# Assignment 3, Digital Signal Processing: IIR Filters
# University of Glasgow
# School of Engineering

Mustafa Biyikli & Katarzyna Lenard

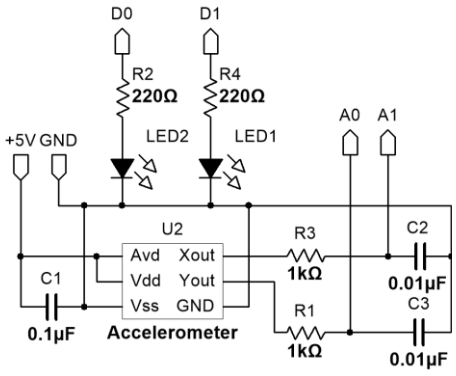
## Introduction

Causal signal processing is a must for applications that require real-time filtering of a signal. Finite impulse response (FIR) filters provide a linear phase response, however, they introduce large delays. Infinite impulse response (IIR) filters provide a non-linear phase response, however, the group delay may be as low as a single sample. Therefore, IIR filters prove very useful for systems that require fast and efficient signal processing. An application of such a system would be deployment of airbags during a car crash where decisions must be made within 50ms after the initial contact with the crash opponent (Weinert et al., 2018). Unfiltered, capacitive accelerometers are highly susceptible to DC noise (Dong et al., 2006), which makes them unreliable for thresholding. The aim of this study was to implement a Direct Form II IIR filter using Python. The objective was to filter the output of an analog capacitive accelerometer, MMA3201KEG, in real-time, for reliable impact detections.

## Methods

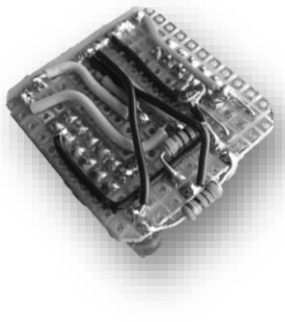
### Data Acquisition and Setup

An analog circuit was designed and built using the MMA3201KEG accelerometer chip. Figure 1 (a) shows the schematic of the analog circuit. A 0.1µF capacitor was used to decouple the power source. A low-pass filter of 1kΩ and 0.01µF was used on the accelerometer outputs to minimise the clock noise from the switched capacitor filter circuit. Red and yellow LEDs were added to provide visual feedback on the detected impact. I/O pins were used for connecting the module to a microcontroller. Figure 1 (b) and (c) show the implemented circuit.




(a) Schematic of the analog circuit

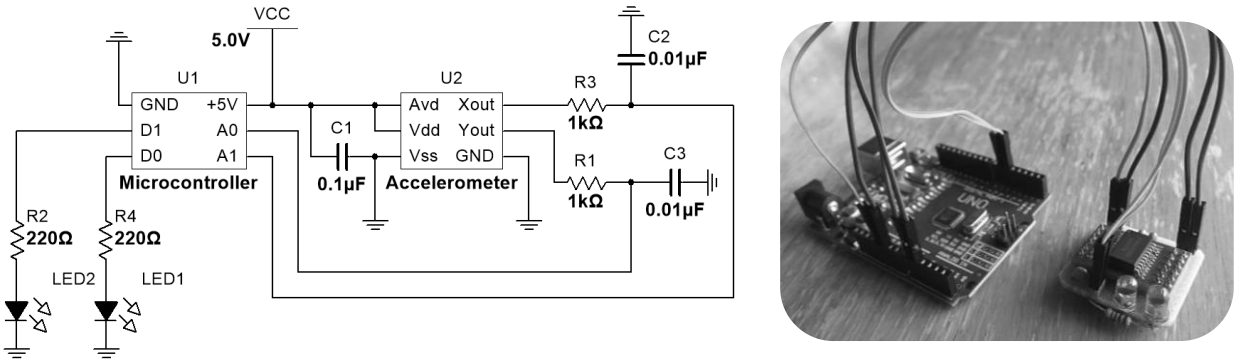
(b) Top View



(c) Bottom View

Figure 1, Schematic & implementation of the analog circuit.

An Arduino UNO was used as the microcontroller. Programming was done using Python and the sampling rate, Fs, was fixed at 100Hz using the Python library, pyfrimata2. This allowed for an accurate representation of the frequency response. Figure 2 (a) and (b) show the schematic and the implementation of the circuit used to record the acceleration data.

