ENG5027: Digital Signal Processing

SCHOOL OF ENGINEERING

# Assignment 3, Digital Signal processing: IIR Filters

| **Author** | **GUI** |
|---|---|
| **Anton Saikia** | 2426828S |
| **Gabriel Galeote Checa** | 2413232C |

UNIVERSITY OF GLASGOW

GLASGOW, DECEMBER 2019

# Contents

## 1 Objectives

The aim of this assignment was to design an electronic system for a certain application and perform real-time digital signal processing by using IIR filters. In this project, the topic chosen was ”detection of heartbeat by using photophlethysmography”.

The Heart Rate of a person is an indicator of stress and the general health condition of an individual. For instance, through parameters such as RR-intervals, which is the time between two R-waves in an electrocardiogram (ECG), it is possible to derive some interpretations about the state of one’s heart healthiness. An irregular RR-interval or unexpected peaks within that interval could be indicate anomalies in heart functioning, such as arrhythmia or other conditions. The heartbeat sensor system designed in this project is based on the principle of photophlethysmography, where the change in blood volume in any part of the body is detected by measuring the change in the intensity of light passing through that body part. The flow of blood volume is decided by the rate of heart pulses and since light is absorbed by blood, the signal pulses are equivalent to the heart beat pulses.

Typically there are two methods of doing this: *Transmission*, where light emitted from the light source is transmitted through any vascular region of the body like earlobe or finger and received by the detector. *Reflection*, where light emitted from the light source is reflected by the regions and received with a certain angle. The system consists of a light emitting diode, typically infrared or green, and a photoresistor/LDR that measures the variations of the light intensity through the tissue. When a heartbeat occurs, the tissue blood pressure changes causing a change in light intensity sensed by the photoresistor. Using this variation, the signal is amplified and sent to an ADC and to a microcontroller, computer or data acquisition board.

## 2 Methodology

### 2.1 Design of the circuit

For the design and fabrication of the circuit of this project, a conventional Light-Dependant Resistor (LDR) or photoresistor is chosen as the sensor and placed in a voltage divider to move the offset and measure the variations in the resistance of the sensor. The offset value of the sensor is 2 $K\Omega$, so another 2 $K\Omega$ resistor is set in a voltage divider. To measure the heartbeat from a finger, a light source such as an LED is placed right above the fingernail, while the finger tip is kept on the LDR surface, as shown in Fig. 2 in. The output of the sensor is given to an operational amplifier in a non-inverting configuration, amplifying the sensor signal by 100 times. The amplifier output is then connected to an analogue input pin of the Arduino. The schematic of the designed circuit is shown in Fig. 3.

