ENG5027: Digital Signal Processing - Assignment 2
FIR Filters

Zeynep Oner - 2693256O
Vishwa Poswal - 2699522P
Zainab Meerza - 2361575M

University of Glasgow

QUESTION 1

In the first question, the aim was to create functions, in Python, to calculate and return FIR filter coefficients for a bandstop and a highpass filter, respectively. To initiate the task, general process of calculation of FIR filter coefficients was observed. It was identified that due to similarities in the calculation process, the highpass filter coefficients could be calculated using the bandstop function.

Based on the aforementioned observation, a function, bandstopDesign, was produced, to calculate FIR coefficients for bandstop filter. This function accepted sampling frequency of the data to be filtered, lower and upper limits of stopband rejection frequencies, and a factor for optimum tap calculation with a default value of 6. This factor was included as an argument to simplify the optimisation process.

The input arguments were processed using functions from the numpy library. The arguments for stopband rejection frequencies were used to calculate frequency resolution, i.e., maximum permissible frequency difference between two consecutive samples for successful filtering. This frequency resolution was further processed with sampling frequency and optimisation factor to calculate the number of taps.

The value of taps was then used to produce an ideal bandstop spectrum. This was achieved by creating an array of ones which contained the same number of elements as the number of taps. Further, the value of taps was used to identify the sample numbers corresponding to the respective cutoff frequencies. As the frequency response is mirrored at half of sampling frequency, these sample numbers were used to set the value of elements lying within this range, and its mirror, to zero, to produce the ideal bandstop response in the frequency domain.

This bandstop spectrum was transformed to the time domain by inverse fast fourier transform, and its real values were used to obtain coefficients for FIR filtering. The array returned by this process comprised of coefficients in positive time followed by those in negative time. In order to correct the time sequence, the coefficients were swapped such that those in negative time were followed by those in positive time. Finally, the swapped coefficients were then multiplied by the blackman window. This window was used because its response is very close to 1 and because it returned coefficients for best stopband rejection.

The next step was to produce the highpass filter function using the designed bandstop filter function. This was done for simplification of the code. Similar to the bandstop design, the highpass filter accepted the sampling frequency of the data to be filtered and factor for tap number optimisation, however, it only accepted one value for cutoff frequency. These values were passed to the designed bandstop function such that the input cut off frequency was used as upper limit frequency of bandstop filter and lower cutoff frequency was set to 0. Thus, the bandstop filter function was used for calculation of highpass filter coefficients.

Finally, these functions were used as methods in the class, NumericFIRCoefficients. Class type implementation improved accessibility. The code for this can be seen in appendix A.1.

QUESTION 2

The aim of this question was to implement a Python FIR filter class. The code for this can be seen in Appendix A2. This design was implemented because the class structure allows the code to be reused, and makes it easy to access from other python modules. It also allows class inheritance for higher-level classes.

An FIR filter class, FIRfilter was produced. Its init method accepted FIR filter coefficients as an argument, and initialized the number of taps, coefficients and buffer attributes. These attributes were then used in the dofiter method to filter the input signal in real time, using the coefficients calculated in question 1.

The dofilter method was designed such that it accepted a scalar value of the raw signal and returned the filtered value to allow for real time processing of the signal. Each buffer value was shifted by 1 position in an iteration to make room for the new value. This was followed by assigning value to the space in the buffer, and calculation of the filtered value by summation of element wise product of buffer with the respective coefficients. Finally, the filtered value is returned.

The FIRfilter class within the firfilter.py code was imported into hpbsfilter.py, and its dofilter method was implemented in the main function, as seen in appendix, A1. According to Zeynep Oner's student matriculation number, the following ECG data file was downloaded from moodle: ECG-msc-matric-6.dat. This ecg signal was processed using commands from numpy library, while matplotlib was used for the respective plots. The initial processing of this data involved setting the sampling frequency, the respective cut off frequencies for bandstop and highpass filtering, and calculation of length of this ecg data. To plot in the time domain, the duration and time vector were also calculated. The raw unfiltered ECG signal was then plotted in both time and frequency domains. These plots are shown on figures 1 and 2.

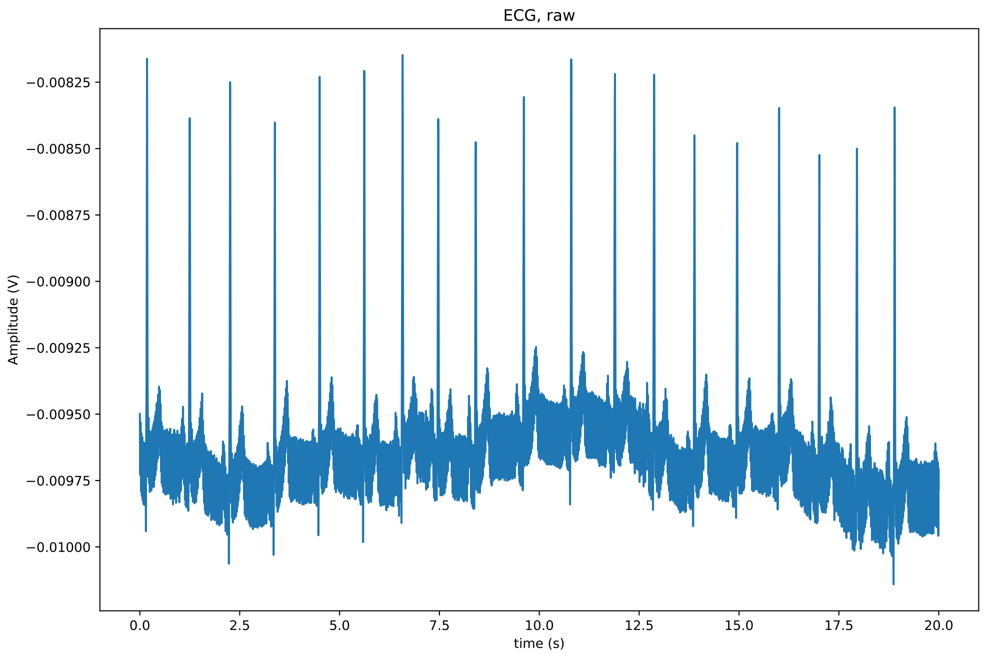

Figure 1. Raw unfiltered ECG signal in the time domain.