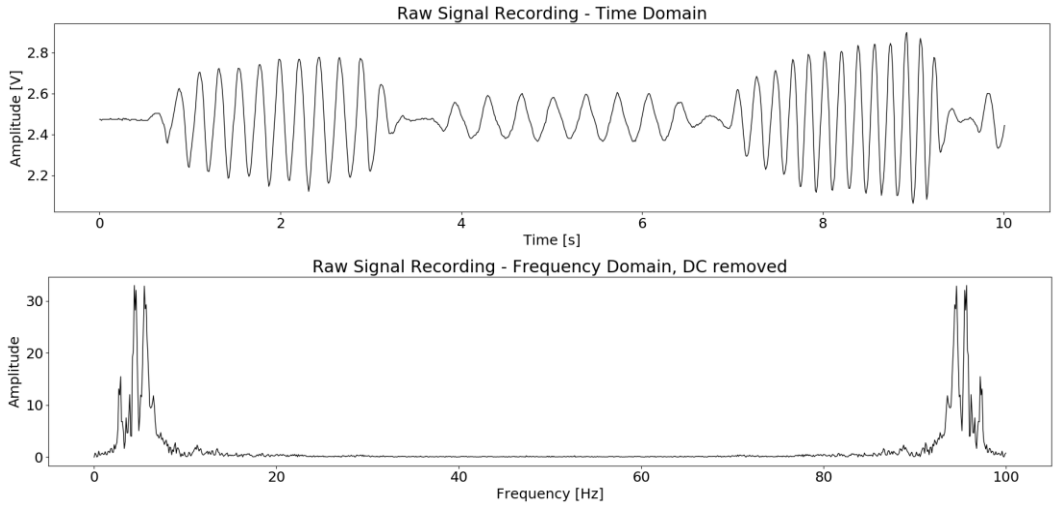



(a) Schematic of the circuit (b) Implementation

Figure 2, Schematic & implementation of the data acquisition circuit.

## Filter Responses & SOS Coefficients

In order to find out which frequencies needed to be filtered, the frequency spectrum of low and high acceleration movements were analysed. The movements were recorded for 10 seconds and then the fast Fourier transform (FFT) was applied, producing 2 consecutive plots. The top plot shows the raw accelerometer signal and the bottom one shows its frequency response, excluding DC.




(a)

(b)

Figure 3, Low acceleration movements with corresponding frequency spectrum.

Only the y-axis output of the accelerometer was investigated as the response of both axes were similar. First, low acceleration movements were performed by moving the circuit board along the y-axis. As it can be seen in Figure 3, all frequencies fell below 20 Hz, irrespectively of how quickly the direction was changed. Then, the impact was simulated by hitting the circuit board on its side, and the Fourier transform of the movement was plotted. This is shown in Figure 4.

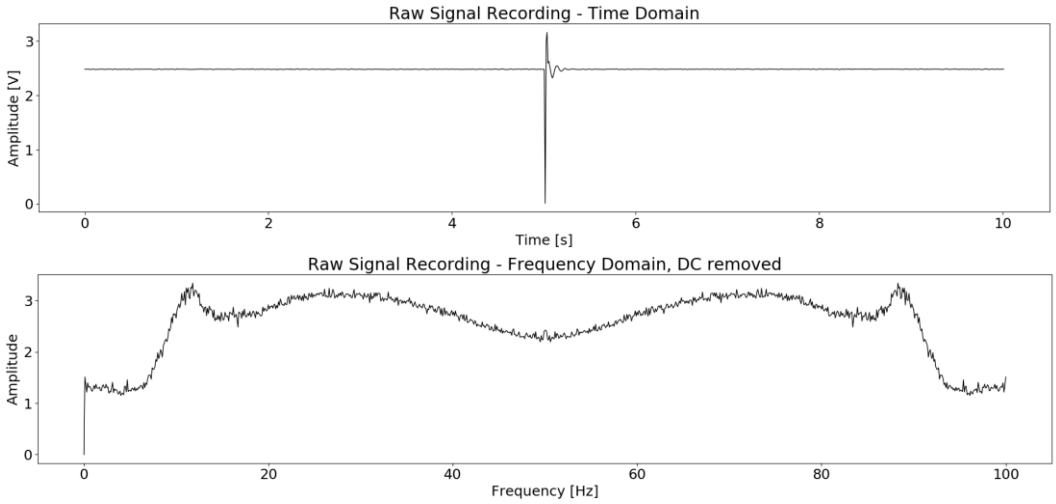



Figure 4, High acceleration movement (impact) with corresponding frequency spectrum.