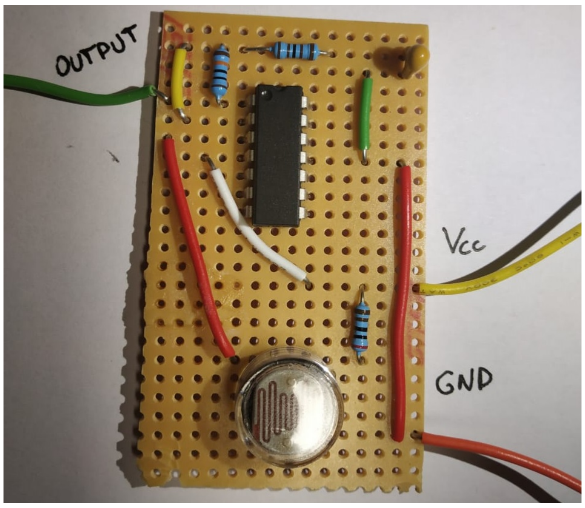

Figure 1: Designed sensor system

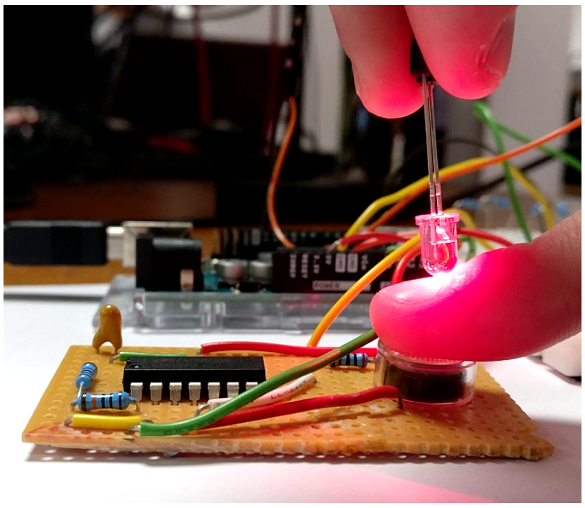
Figure 2: Experimental setup

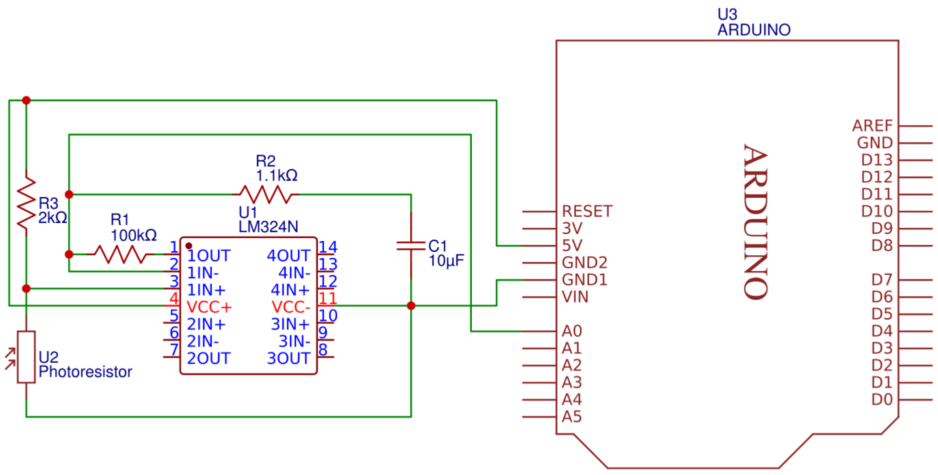

Figure 3: Schematic of the circuit for the device.

## 2.2 Filter Design

The first step in any signal processing application is to analyse the time and frequency behaviour of the recorded signal. In this case, the target signal is measuring the blood volume variations in any finger as a result of heartbeats. In other techniques like electrocardiography, the electrical signal produced in the heart as a results of the biochemical process occurring in the heart during the heartbeat (Guyton and Hall, 1992; Tortora and Derrickson, 2008; Fox, 2006). The heartbeat does not only produces an electrical change in the body but also a change in blood volume in the different parts of the body which is more measurable in areas with more vascularization than in others (Shelley et al., 2001). For example, while volume variations in areas like in the toe are very little, in the earlobe this variation is easily measurable as there is no bone and the tissue is very vascular. Although that area would be ideal, the conventional approach is to measure the blood volume variations in the finger due to easy accessibility and easy fabrication of the device.

The typical heart rate for an adult male is around 60 beats per minute (bpm), which means that there is a beat every second. Thus, it would be expected that have a harmonic in frequency of the target signal at around 1 Hz. Assuming that the maximum frequency of a person could be 220, the maximum deviation of the fundamental frequency is up to $220/60 = 3.66Hz$. However, the heartbeat wave detected on the sensor has the always the same characteristics and the aim is to isolate it for a further peak detection if necessary. Therefore, it can be assumed that every frequency above that range is either harmonics or noise. Given the limited sampling frequency at 100 Hz of an Arduino and a Nyquist frequency of 50 Hz, the main harmonics of the 1 Hz peak are within the range from 1 to 10 Hz. For that reason, all the frequencies above 10 Hz were considered as non-meaningful frequencies for the analysis and consequently filtered out.