The frequency domain plot of the raw ECG signal is shown on figure 2. It clearly shows the unwanted 50 Hz and baseline wander marked by the red arrows. Baseline wander is a low-frequency interference within the ECG signal which can give inaccurate and misleading data. The spectral content of the baseline wander is confined to an interval well below 1 Hz. In figure 3 it can be seen as the huge peak near the 0 Hz (DC) point. The unwanted 50 Hz has a small peak in the frequency domain at 50 Hz.

The baseline wander and the unwanted 50Hz noise were removed by processing the raw ecg data through the highpass and bandstop filters to get the the respective coefficients. While the cutoff frequencies in the bandstop were set to remove the 50 Hz noise, that in the highpass filter was set to remove the baaseline wander. The calculated coefficients were then input to the designed FIR filter class before the raw ECG signal was process through the FIR filter method.

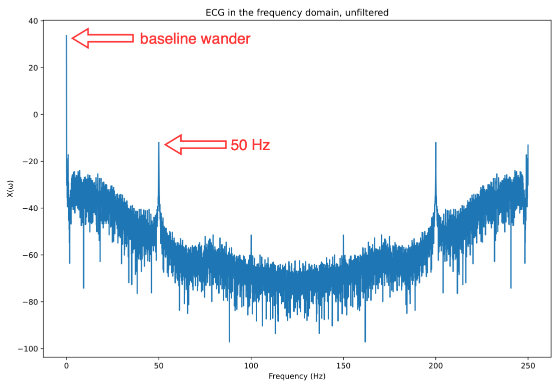

Figure 2. The raw ECG signal in the frequency domain, unfiltered in dB.

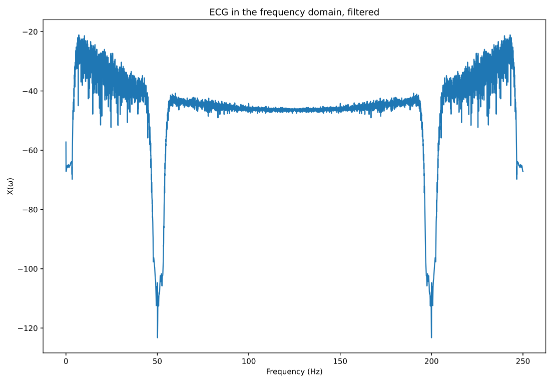

Figure 3. The ECG signal in the frequency domain after being filtered by an FIR filter, in dB.

Figure 3 shows the filtered ECG. It can be seen that the amplitude of the unwanted frequencies have been attenuated. The filtered ECG signal in time and frequency domains are shown in figures 3 and 4, respectively.

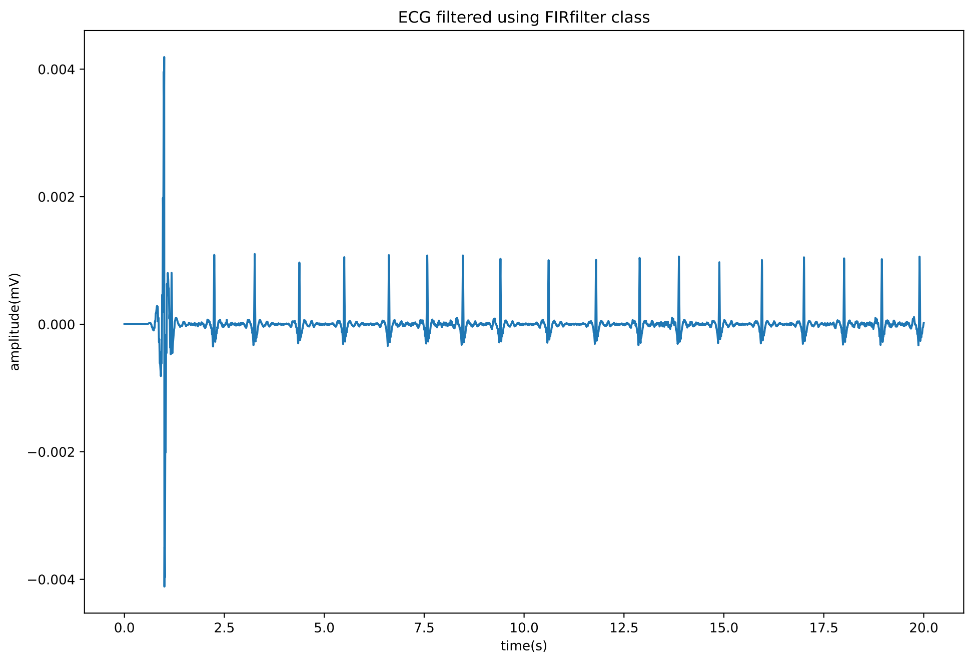

Figure 4. The ECG signal in the time domain after being filtered by an FIR filter.

We can clearly see the DC component and the 50 Hz noise removed. For comparison, the filtered and unfiltered data were plotted together, and zoomed in. The filtered ECG is shown in the figures 5 and 6.