From Figure 4, it can be seen that all frequencies are mapped throughout the spectrum. This showed aliasing of the signal, which was expected at a sampling rate of 100Hz as the MMA3201KEG bandwidth response was 400Hz. However, the impacts could still be determined by excluding the previously identified frequencies shown in Figure 3. Therefore, a high pass filter with 20Hz cut-off frequency was chosen with an intention to filter the low acceleration movements out and remove the DC noise.

The coefficients of an IIR filter are based on the numerator (zeros) and the denominator (poles) of the filter transfer function, $H(z)$, based on the desired filter response. Equation 1 shows the canonical form of the second order filter transfer function.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (1)$$

Where, $b_0$, $b_1$ and $b_2$ are the FIR coefficients, $a_1$ and $a_2$ are the IIR coefficients and $z^{-1}$, $z^{-2}$ represent the delay.

The coefficients of the transfer function can be derived analytically, however, in this case, they were obtained using a high-level Python function *scipy.signal.butter(),* implementing a Butterworth filter.

### Real-Time Filtering with IIR

High order structures can be created by cascading 2nd order IIR filters, where each IIR filter requires its own coefficients. Therefore, the coefficients for each IIR filter were created and fed into the filter in an order. Figure 5 shows an example of how they cascade.

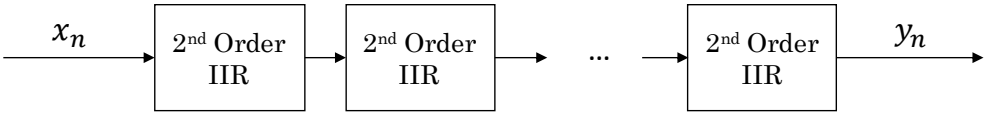



Figure 5, A chain of 2nd order IIR filters.

The Butterworth function generates second-order section (SOS) coefficients, which depend on the given order and the cut-off frequency. A higher order provides a better filtering performance, but at the same time, causes a larger delay and may make the filter unstable. The order chosen for this project was 6. The decision was based on trial and error. 6th order Butterworth analogue filter produced 3 sets of coefficients for 3 consecutive 2nd order IIR

digital filters. If the order number was odd, it would create the last set of coefficients for a 1st order IIR filter, hence, only even numbers were taken under consideration.

In this study, a 2nd order IIR filter was built based on the Direct Form II topology and optimised to accommodate any number of SOS coefficients. The dataflow diagram of this topology is shown in Figure 6.

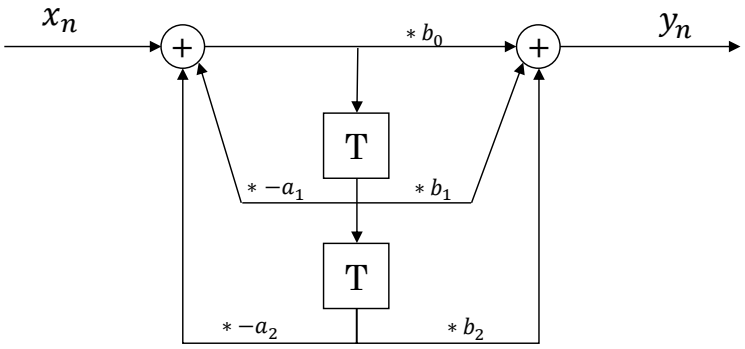



Figure 6, Direct Form II of 2nd order IIR Filter.

Where $x_n$ and $y_n$ are input and output data, respectively, $a_1$ and $a_2$ are the IIR coefficients and $b_0$, $b_1$ and $b_2$ are the FIR coefficients. T represents a delay block.

Following the creation of the IIR filter, the raw and filtered data from both output axes were plotted and visualised in real-time. A screenshot of the real-time visualisation are shown in Figure 7.

