourses unlocking. The mentioned design strategy provides an easy to use and understand interface for the user. In addition, we extended the plotting class to support multiple class instances, to support multiple lines with different colour and event signalization.

**Ex 2**

We setup the system to collect samples from three analogue ports with frequency of 100Hz. To verify the system, we run our program for about 70 seconds and log the data to the file. With recorded data postprocessing we get following results:

```
>>> execution time [s]: 72.81499457359314
>>> recorded samples [-]: 7293
>>> actual sampling rate [Hz]: 100.15817685501081
```

The above-mentioned result confirms that system sampling rate is roughly 100Hz.

**Ex 3**

Our device uses an analogue accelerometer as the data source. Due to significant high frequency noise, we need apply filter to incoming data before applying any business logic (feature function – event detector for the “fall” signal). Also, for our application we want to preserve the DC signal component, that represent the gravitational acceleration. Furthermore, to reduce the time delay of the filtered signal, we decided to use IIR filter.

Given above mentioned assumptions, we selected the low-pass frequency with cut-off frequency at 2.5 Hz (or 0.05 Nyquist normalized frequency). To ensure significant reduction of the high frequencies on output signal we selected 40dB of attenuation on the stop-band filter region. Finally, to smooth the magnitude system response for high frequency, we decided to use three second order IIR filters.

To find IIR coefficients for each of the second order IIR we use the SciPy signal library. To be specific, we use Cheby2 method.

```
freq = 2.5 / 50
sos = signal.cheby2(6, 40, freq * 2, output='sos')
sos
>>>[[919e-5, -837e-5, 919e-5, 1, -1.49, 0.562],
>>> [1,       -1.808,   1,       1, -1.68, 0.744],
>>> [1,       -1.89,    1,       1, -1.87, 0.919]]
```

Where each row represents second-order filter. First tree elements representing numerator coefficients, other are denominator coefficients.

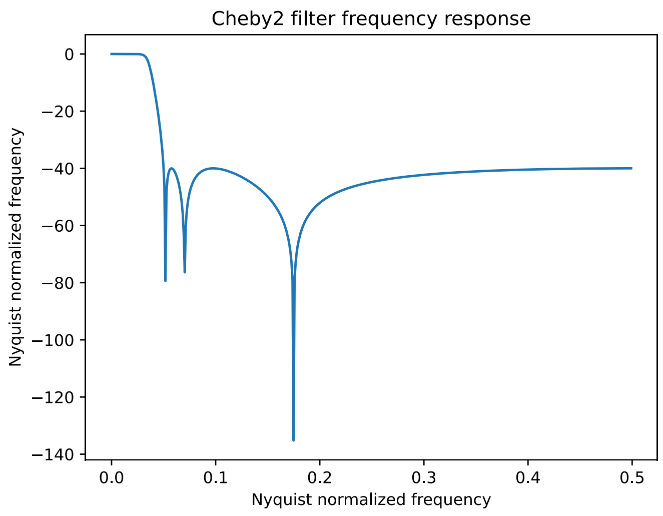

Figure 5: Low-pass IIR filter frequency response (chain of three second order IIR blocks

## Ex 4

For the IIR filter implementation we used the Iterator design pattern. This design pattern contains of the wrapper class that iterates over executable objects that modify the input and pass it to next element in the line.

In our code, the Iterator class is:

```
class IIRFilter:
    def __init__(self, sos):
        self.filter_chain = [IIR2Filter(i[:3], i[3:]) for i in sos]

    def filter(self, input):
        out = input
        for fil in self.filter_chain:
            out = fil.filter(out)
        return out
```

This class at the creation initialize the chain of the second order IIR filers based on the “SOS” coefficients. When its filter method is executed, the iterator object put the input value into chain of the second order filters.