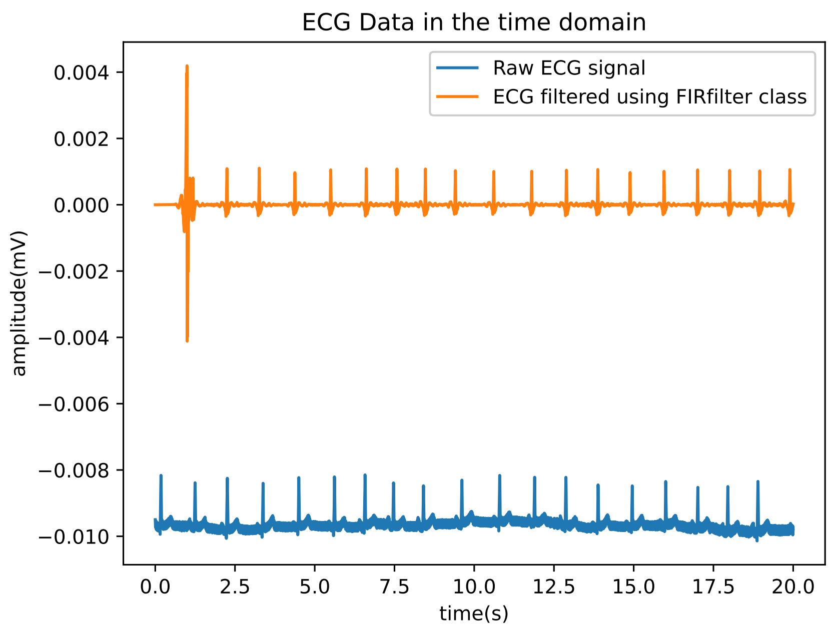

Figure 5. The full ECG data - before and after filtering using the FIR filter.

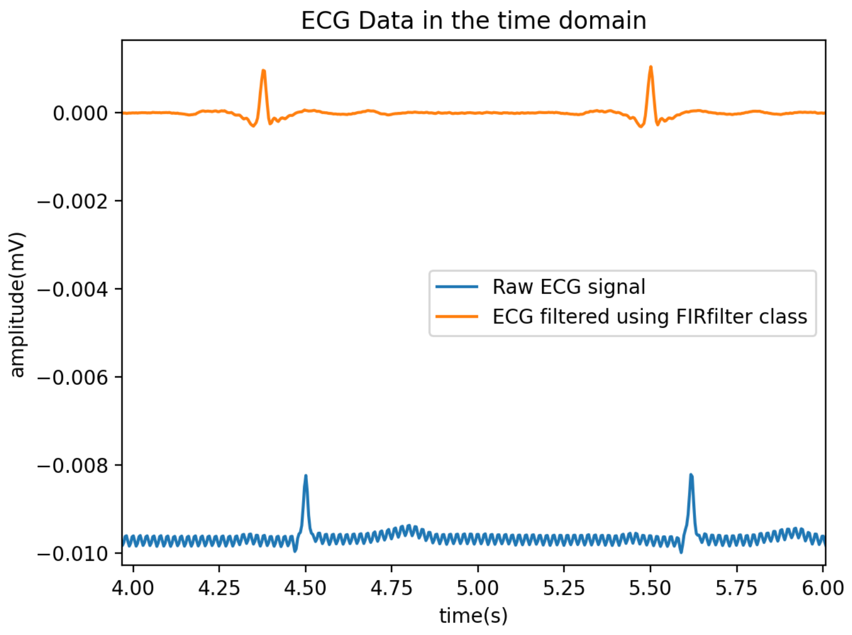

Figure 6. Zoomed in on figure 5 showing two pulses.

The zoomed in view in figure 8 once again confirms that the PQRST is intact and has not been distorted, and that most of the noise has been removed from the original ECG signal.

QUESTION 3

In this question, the aim was to use an adaptive LMS filter for 50 Hz noise removal. An LMS adaptive filter function, doFilterAdaptive, was then created within the FIRfilter class. It was designed to accept scalar values of the raw signal and the canceller, along with the learning rate. The respective arguments were then used in a for loop to modify or adapt the coefficients of the filter. This was then implemented in the main function of lmsfilter.py. This implementation can be seen in appendix A3 of the code.

The initial processing in this function was similar to that in implementation of question 2. Additions for implementation of LMS filter include setting value of noise and learning rate respectively. These values are used in noise removal using LMS filter.

LMS filtering is achieved by iterative modification of coefficients. The filter coefficients change according to an error function $e(n)$, where the adaptive input signal $d(n)$ is compared with the output signal $y(n)$. This adaptive signal can learn over time, and the learning is controlled by the learning rate. Finally, baseline wander is removed using highpass filter in each iteration. The results obtained are shown in the figures below. μ.

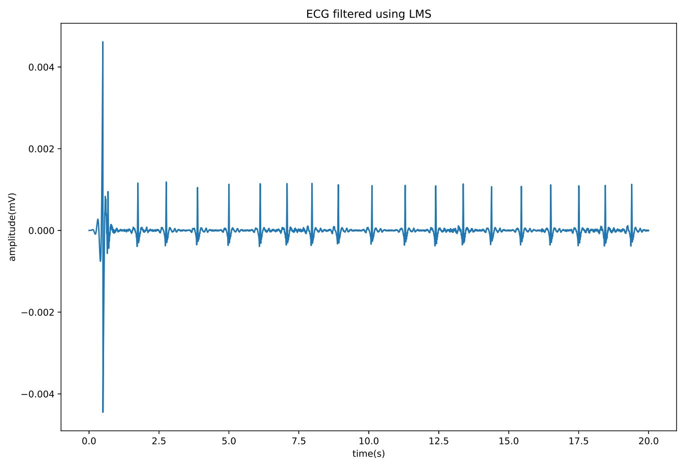

Figure 7. The ECG signal filtered using the FIR LMS filter - in the time domain.