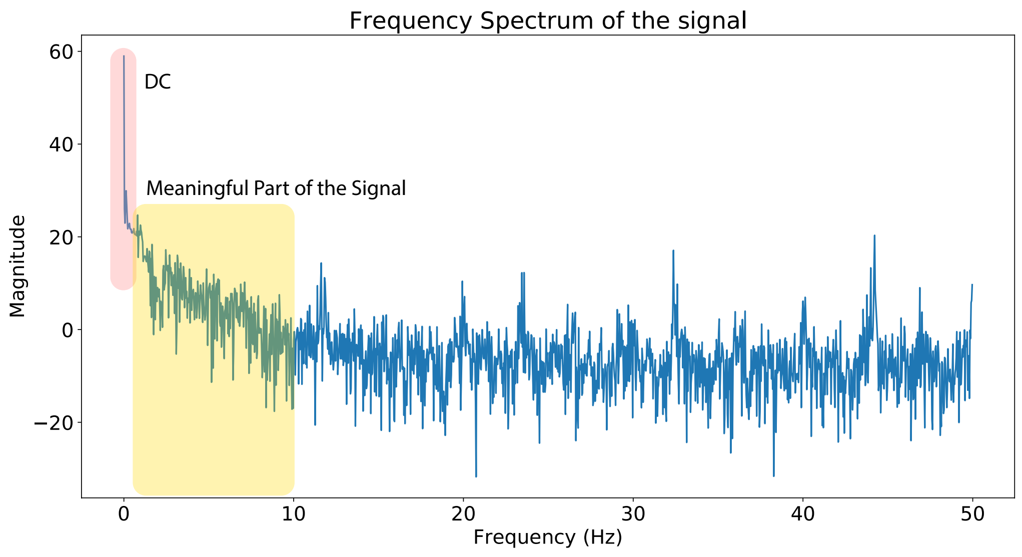

Figure 4: Spectrum in frequency domain of the pre-recorded signal

A sample signal was recorded for analysis and filter design. The spectrum in frequency domain of that signal is the shown in Fig. 4. It can be seen in the figure, that there is a big component of DC in the signal which needs to be removed. The peaks at frequencies higher than 10 Hz could be due to the environment and erroneous readings. The next step was to decide which filter type and cutoff frequencies to use.

As mentioned before, in order to remove the DC component and attenuate the frequencies above 10 Hz a band pass filter is needed to be implemented. However, the filter was designed to be even more restricted to 4 Hz because that is where the main energy of the peaks will be found. A band pass filter is essentially a high pass filter combined with a low pass Filter. The lower cutoff frequency will remove the DC component while the upper cutoff frequency will attenuate all frequencies above 4 Hz. After thorough analysis the cutoff were chosen at 0.8 Hz and 4 Hz.

Three filter types were studied as they were the most commonly used types for IIR filter applications: Butterworth (butter), Chebyshev1 (cheby1) and Chebyshev2 (Cheby2). The frequency responses of the designed Butter, cheby1 and cheby2 filters are shown in Figs. 5, 6 and 7 respectively. The order of the filter chosen in all the cases was 2 because in realtime applications the filter order, affects the processing time and as it adds a delay due to the computational cost. Thus, the efficiency of the filter must be taken into consideration. The frequency response for a second order filter was adequate for this application.

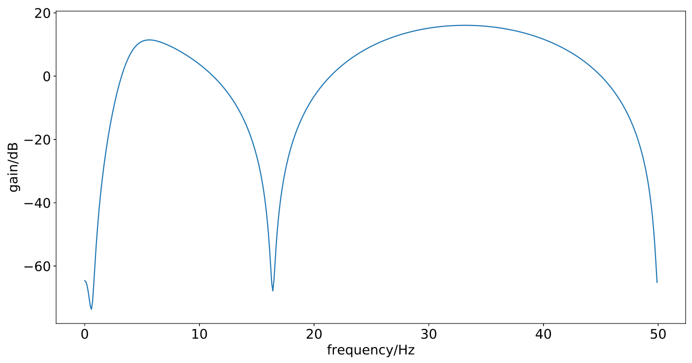

Figure 5: Butterworth filter frequency response.

The outputs of the filtered signal for each of the filters given before are very similar, there were only slight differences on the ripples and amplitude of the main peaks of the signal. Due to design simplicity the Butterworth filter was chosen as the filter type for implementation. The time domain and frequency response spectrum of the recorded signal before and after using the designed Butterworth filter is shown in Figs. 8 and 9, respectively.

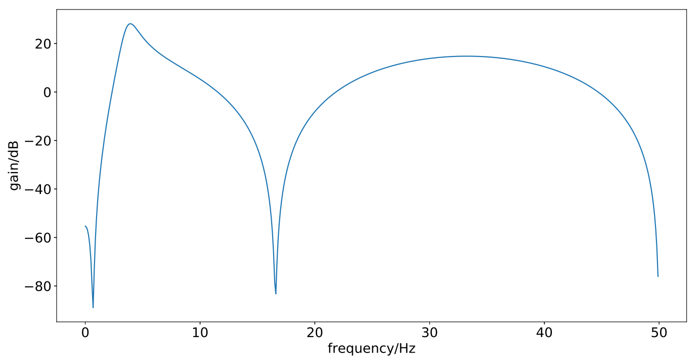

Figure 6: Chebyshev1 filter frequency response with a maximum ripple allowed below unity gain in the passband of 5 dB.