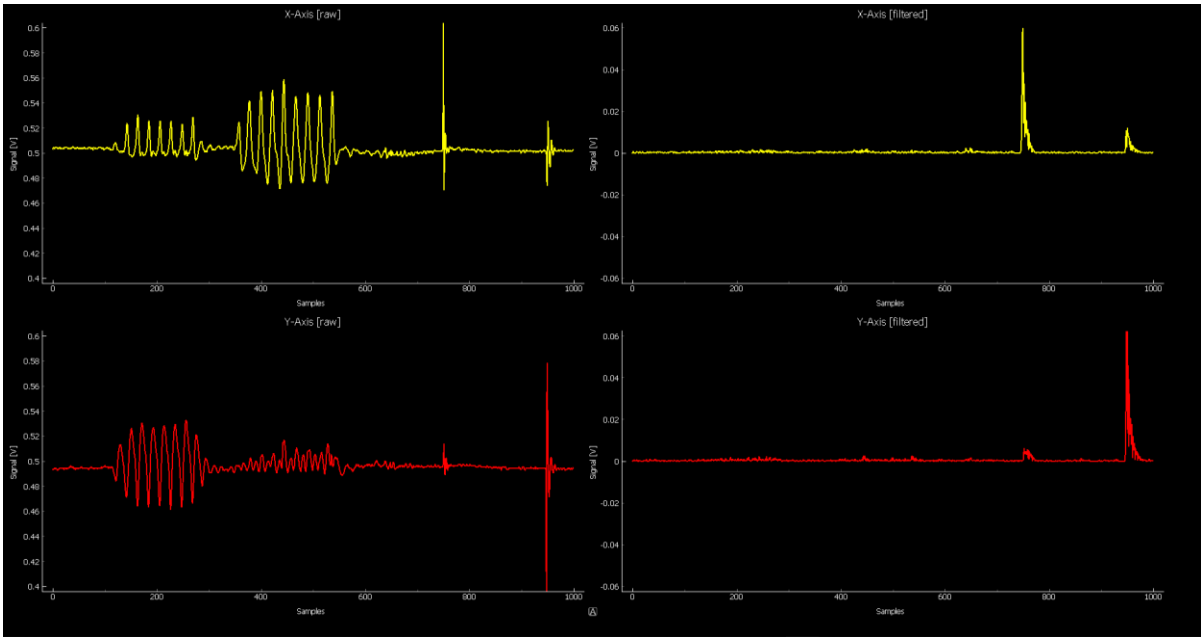



Figure 7, A screenshot of real-time filter.

Plots on the left represent the raw signal outputs of the X-axis (top) and Y-axis (bottom) of the accelerometer. Plots on the right are the filtered responses of the corresponding raw output signals. Figure 7 shows the low acceleration movements followed by an impact (short, strong, high acceleration movement) in each of the axes. As it can be seen, the low acceleration movements, irrespectively of how quickly they are performed, are filtered out,

as they fall below the 20Hz point on the frequency spectrum. Where high acceleration movements are detected by the IIR filter.

In order to indicate the hypothetical airbag deployment, red and yellow LED lights were activated, simulating frontal and lateral impacts, respectively. This was achieved by applying a threshold value. For testing and video purposes the threshold value was set to 35mV, on the filtered signals. Figure 8 shows a block diagram of the system.

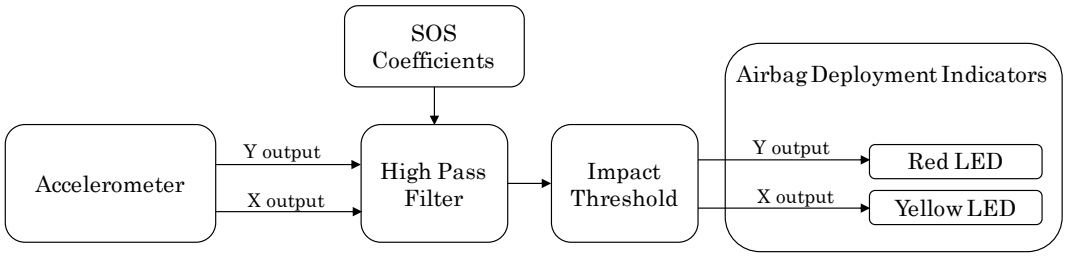



Figure 8, Block diagram of the system.

## Discussion

As it can be seen from Figure 7, the IIR filter proved to be successful. The low acceleration movements falling below 20Hz in the frequency spectrum of the accelerometer were filtered out. Impact accelerations within the IIR filter frequency range were detected. Further investigation would be required in order to obtain the correct threshold values for airbag deployment in a car crash. MMA3201KEG, analog accelerometer chip produces a linear voltage output within a maximum range of ±40g, where, g, represents an acceleration equating to 9.81m/s$^2$. Therefore, it can be used for crash detection, however, with a characteristic bandwidth response of 400Hz, the output sampling rate should be much higher than 100Hz. This study was constrained to sampling data at 100Hz, therefore, a true representation of the frequency spectrum could not be obtained. As it can be seen in Figure 4, there was aliasing of the signal as the maximum frequency was above the Nyquist frequency (Fs/2). However, for the purposes of this study, using a high-pass IIR filter allowed for detecting the impacts and filtering out the oscillations.