Further, figure 10 confirms that the PQRST is intact and has not been distorted, and most of the noise has been removed from the original ECG signal.

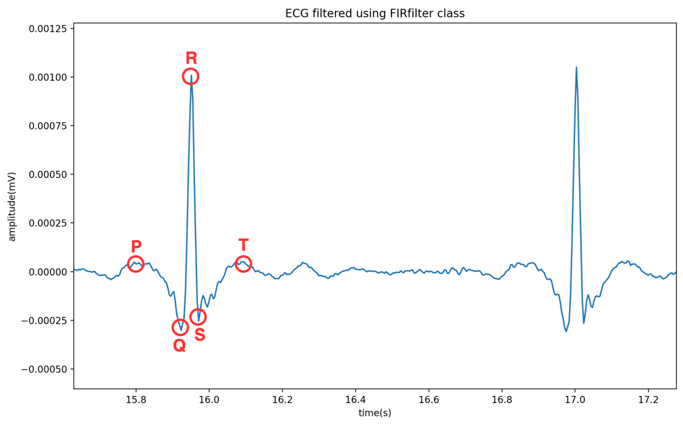

Figure 8. The ECG signal filtered using the FIR LMS filter - zoomed in on two pulses

QUESTION 4

In the last question, it was required to detect R-peaks in our ECG using a wavelet. R-peaks are the largest peak in our ECG, and thus they are detectable using different kinds of wavelets. The wavelet we used is a sinc wavelet, because it is easy to implement and it works well since the desired peak is the largest peak. When we plot the time reversed the extracted template and the wavelet side by side, their similarity is observed.

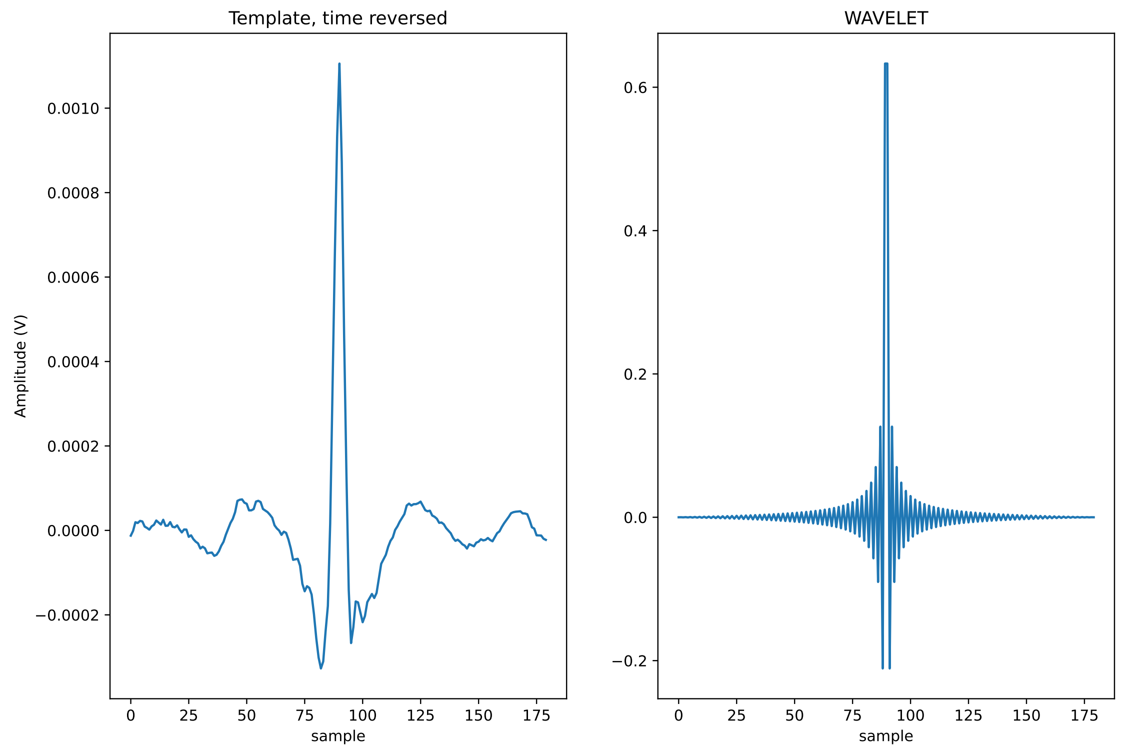

Figure 9. Time-reversed template vs. Wavelet

It was observed that the peak in our sinc wavelet aligns with our R-peak, which means that the wavelet can be used to detect the R-peaks in the entire signal.

For detection, the FIRfilter class was used to filter the ECG signal with the wavelet and the template. These were then compared. This way, the R-peaks were much more defined. Additionally, the detected signals were squared to increase the signal-to-noise ratio. Further, a threshold was applied to finalize the detector. The threshold values were determined by inspecting the previously found signals. Any sample larger than the threshold was identified a peak, and was assigned a value of 1. The other signals had a value of 0. The detector signals can be seen below:

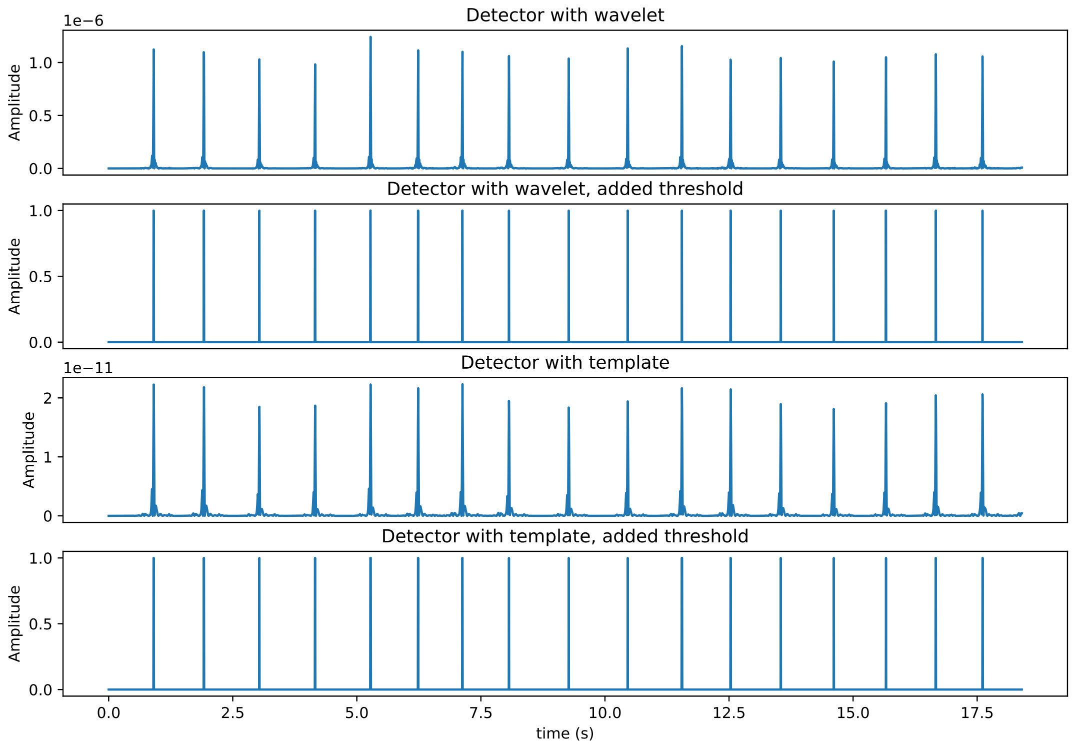

Figure 10. Detector signal, with and without threshold

Both the detectors had successfully detected all the R-peaks. It was observed see that the only major difference was that detection with template gave peaks which were much smaller than in value, but that was overcome by decreasing the threshold. Then, R-peaks were used to find the momentary heart rate. The heart rate was found by calculation of time difference between two peaks, inverting it, and then multiplying it by 60 to get the beats per minute. Further, from figure 12, it was also observed that momentary heart rate for both, wevelet and template, would be identical, as the detector was identical.

There was one problem that was encountered, that some peaks were too small in amplitude. Therefore, to include all peaks, the threshold is made as small as possible. However, for a small threshold, the final detector signal included more than 1 samples for some peak. The heart rate calculator then took the two samples as two separate peaks, and the instantaneous heart rate was found incorrectly. To overcome this, a state machine was coded. It had two states: `READ` and `NOREAD`. The states were calculated by a time threshold, and if the difference in time between the current t and the first peak was smaller than the threshold, the second peak was not read. In this case, the state machine returned the value `NOREAD`. Otherwise, the state machine returned the value `READ`. If the state was `NOREAD`, the second time variable was not updates. With this state machine, excessive samples were ignored and the heart rate was found correctly. The momentary heart rate vs time was found by inspecting all the peaks.

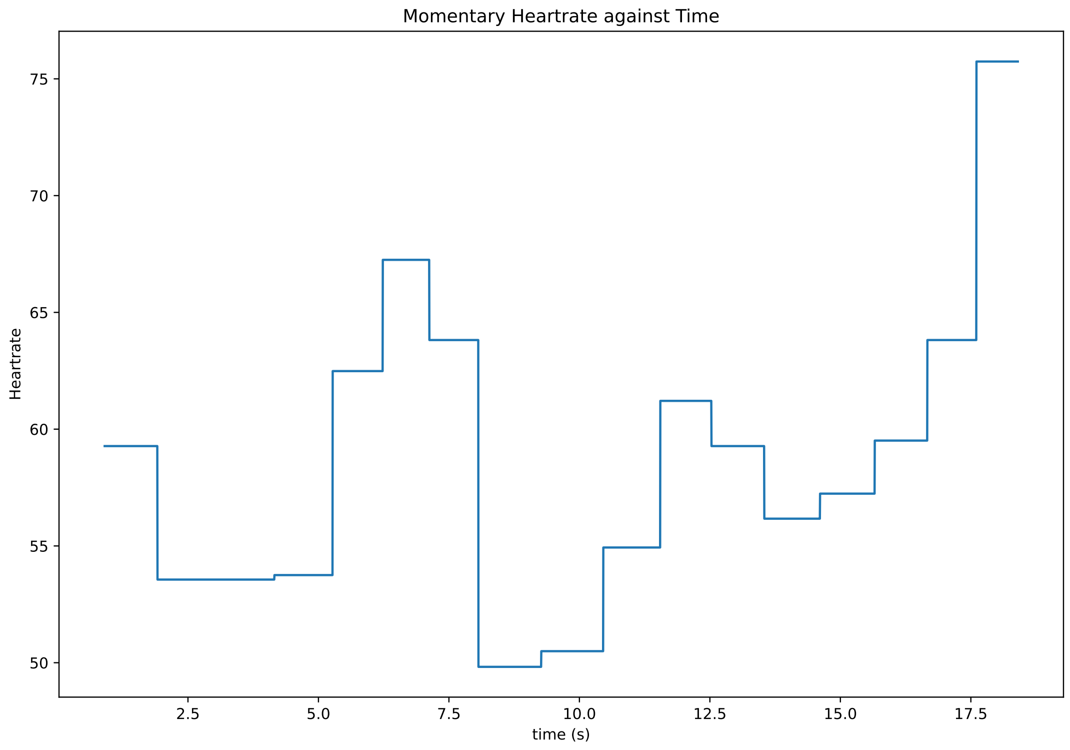

Figure 11. Momentary heartrate vs. time

The momentary heart rate against time graph was found with the detector that was created using the wavelet, as seen in figure 13.