The executable objects are second order IIR filters.

```
class IIR2Filter:
    def __init__(self, b, a):
        self.a0, self.a1, self.a2, self.b0, self.a1, self.b2 = *a, *b
        self.buffer_m1, self.buffer_m2 = 0, 0

    def filter(self, input):
        in_sum = input - self.a1 * self.buffer_m1 - self.a2 * self.buffer_m2
        out_sum = self.b0 * in_sum + self.b1 * self.buffer_m1 + self.b2 * self.buffer_m2
        self.buffer_m2 = self.buffer_m1
        self.buffer_m1 = in_sum
        return out_sum
```

During object initialization we need to provide a filter's transfer function coefficients, that corresponds to the "SOS" values. The functional method "filter()" calculates current filtered value based on the single input value and saves its state for the future calculation. Such implementation enables the class to be used in the real-time environment.

## Ex 5

We successfully used our Arduino as a real time data acquisition card for reading analogue values from our accelerometer and then applied filtering to reduce noise through digital signal process methods. To visualize the input and processed data we decided to use two figures plot. In the figure six, we plot raw accelerometer data for each axis and the combined space acceleration vector. In the figure seven, we present the same data but filtered using second order IIR filters, with a separate filter for each measurement axis. In addition, in the figure eight, the plot of the combined acceleration vector shows the status of the event. If the fall event was detected, the plot line changes colour to red for 2 seconds, otherwise the curve is white. As shown in figure 8, the red line represents the fall event when the signal passes our set threshold. By comparing the real time original and filtered signals simultaneously, we can conclude that an effective filtering of the accelerometer's analogue signals was implemented. Tests for our application demonstrated that our filter was properly tuned to correctly detect fall events and avoid false positives from actions such as bending over to pick something from the floor.

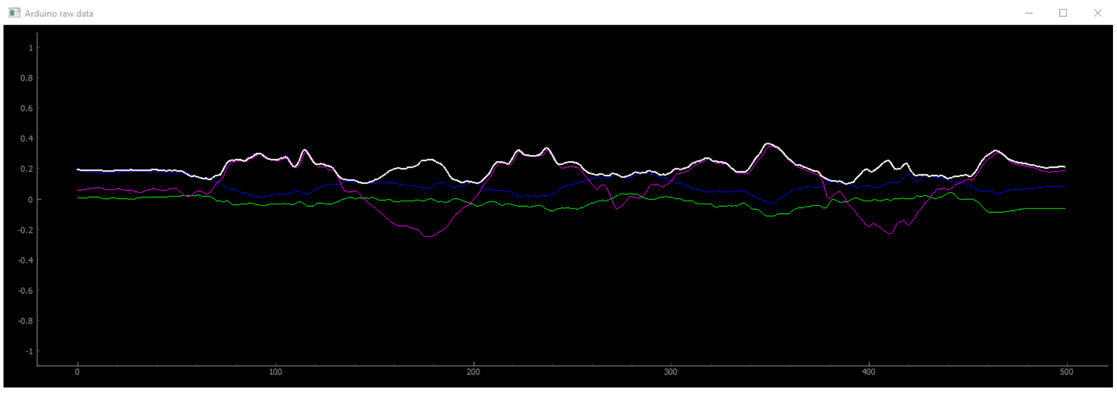

*Figure 6: Plot with original sensor values (white – combined vector, blue – x axis, g – y axis, purple – z axis)*