Although it has the widest transition width, Butterworth filter was chosen as it has no ripples in both pass and stopbands. Using a higher sampling rate, the differences between the IIR filter types may be investigated and the impact detection performance can be improved.

Compared to an FIR filter, the response of an IIR filter was immediate. An FIR filter has a sample-delay equal to half the number of taps. An IIR filter has a group delay, which means that it varies for different frequencies up until the Nyquist frequency, Fs/2. The group delay of our 6$^{th}$ order IIR filter is shown in Figure 9.

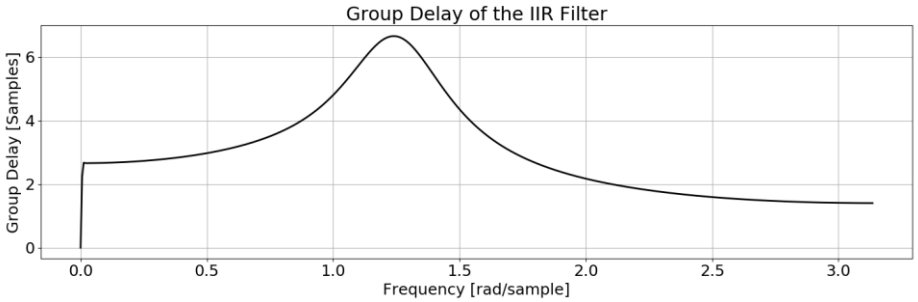



Figure 9, Group delay of the 6th order Butterworth IIR filter

Therefore, the impact frequencies close to the Nyquist frequency required around 2 samples of delay to be processed. At a sampling rate of 100Hz, this equates to 20ms of delay, proving the response of the filter quick enough for detecting a crash and deploying an airbag within the previously stated 50ms.

## Conclusions

This study focuses on causal signal processing for detecting an impact. An analog accelerometer chip, MMA3201KEG, was used to design and implement an analog circuit. The output signal was recorded in 2 separate channels. The frequency spectrum of the signal was analysed and a high-pass filter with a cut-off frequency at 20Hz was justified as suitable for impact detection. The SOS coefficients were generated using a $6^{th}$ order Butterworth filter and fed into a cascade of $2^{nd}$ order IIR filters implemented in Python. The filter successfully removed DC and low acceleration movements while detecting the impacts.

Additionally, a threshold was applied on both axes for activating LEDs to simulate the impact detection. A YouTube clip demonstrating the IIR filter can be found at, https://www.youtube.com/watch?v=uSm0Zic1wVA.

Such a system can easily be implemented for deploying a safety mechanism during a car crash, such as the frontal and lateral airbags.

## References

Dong, P., Li, X., Wang, Y., Feng, S. & Li, S. An Axial-beam Piezoresistive Accelerometer for High-performance Crash Detection of Automotive Industry. 2006 2006. IEEE, 1481-1484.

Weinert, F., Leschke, A. & Bonaiuto, V. 2018. INNOVATIVE APPROACH TO CRASH DETECTION IN PASSENGER CARS. *International Journal of Automotive Science And Technology*, 1-8.

## Appendix A – IIR Classes

```
class IIR2Filter:
    def __init__(self, sos):
         # FIR Coefficients
        self.b0 = sos[0]; self.b1 = sos[1]; self.b2 = sos[2]
        # IIR Coefficients
        self.a0 = sos[3]; self.a1 = sos[4]; self.a2 = sos[5]
        self.buffer1 = 0
        self.buffer2 = 0

    def dofilter(self, x):
        acc_input = x - (self.buffer1*self.a1) - (self.buffer2*self.a2)
        acc_output
= (acc_input*self.b0) + (self.buffer1*self.b1) + (self.buffer2*self.b2)

        self.buffer2 = self.buffer1
        self.buffer1 = acc_input
        return acc_output

class IIRFilter:
    def __init__(self, sos):
        self.sos = sos
        self.order = len(sos)
        self.slave = []
        for n in range(self.order):
            self.slave.append(IIR2Filter(self.sos[n,:]))
    def dofilter(self, x):
        self.y = x
        for i in range(len(self.sos)):
            self.y = self.slave[i].dofilter(self.y)
        return self.y
```