A.1 hpbsfilter.py

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

'''
Class for generation of Numeric Coefficients for FIR filtering
\Method highPassDesign: Generates Highpass Numeric Coefficients for FIR filtering
\Method bandstopDesign: Generates Bandstop Numeric Coefficients for FIR filtering
'''
class NumericFIRCoefficients():
    '''
    HighPass Filter Function
    \parameter sample_rate: Samping frequency of the data to be filtered
    \parameter cut_f: Cut off frequency of the high pass filter
    \parameter factor: Factor for tap calculation. Helps optimise Stopband Rejection. Default value is
        6.
    '''
    def highpassDesign(fs, fc, factor = 6):
        return NumericFIRCoefficients.bandstopDesign(fs, 0, fc, factor)


    #BandStop Function: Filters signal for freqnecies between the two input values
    '''
    Bandstop Filter Function
    \parameter sample_rate: Samping frequency of the data to be filtered
    \parameter cut_f1: Lower limit of frequencies to be filtered
    \parameter cut_f1: Upper limit of frequencies to be filtered
    \parameter factor: Factor for tap calculation. Helps optimise Stopband Rejection. Default value is
        6.
    '''
    def bandstopDesign(fs, f1, f2, factor=6):
        f_res = np.abs(f2 - f1) #Resolution, i.e., maximum frequency difference between two consecutive
            samples
        taps = int(factor*(fs/f_res)) #Calculation of Number of Taps using Sampling Frequency,
            Frequency Resolution, and Factor
        bandstop_spectrum = np.ones(taps) #Array for Ideal Bandstop Response in Frequency Domain
        k_cutoff1 = int(f1/fs * taps) #Calculation of Samples Corresponding to Lower Limit of
            frequencies to be filtered
        k_cutoff2 = int(f2/fs * taps) #Calculation of Samples Corresponding to Upper Limit of
            frequencies to be filtered

        bandstop_spectrum[k_cutoff1: k_cutoff2+1] = 0 #Ideal Bandstop Response in Frequency Domain
        bandstop_spectrum[taps - k_cutoff2:taps - k_cutoff1] = 0 #Mirrored Ideal Bandstop Response in
            Frequency Domain

        bandstop_coeff = np.real(np.fft.ifft(bandstop_spectrum)) #Real values of IFFT Response, i.e.,
            unswapped coefficients
        h_bandstop = np.append(bandstop_coeff[int(taps/2):taps], bandstop_coeff[0:int(taps/2)])
            #Corrected (Swapped) Coefficients
        h_bandstop = h_bandstop * np.blackman(taps) #Window multiplied for Better Stopband Rejection

        return bandstop_spectrum, h_bandstop #Values of Ideal Bandstop Response and Coefficients for
            FIR Filter returned
```

```
'''
Main Function
Implementation Highpass and Bandstop filters
'''
def main():
    ecg = np.loadtxt('ecg.dat')
    fs = 250  #Sampling rate
    N = len(ecg)  #Number of samples
    f1 = 45  #Cutoff frequency 1
    f2 = 55  #Cutoff frequency 2
    fc = 5   #Highpass cutoff frequency to remove baseline wander
    duration = N /fs #Duration of the ecg
    t = np.linspace(0, duration, N)

    # obtaining the coefficients from q1
    highpass_spectrum, h_highpass = NumericFIRCoefficients.highpassDesign(fs, fc)
    bandstop_spectrum, h_bandstop = NumericFIRCoefficients.bandstopDesign(fs, f1, f2)

    # filtering the ECG signal using the FIRfilter class
    FIR_highpass = firfilter.FIRfilter(h_highpass)
    FIR_bandstop = firfilter.FIRfilter(h_bandstop)

    output = np.zeros(N)
    output2 = np.zeros(N)

    for i in range(N):
        # output removing the baseline wander using the highpass filter
        output[i] = FIR_highpass.dofilter(ecg[i])
        # output 2 removing the 50Hz interference using the bandstop filter
        output2[i] = FIR_bandstop.dofilter(output[i])

    #Plots for raw and filtered ecg
    #Plotting the raw ecg
    plt.figure(figsize=(12, 8))
    plt.plot(t, ecg)
    plt.xlabel('time (s)')
    plt.ylabel('Amplitude (V)')
    plt.title('ECG, raw')
    plt.savefig('ecg_raw.eps', format='eps')

    #Plotting raw ecg in frequency domain
    ecgf = np.fft.fft(ecg)
    faxis = np.linspace(0, fs, N)
    plt.figure(figsize=(12, 8))
    plt.plot(faxis, 20*np.log10(np.abs(ecgf)))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('X(\u03C9)')
    plt.title('ECG in the frequency domain, unfiltered')
    plt.savefig('ecgf_unfiltered.eps', format='eps')

    #Plotting the filtered ecg
    ecg_filtered = output2
    plt.figure(figsize=(12, 8))
    plt.title('ECG filtered using FIRfilter class')
    plt.xlabel('time(s)')
    plt.ylabel('amplitude(mV)')
    plt.plot(t, ecg_filtered)
    plt.savefig('ecg_filtered_hpbsfilter.eps', format='eps')