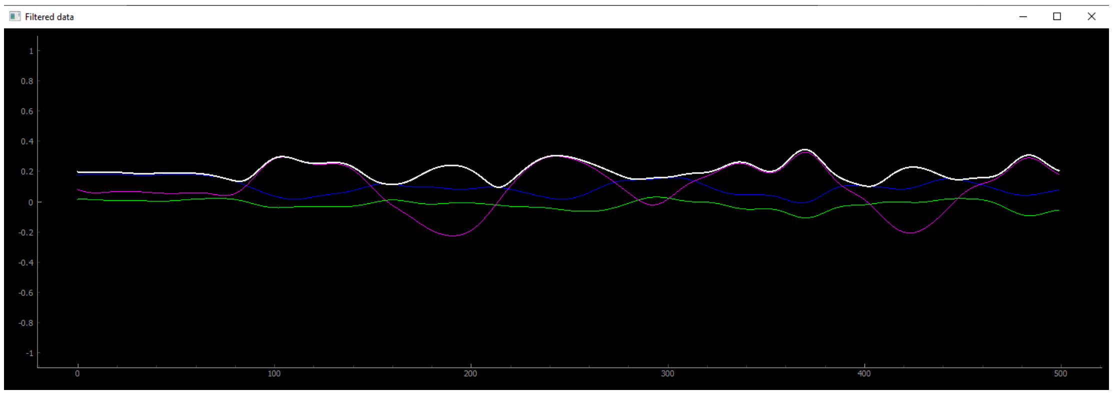

*Figure 7: Plot with filtered sensor values  (white – combined vector, blue – x axis, g – y axis, purple – z axis)*

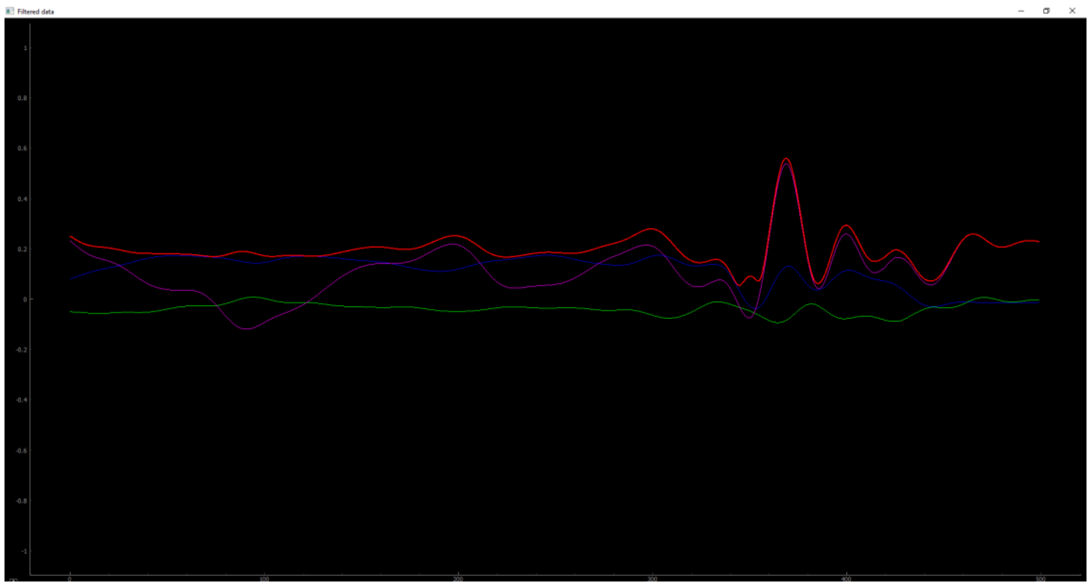

*Figure 8: Plot with filtered values showing "fall event" indicated by red line.*

**YouTube Link:**

A presentation video with demos of our fall detection application can be seen at: https://youtu.be/YQ3CkbUhUGs