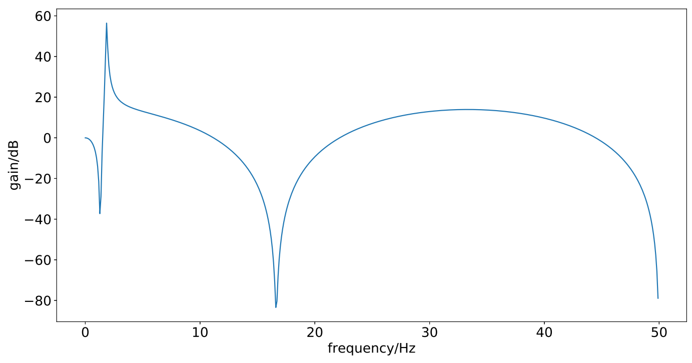

Figure 7: Chebyshev2 filter frequency response with a minimum attenuation in the stop band of 60 dB.

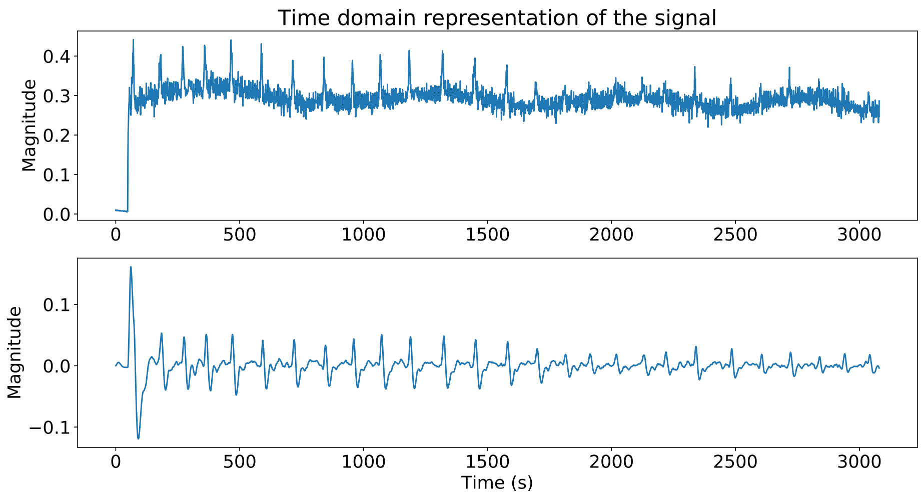

Figure 8: Time Domain Representation of the signal unfiltered and filtered from up to down.

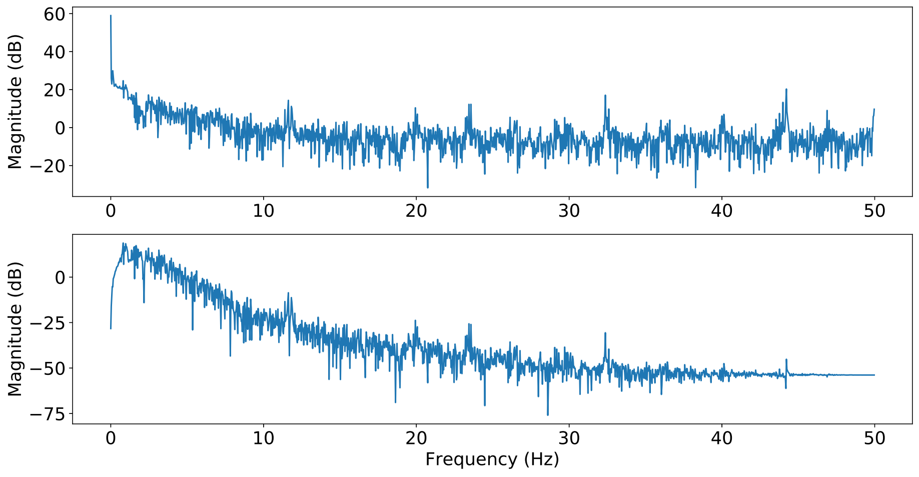

Figure 9: Frequency Spectrum of the signal unfiltered and filtered from up to down.