    #Plotting raw ecg in frequency domain
    ecgf_filtered = np.fft.fft(ecg_filtered)
    plt.figure(figsize=(12, 8))
    plt.plot(faxis, 20*np.log10(np.abs(ecgf_filtered)))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('X(\u03C9)')
    plt.title('ECG in the frequency domain, filtered')
    plt.savefig('ecgf_filtered.eps', format='eps')

    plt.show()

    return ecg_filtered, fs

if __name__ == '__main__':
    main()
```

A.2 firfilter.py

```
import numpy as np

'''
FIR Filter Class
\method __init__: For declaration of variables used in following methods
\method dofilter: Function for FIR filter
\method doFilterAdaptive: Function for Least Mean Square Filter for adaptive filtering
'''
class FIRfilter:

    def __init__(self, _coefficients):
        self.number_of_taps = len(_coefficients) #Number of Taps
        self.coefficients = _coefficients
        self.buffer = np.zeros(self.number_of_taps) #Creating Buffer

    '''
    dofilter
    \parameter v: scalar value of coefficient to process the respective value of ecg
    '''
    def dofilter(self, v):
        for i in range(self.number_of_taps - 1): #Shifting Values and Making Room for new values
            self.buffer[self.number_of_taps - i - 1] = self.buffer[self.number_of_taps - i - 2]
        self.buffer[0] = v #Setting input as new value in buffer
        result = np.inner(self.buffer, self.coefficients) #Calculating result: Multiplying input with
            coeff.
        return result
    '''
    doFilterAdaptive
    \parameter signal: Scalar vale from signal to be filtered
    \parameter noise: Noise to be removed from the signal
    \parameter learningRate: Rate of adaptation for filtering
    '''
    def doFilterAdaptive(self, signal, noise, learningRate):
        for j in range(self.number_of_taps):
            self.coefficients[j] = self.coefficients[j] + (signal - noise) * learningRate *
                self.buffer[j]
```

A.3 lmsfilter.py

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

'''
Main Function
Implementation least mean square filter
'''
def main():
    ecg = np.loadtxt('ecg.dat')
    fs = 250            #Sampling rate
    N = len(ecg)        #Number of samples
    fc = 5              #Highpass cutoff frequency to remove baseline wander
    duration = N /fs    #Duration of the ecg
    t = np.linspace(0, duration, N)
    noise = 50
    learning_rate = 0.001

    highpass_spectrum, h_highpass = hpbsfilter.NumericFIRCoefficients.highpassDesign(fs, fc)
    FIR_highpass = firfilter.FIRfilter(h_highpass)

    signal_in = firfilter.FIRfilter(np.zeros(len(h_highpass)))
    lms_output = np.empty(len(ecg))
    filtered_signal = np.empty(len(ecg))

    for i in range(len(ecg)):
        # output removing the 50Hz interference using the LMS filter
        ref_noise = np.sin(2.0 * np.pi * noise / fs * i)
        canceller = signal_in.dofilter(ref_noise)
        output_signal = ecg[i] - canceller
        signal_in.doFilterAdaptive(ecg[i], canceller, learning_rate)
        lms_output[i] = output_signal
        # now remove the baseline wander using the highpass filter
        filtered_signal[i] = FIR_highpass.dofilter(lms_output[i])

    # plotting the ecg signal filtered using LMS
    plt.figure(2, figsize=(12, 8))
    plt.plot(t, filtered_signal)
    plt.xlabel('time(s)')
    plt.ylabel('amplitude(mV)')
    plt.title('ECG filtered using LMS')
    plt.savefig('ecg_filtered_lmsfilter.eps', format='eps')
    plt.show()

if __name__ == '__main__':
    main()
```