**Code can be also found at GitHub repository:**

https://github.com/szgula/UofG_DSP/tree/allan_iir_changes/Project_3_IIR

**realtime_iir_main.py**

```
#!/usr/bin/python3
"""
Base on the MIT licence code:
pyFirmata2/examples/realtime_two_channel_scope.py by @berndporr
"""

import sys
from collections import deque
import pyqtgraph as pg
from pyqtgraph.Qt import QtCore, QtGui
import numpy as np
from pyfirmata2 import Arduino
from IIR_filter import IIRFilter, IIR2Filter, return_filter
import time

class QtPanningPlot:
    """
    Real-time plotting class
    """
    def __init__(self, title):
        x_range = 500
        self.win = pg.GraphicsLayoutWidget()
        self.win.setWindowTitle(title)
        self.plt = self.win.addPlot()
        self.plt.setYRange(-1, 1)
        self.plt.setXRange(0, x_range)
        self.labels = ['x', 'y', 'z', 's']
        self.curve = {name: self.plt.plot() for name in self.labels}
        self.data_ = {name: deque([0] * x_range, maxlen=x_range) for name
in self.labels}
        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()
        self.pens = {'x': pg.mkPen('b', width=1),
                     'y': pg.mkPen('g', width=1),
                     'z': pg.mkPen('m', width=1),
                     'e': pg.mkPen('r', width=2),
                     's': pg.mkPen('w', width=2), }
        self.event = False

    def update(self):
        for label_name in self.labels:
            pen_ = self.pens[label_name]
            if self.event and label_name == 's': pen_ = self.pens['e']

self.curve[label_name].setData(np.hstack(self.data_[label_name]), pen=pen_)

    def addData(self, x, y, z, s, e=False):
        for label_name, input_ in [('x', x), ('y', y), ('z', z), ('s', s)]:
            self.data_[label_name].append(input_)
        self.event = e
```

```
class ArduinoScope:
    """
    Visualization class for proof-of-concept fall detector (Arduino based)
    """
    def __init__(self, filters, debug=False):
        self.PORT = Arduino.AUTODETECT
        self.app = QtGui.QApplication(sys.argv)
        self.qtPanningPlot1 = QtPanningPlot("Arduino raw data")
        self.qtPanningPlot2 = QtPanningPlot("Filtered data")
        self.samplingRate = 100
        self.board = Arduino(self.PORT)
        self.board.samplingOn(1000 / self.samplingRate)
        self.filters = filters
        self.debug_ = debug
        if self.debug_: self.file = open('Arduino Python/xyz_output.txt',
'w')
        self.timestamp = 0
        self.event = False
        self.event_time = 0

    def __enter__(self):
        self.board.analog[0].register_callback(self.callback)
        for i in range(3):
            self.board.analog[i].enable_reporting()
        self.time_s = time.time()
        self.app.exec_()

    @staticmethod
    def _convert_to_normalized_acc(x):
        # 3.3V is a sensor output upper voltage, 5V is a upper A/D
converter voltage, 2 stands for two regions (+ and -)
        translate_val = 3.3 / (5 * 2)
        scalar_val = 1 / translate_val
        norm_acc = (x - translate_val) * scalar_val if x is not None else 0
        return norm_acc

    def _get_filtered_values(self, x, y, z):
        x_f = self.filters[0].filter(x)
        y_f = self.filters[1].filter(y)
        z_f = self.filters[2].filter(z)
        return x_f, y_f, z_f

    @staticmethod
    def _get_3d_vector(x, y, z):
        return (x**2 + y**2 + z**2)**0.5

    def callback(self, data, *args, **kwargs):
        ch0 = self._convert_to_normalized_acc(data)
        ch1 = self._convert_to_normalized_acc(self.board.analog[1].read())
        ch2 = self._convert_to_normalized_acc(self.board.analog[2].read())
        ch0_f, ch1_f, ch2_f = self._get_filtered_values(ch0, ch1, ch2)

        if ch0 and ch1 and ch2:
            vector = self._get_3d_vector(ch0, ch1, ch2)
            vector_f = self._get_3d_vector(ch0_f, ch1_f, ch2_f)
            self.qtPanningPlot1.addData(ch0, ch1, ch2, vector)

            # Handle fall event
            if vector_f > 0.45:
                self.event = True
                self.event_time = 0
            if self.event: self.event_time += 1
            if self.event_time > 200: self.event = False
            self.qtPanningPlot2.addData(ch0_f, ch1_f, ch2_f, vector_f,
self.event)

        self.timestamp += (1 / self.samplingRate)
        if self.debug_:
            self.file.write(f'{round(self.timestamp, 4), ch0, ch1, ch2,
ch0_f, ch1_f, ch2_f} \n')

    def __exit__(self, exc_type, exc_val, exc_tb):
        time_f = time.time()
        self.board.samplingOff()
        self.board.exit()
        if self.debug_:
            self.file.close()
        print(f'execution_time: {time_f - self.time_s} actual sampling rate
{(time_f - self.time_s) / (self.timestamp / self.samplingRate)}')


if __name__ == "__main__":
    my_iirs = [return_filter() for _ in range(3)]
    with ArduinoScope(my_iirs) as scope:
        time_sleep = 100
        time.sleep(time_sleep)
    print('Finished')
```