## 2.3 Processing of the Signal

Once the signal was analysed, the filter was designed and the circuit was assembled, the code had to be be programmed in order to establish communication between the python code and the Arduino board by using *pyFirmata*. Then, the sensor is read by the Arduino and the reading is sent to the computer where the processing of the signal is done. The data flow diagram of the code is shown in Fig. 10. Here, all the steps are summarised from the initialisation of the script to the plotting of the results.

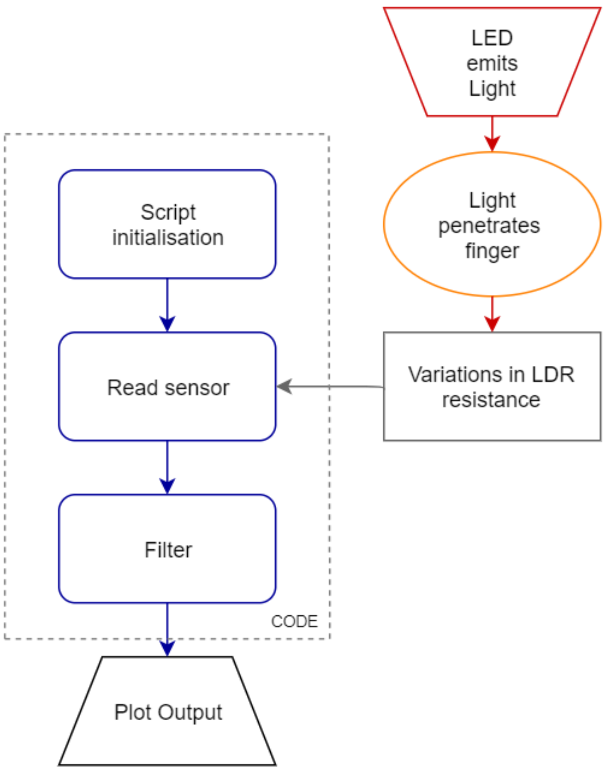

Figure 10: Data flow diagram

## 3 Results and Discussion

The output of the processing shows bigger peaks corresponding to the heartbeats of the recorded signal. The aim of this project was to filter and isolate the peaks of the recorded signal detected by the LDR which meant to make those peaks bigger and more evident while removing the noise. This was clearly achieved as shown in Fig. 11. In that figure it is shown the plot of the realtime processing and as it can be seen, the filtered signal has significantly more pronounced peaks and is visibly less noisy than the original signal. In this figure it can also be seen that in the bottom signal which had a higher heartrate obtained by performing some exercise, has time intervals shorter than the signal with the normal heartbeat.