## Appendix B – Real Time IIR Main

```
import sys
import pyqtgraph as pg
from pyqtgraph.Qt import QtCore, QtGui
import numpy as np; import scipy.signal as signal
from pyfirmata2 import Arduino

import IIR2Filter as iir

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

# LED pins
yellow_LED = 5; red_LED = 6
```

```
# Avtivation Threshold [V] - for filtered signal
THRESHOLD = 0.035

# create a global QT application object
app = QtGui.QApplication(sys.argv)

# signals to all threads in endless loops
running = True
pg.setConfigOption('background', 'k')
pg.setConfigOption('foreground', 'w')

class AddPlot:
    '''
    Feed the class with:
        linecolor - i.e: 'k', 'w', 'b', 'r', 'g', 'y'\n
        title - i.e: 'AmazingPlot'\n
        xlabel & ylabel in string format\n
        position in list format - i.e: [2, 1] for 2nd row 1st column\n
        yrange & xrange in list format\n
    '''
    win = pg.GraphicsWindow()
    win.setWindowTitle("IIR Filter")

    def __init__(self, linecolor, title, xlabel, ylabel, position, yrange,
xrange=[0,600]):
        self.plt = self.win.addPlot(*position)
        self.plt.setTitle(title)
        self.plt.setLabel('bottom', xlabel); self.plt.setLabel('left',
ylabel)
        self.plt.setFixedWidth(self.win.width()/2.1); self.plt.setFixedHeig
ht(self.win.height()/2.1)
        self.plt.setYRange(*yrange); self.plt.setXRange(*xrange)
        self.curve = self.plt.plot(pen=pg.mkPen(linecolor, width=2))
        self.data = []
        self.datalength = xrange[1] - xrange[0]
        self.red_LED_timer = []; self.yellow_LED_timer = []
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.update)
        self.timer.start(100)
        self.win.show()

    def update(self):
        self.data=self.data[-self.datalength:]
        self.plt.setFixedWidth(self.win.width()/2.1); self.plt.setFixedHeig
ht(self.win.height()/2.1)
        if self.data:
            self.curve.setData(np.hstack(self.data))

        if board.digital[red_LED].read() == 1:
```

```
            self.red_LED_timer.append(self.data)
            if len(self.red_LED_timer) == 30:
                board.digital[red_LED].write(0)
        elif board.digital[red_LED].read() == 0:
            self.red_LED_timer.clear()

        if board.digital[yellow_LED].read() == 1:
            self.yellow_LED_timer.append(self.data)
            if len(self.yellow_LED_timer) == 30:
                board.digital[yellow_LED].write(0)
        elif board.digital[yellow_LED].read() == 0:
            self.yellow_LED_timer.clear()

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

# Let's create two instances of plot windows
Plot1 = AddPlot('y', 'X-Axis [raw]', 'Samples', 'Signal
[V]', [1, 1], [0.4, 0.6])
Plot2 = AddPlot('r', 'Y-Axis [raw]', 'Samples', 'Signal
[V]', [2, 1], [0.4, 0.6])
Plot3 = AddPlot('y', 'X-Axis [filtered]', 'Samples', 'Signal
[V]', [1, 2], [-0.06, 0.06])
Plot4 = AddPlot('r', 'Y-Axis [filtered]', 'Samples', 'Signal
[V]', [2, 2], [-0.06, 0.06])

# sampling rate: 100Hz
Fs = 100
# Cut off frequency [Hz]
cut_off = 20

# Creation of the coeffs for high-pass
f1 = cut_off/Fs * 2
sos = signal.butter(6, [f1], 'high', output='sos')

# Feed the sos coefficients into the FIR filter for each axis
IIR_X = iir.IIRFilter(sos)
IIR_Y = iir.IIRFilter(sos)

# called for every new sample which has arrived from the Arduino
def callBack(data):
    # send the sample to the plotwindow
    ch0 = data
    ch0_filtered = abs(IIR_X.dofilter(ch0))
    Plot1.addData(ch0)
    Plot3.addData(ch0_filtered)
    if ch0_filtered > THRESHOLD:
        board.digital[yellow_LED].write(1)

    ch1 = board.analog[1].read()
```

```
    if ch1:
        Plot2.addData(ch1)
        ch1_filtered = abs(IIR_Y.dofilter(ch1))
        Plot4.addData(ch1_filtered)
        if ch1_filtered > THRESHOLD:
            board.digital[red_LED].write(1)

# 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()

# showing all the windows
app.exec_()

# needs to be called to close the serial port
board.exit()
print("Finished")
```