**IIR_filter.py**

```
import numpy as np
import matplotlib.pyplot as plt


class IIRFilter:
    """ Iterator for the chain of second order IIR filters"""
    def __init__(self, sos):
        """
        :param sos: SOS format according to the SciPy Signal notation
standard
        """
        self.filter_chain = [IIR2Filter(i[:3], i[3:]) for i in sos]

    def filter(self, input):
        """
        Execute the chain of 2'nd order IIR filers
        :param input: single instance value to filter
        :return: filtered value
        """
        out = input
        for fil in self.filter_chain:
            out = fil.filter(out)
        return out


class IIR2Filter:
    """ Second order IIR filter implementation """
    def __init__(self, b, a):
        """
        :param b: iir nominator coefficients
        :param a:iir denominator coefficients
        """
        self.a0, self.a1, self.a2, self.b0, self.b1, self.b2 = *a, *b
        self.buffer_m1 = 0
        self.buffer_m2 = 0

    def filter(self, input):
        """
        Calculate output of the second order IIR filter
        :param input: single instance value
        :return: filtered value
        """
        in_sum = input - self.a1 * self.buffer_m1 - self.a2 *
self.buffer_m2
        out_sum = self.b0 * in_sum + self.b1 * self.buffer_m1 + self.b2 *
self.buffer_m2
        self.buffer_m2 = self.buffer_m1
        self.buffer_m1 = in_sum
        return out_sum

    @staticmethod
    def notch(f, r):
        b0, b1, b2= 1, -2 * np.cos(2*np.pi*f), 1
        a0, a1, a2 = 1,  -2 * r* np.cos(2*np.pi*f), r*r
        return a0, a1, a2, b0, b1, b2


def return_filter():
    """ Return chain of three second order IIR filter for accelerometer
data"""
    freq = 2.5 / 50
    try:
        from scipy import signal
        sos = signal.cheby2(6, 40, freq * 2, output='sos')
    except ModuleNotFoundError:
        sos = np.array([[0.00919895, -0.00837274,  0.00919895,  1., -
1.49146173, 0.56295521],
                        [1., -1.80890255,  1.,  1., -1.68684556,
0.74404189],
                        [1., -1.89526908,  1.,  1., -1.87056254,
0.91962919]])
    my_factory = IIRFilter(sos)
    return my_factory


if __name__ == "__main__":
    from scipy import signal
    with open('output.txt', 'r') as f:
        a = f.readlines()
    data = list(map(float, a))
    sampling_rate = 100  # Hzimp
    noise_frequency = 5  # Hz
    r = 0.99
    temp = IIR2Filter.notch(noise_frequency/sampling_rate, r)
    my_iir = IIR2Filter(temp[3:], temp[:3])

    b, a = signal.cheby2(6, 40, 0.05*2)
    w, h = signal.freqz(b, a)
    freq = 5 / 50
    sos = signal.cheby2(6, 40, freq*2, output='sos')

    my_iir = IIR2Filter(sos[0, :3], sos[0, 3:])
    my_factory = IIRFilter(sos)

    out = [my_factory.filter(x) for x in data]
    scalar = np.mean(out[1000:-1000]) / np.mean(data[1000:-1000])
    out_mod = [x/scalar for x in out]
    filtered = signal.sosfilt(sos, data)


    plt.plot(out_mod, label=f'freq = {freq}')
    plt.legend()

    plt.plot(data, label='row')
    plt.legend()
    print('end')
```