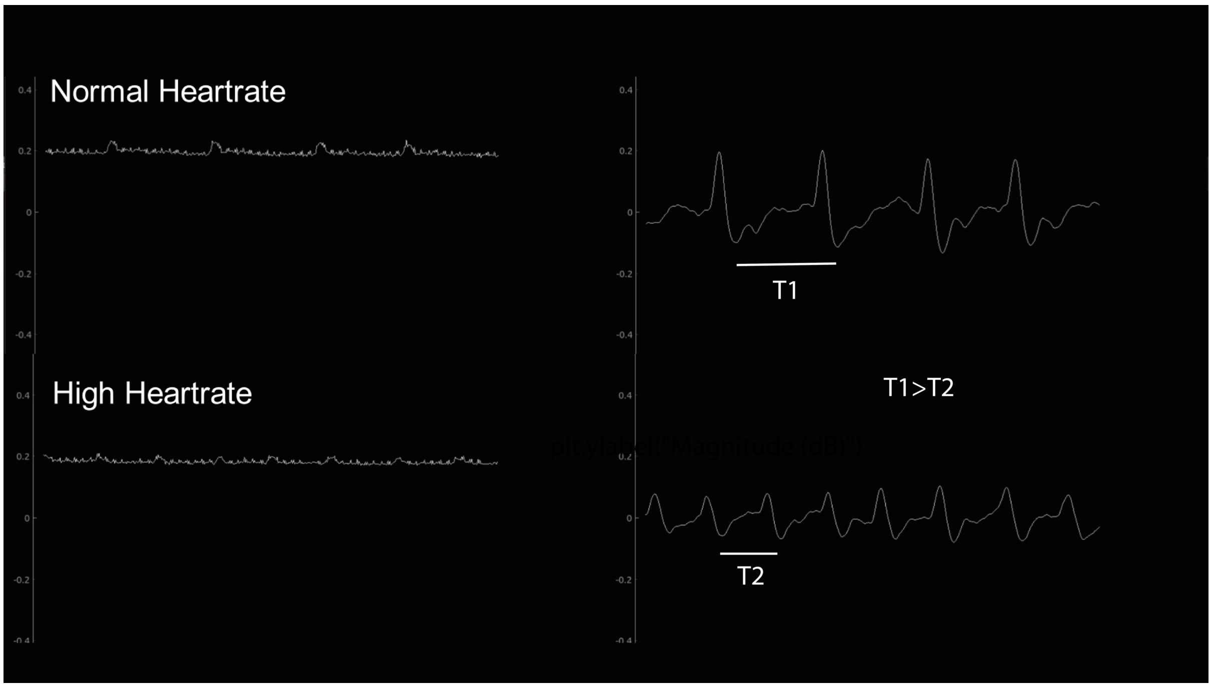

Figure 11: Realtime filtering plot

A slight delay is visible in the filtered signal. This delay can be attributed to the computational cost of the designed IIR filter. The computational cost is directly proportional to the order of the IIR filter so, increasing the order of IIR filters to the chain will result in a greater computational cost and consequently, more delay. It was also observed that the amplitude of the output is very sensitive to the incident light on the finger as well as the pressure applied by the finger on the LDR surface.

Link to github repository: https://github.com/GGChe/DigitalSignalProcessing

## 4 Bibliography

Fox, S. I. (2006). *Human Physiology 9th Editon.* McGraw-Hill press, New York, USA.

Guyton, A. C. and Hall, J. E. (1992). Human physiology and mechanisms of disease.

Shelley, K., Shelley, S., and Lake, C. (2001). Pulse oximeter waveform: photoelectric plethysmography. *Clinical monitoring*, pages 420–428.

Tortora, G. J. and Derrickson, B. H. (2008). *Principles of anatomy and physiology.* John Wiley & Sons.