A.4 hrdetect.py

```python
import numpy as np
import matplotlib.pyplot as plt
import hpbsfilter
import firfilter
import scipy.signal as signal

class TimeStates():
    """
    This class is a very simple state machine that is used to
    determine whether a particular T value is being used while
    calculating the pulse.

    In the detector with threshold graph, it can be seen that the
    peaks where the detector is 1 include 2 samples. This is because
    some peaks in the detector are low in amplitude, and the threshold
    has to be low as well. We need to override the 2nd sample to calculate
    the pulse; as the way to calculate the pulse is to find 2 consecutive
    peaks, calculate the time difference, invert it and multiply by 60.

    The way to override the 2nd sample is by introducing this state machine
    with 2 states: READ and NOREAD. When the state is READ, we take the value
    of the time we currently are in and use it to calculate the pulse. If the
    state is NOREAD, the t value is ignored.

    The state is determined by a threshold. If the time difference between the
    current time and our first peak is smaller than the threshold, it means
    that we are still on the same peak and the state should be NOREAD, which
    means this particulat t value must be ignored. Else, the state is READ and
    we are on another peak.
    """
    _state = 'READ' # the initial state is read

    def switch_states(self, t, t1):
        delta_t = t - t1
        threshold_state = 5E-3
        if delta_t < threshold_state:
            self.state_ = 'NOREAD'
        else:
            self.state_ = 'READ'
        return self.state_

def create_template(ecg, fs):
    """
    Extract the template from the ECG.
    """
    template = ecg[700:880]
    template_reversed = template[::-1] # template reversed
    N = len(template)
    duration = N / fs
    t = np.linspace(0, duration, N)
    return t, template, template_reversed, len(template)


def create_wavelet(N, fs):
    """
    Time reverse the template and obtain the wavelet
    that will be used to filter the ECG.
    """
    duration = N / fs
    t = np.linspace(-duration/2, duration/2, N)
    wavelet = np.sinc(t*fs)
    return wavelet


def detect(ecg, fs, wavelet):
    """
    Detection algorithm that filters the signal, then
    squares it in order to increase the SNR.
    """
    ecg_cropped = ecg[400:]
    N = len(ecg_cropped)
    duration = N / fs
    t = np.linspace(0, duration, N)
    FIR_wavelet = firfilter.FIRfilter(wavelet)
    detector = np.zeros(N)
    for i in range(N):
        detector[i] = FIR_wavelet.dofilter(ecg_cropped[i])
    detector = detector**2
    return t, detector, N


def apply_threshold(detector, N, template_type):
    """
    Applies a threshold and determines the peaks
    """
    threshold = 0.0
    if template_type == 'wavelet':
        threshold = 9E-7 # amplitude value to determine the peaks
    elif template_type == 'template':
        threshold = 1.8E-11
    detector_new = np.zeros(N)
    for i in range(N):
        if detector[i] > threshold:
            detector_new[i] = 1
    return detector_new


def calculate_heartrate(t, detector, N, fs):
    """
    The momentary heartrate is calculated by going through the peaks
    one-by-one, finding the interval between two consecutive peaks, and
    inversing it. The result gives us the beats per second, and by multiplying
    with 60, we get the beats per minute.
    """
    heart_rate = np.zeros(N)
    t1_temp = 0 # temp value to store previous peak
    peak_range = np.arange(0, N) # we reduce the peak range in order to override previous peaks
    count = 1 # counter to count peaks
    time_state = TimeStates()
    for i in range(N):
        t1 = t1_temp # time of the first peak
        t2 = np.max(t) # time of the second peak initialized as the maximum time
        for i in peak_range:
            if detector[i] == 1: # we found a peak
                state = time_state.switch_states(t[i], t1) # determine state according to delta_t
                if count < 2: # only the first peak is found with this
                    t1 = t[i]
                    count += 1
                elif count == 2: # any subsequent peak falls into this category
                    if state == 'READ':
                        t2 = t[i]
                        count += 1
                else: # two peaks have been located, and we need to leave the loop
                    break

        delta_t = t2 - t1 # time difference between two peaks
        hr_momentary = 1/delta_t * 60 # 1/delta_t is the frequency of peaks, times 60 gives bpm

        sample_1 = int(t1 * fs) # sample where the first peak is
        sample_2 = int(t2 * fs) # sample where the second peak is
        heart_rate[sample_1:sample_2+2] = hr_momentary # momentary heartrate between two samples

        t1_temp = t2 # second peak in iteration i is the first peak in iteration i+1
        peak_range = np.arange(sample_2+2, N) # peak range is updated
        count = 2

    # crop the part before the first peak, as the heart rate is not calculated
    # before it and the array is 0
    hr_init = 0.0
    for sample in heart_rate:
        if sample > 0:
            hr_init = np.where(heart_rate == sample)[0][0]
            break
    heart_rate_cropped = heart_rate[hr_init:]

    # update the t variable for a correct x axis
    t_cropped = t[hr_init:]

    return t_cropped, heart_rate_cropped


def main():
    ecg_filtered, fs = hpbsfilter.main()

    t_template, template, template_reversed, N_template = create_template(ecg_filtered, fs)
    wavelet = create_wavelet(N_template, fs)
    t, detector_wavelet, N = detect(ecg_filtered, fs, wavelet)
    t, detector_template, N = detect(ecg_filtered, fs, template)
    detector_wavelet_new = apply_threshold(detector_wavelet, N, 'wavelet')
    detector_template_new = apply_threshold(detector_template, N, 'template')
    t_heartrate, heart_rate = calculate_heartrate(t, detector_wavelet_new, N, fs)

    plt.figure(1, figsize=(12, 8))
    plt.plot(t_template, template)
    plt.title('One heartbeat')
    plt.xlabel('time (s)')
    plt.ylabel('Amplitude')
    plt.savefig('heartbeat.eps', format='eps')
    plt.figure(2, figsize=(12, 8))
    plt.subplot(121)
    plt.plot(template, label='template, time reversed')
    plt.title('Template, time reversed')
    plt.xlabel('sample')
    plt.ylabel('Amplitude (V)')
    plt.subplot(122)
    plt.plot(wavelet, label='WAVELET')
    plt.title('WAVELET')
    plt.xlabel('sample')
    plt.savefig('template-wavelet.eps', format='eps')

    fig, ax = plt.subplots(4, 1, figsize=(12, 8))
    ax[0].plot(t, detector_wavelet)
    ax[0].set_ylabel('Amplitude')
    ax[0].set_title('Detector with wavelet')
    ax[0].get_xaxis().set_visible(False)

    ax[1].plot(t, detector_wavelet_new)
    ax[1].set_ylabel('Amplitude')
    ax[1].set_title('Detector with wavelet, added threshold')
    ax[1].get_xaxis().set_visible(False)

    ax[2].plot(t, detector_template)
    ax[2].set_ylabel('Amplitude')
    ax[2].set_title('Detector with template')
    ax[2].get_xaxis().set_visible(False)

    ax[3].plot(t, detector_template_new)
    ax[3].set_xlabel('time (s)')
    ax[3].set_ylabel('Amplitude')
    ax[3].set_title('Detector with template, added threshold')
    plt.savefig('detector.eps', format='eps')

    plt.figure(4, figsize=(12, 8))
    plt.plot(t_heartrate, heart_rate)
    plt.xlabel('time (s)')
    plt.ylabel('Heartrate')
    plt.title('Momentary Heartrate against Time')
    plt.savefig('heartrate.eps', format='eps')

    plt.show()


if __name__ == '__main__':
    main()
```