# 5 Python Code

## 5.1 Filter

```
# !/usr/bin/python3
"""
Authors: Gabriel Galeote-Checa & Anton Saikia

Documentation:

This script is for the digital signal processing of a light-based pulsemeter
    device consisting of a photoresistor and an LED.
A finger is placed between the LED and the photoresistor and by light
    attenuation due to different blood pressures
related to the heartbeat, we can detect a periodic signal.

The process of the script is simple:
Read sensor -> Filter -> Plot

For the filtering of the signal, an IIR filter was implemented in the class
    IIR:

- IIR2Filter(coefficients) --> Creates a 2nd order IIR filter from a given
    coefficients.

- IIRFilter(coefficients) --> Creates an IIR filter of any order as a chain
    of 2nd order IIR filters using the
                        class IIR2filter.
"""

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

"""
# ------------- Initialization of the Script -------------

    - fs : sampling frequency, defined for every application and limited by
        the system specifications.
    - PORT : communication port, detected automatically so we don't have to
```

```
        care about the specific COM port.
    - app : global QT application object for plotting.
    - running = signals to all threads in endless loops that we'd like to run
        these

"""

fs = 100
PORT = Arduino.AUTODETECT
app = QtGui.QApplication(sys.argv)
running = True
board = Arduino(PORT)   # Get the Arduino board.
board.samplingOn(1000 / fs) # Set the sampling rate in the Arduino

class IIR2Filter(object):
    """
    given a set of coefficients for the IIR filter this class creates an
        object that keeps the variables needed for the
    IIR filtering as well as creating the function "filter(x)" with filters
        the input data.

    Attributes:
        @:param coefficients: input coefficients of the IIR filter as an array
            of 6 elements where the first three
        coefficients are for the FIR filter part of the IIR filter and the
            last three coefficients are for the IIR part
        of the IIR filter.
    """

    def __init__(self, coefficients):
        self.IIRcoeff = self.myCoefficients[3:6]
        self.FIRcoeff = self.myCoefficients[0:3]
        self.myCoefficients = coefficients
        self.acc_input = 0
        self.acc_output = 0
        self.buffer1 = 0
        self.buffer2 = 0
        self.input = 0
        self.output = 0

    def filter(self, input):
        """
        :param input: input value to be processed.
        :return: processed value.
        """

        self.input = input
        self.output = 0

        """
        IIR Part of the filter:
            The accumulated input are the values of the IIR coefficients
                multiplied by the variables of the filter:
            the input and the delay lines.
        """
        self.acc_input = (self.input + self.buffer1
                          * -self.IIRcoeff[1] + self.buffer2 * -self.IIRcoeff[2])

        """
        FIR Part of the filter:
            The accumulated output are the values of the FIR coefficients
                multiplied by the variables of the filter:
            the input and the delay lines.

        """

        self.acc_output = (self.acc_input * self.FIRcoeff[0]
                           + self.buffer1 * self.FIRcoeff[1] + self.buffer2
                           * self.FIRcoeff[2])

        # Shifting the values on the delay line: acc_input->buffer1->buffer2
        self.buffer2 = self.buffer1
        self.buffer1 = self.acc_input
        self.input = self.acc_output
        self.output = self.acc_output
        return self.output


class IIRFilter(object):
    """
    given a set of coefficients for the IIR filter this class creates an
        object that keeps the variables needed for the
    IIR filtering as well as creating the function "filter(x)" with filters
        the input data.

    Attributes:
        @:param coefficients: input coefficients of the IIR filter as an array
            of n arrays where n is the order of the
        filter. The array of coefficients is organised in blocks of 6
            coefficients where the first three coefficients
        are for the FIR filter part of the IIR filter and the last three
            coefficients are for the IIR part of the IIR
        filter.
    """

    def __init__(self, mycoeff):
        self.myCoefficients = mycoeff
        self.acc_input = np.zeros(len(self.myCoefficients))
        self.acc_output = np.zeros(len(self.myCoefficients))
        self.buffer1 = np.zeros(len(self.myCoefficients))
        self.buffer2 = np.zeros(len(self.myCoefficients))
        self.input = 0
        self.output = 0
        self.myIIRs = []

        """
        An IIR filter can be calculated as a chain of 2nd order IIR filters.
            For that, the array myIIRs contains
        a list of IIR2Filter classes initialised with the coefficients given.
        """
        for i in range(len(self.myCoefficients)):
            self.myIIRs.append(IIR2Filter(self.myCoefficients[i]))

    def filter(self, input):
        """
        Filter and input value with the IIR filter structure.
        :param input: input value from the reading in real time to be
            processed.
        :return: processed value.
        """

        self.input = input
        self.output = 0
        self.output = self.myIIRs[0].filter(input)
        for i in range(1, len(self.myCoefficients)):
            self.output = self.myIIRs[i].filter(self.output)
        return self.output


class QtPanningPlot:

    def __init__(self, title):
        self.win = pg.GraphicsLayoutWidget()
        self.win.setWindowTitle(title)
        self.plt = self.win.addPlot()
        self.plt.setYRange(-1, 1)
        self.plt.setXRange(0, 600)
        self.curve = self.plt.plot()
        self.data = []
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.update)
        self.timer.start(100)
        self.layout = QtGui.QGridLayout()
        self.win.setLayout(self.layout)
        self.win.show()

    def update(self):
        self.data = self.data[-500:]
        if self.data:
            self.curve.setData(np.hstack(self.data))

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


"""

|  -------------------------- MAIN   -------------------------- |
|  Cutoff frequencies:                                           |
|         a) wc1 = 0.8 Hz to remove DC components                |
|         b) wc2 = 4 Hz because the maximum heartrate is           |
|              220 bpm = 220/60 = 3.67 Hz                        |
|                                                                |
| Order of the filter:                                           |
|          n = 2 for a chain of two 2nd order IIR filter as we   |
|           are using 'sos' for second-order sections            |
|                                                                |
-----------------------------------------------------------------
"""

cutoff = [0.8, 4]
order = 2

for i in range(len(cutoff)):
    cutoff[i] = cutoff[i] / fs * 2
```

```
coeff = signal.butter(order, cutoff, 'bandpass', output='sos')

# If the order of the filter is 1, is one IIR 2nd order filter otherwise, it
    is a chain of IIR filters.
if order > 1:
    myFilter = IIRFilter(coeff)
else:
    myFilter = IIR2Filter(coeff[0])

# Create two instances of Qt plots
qtPlot1 = QtPanningPlot("Arduino 1st channel")
qtPlot2 = QtPanningPlot("Arduino 2nd channel")

# This function is called for every new sample which has arrived from the
    Arduino
def callBack(data):
    qtPlot1.addData(data)
    ch1 = board.analog[1].read()
    if ch1:
        filteredData = myFilter.filter(ch1)
        qtPlot2.addData(filteredData * 10)


# 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 all windows
app.exec_()

# Close the serial port
board.exit()

print("Execution Finished")
```