University of Glasgow

James Watt School of Engineering

Report of Assignment 2: FIR filter

Authors:

Minqiu Zhou 2357594Z
Zhongxu Dong 2357507D

Declaration of Originality and Submission Information

I affirm that this submission is my own / the groups work in accordance with the University of Glasgow Regulations and the School of Engineering Requirements.

ENG 4053: Digital Signal Processing

November 9, 2020

1 Task One: ECG Filtering

1.1 Implementation of a Ring Buffer

Ring buffer is applied to avoid shift that happens every time step, therefore it is efficient and resource-saving. This ring buffer is slightly different from normal ring buffer. Generally speaking, the read and write pointers of a ring buffer should be independent, which means they are two separate parameters. However, in this specific FIR filter task, the read pointer should move with the write pointer to locate the last data, and the read pointer will iterate exactly a loop of data to do the filtering process. Finally, the read pointer will be the same value as it starts. Therefore, the read and write pointer share the same offset in this task. The entire code of ring buffer is shown below:

```
class RingBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [0]*capacity
        self.offset = -1

    def write(self, item):
        self.offset = (self.offset +1) % self.capacity
        self.queue[self.offset] = item

    def read(self):
        tmp = self.queue[self.offset]
        self.offset = (self.offset - 1)%self.capacity
        return tmp
```

In *__init__* function, the ring buffer is set to the specific size by the parameter *capacity*, the array that carries the data is *queue* with the pointer is initialized at -1, which locates the end of the array.
In the *write* function of ring buffer, the pointer moves to the next index first (+1). The $\%self.capacity$ command ensure that pointer is within the range of the size of ring buffer.Then, data is written to the buffer.
The *read* function simply reads the data of the pointer first. Next step of moving pointer is different from that step in write function. Because FIR filter does convolution of coefficients and data, the sequence of data should be inverted, hence, the iterate direction of calculation is the reverse of writing process. Therefore, the offset should be -1.
A brief illustration of pointer movement is presented below:

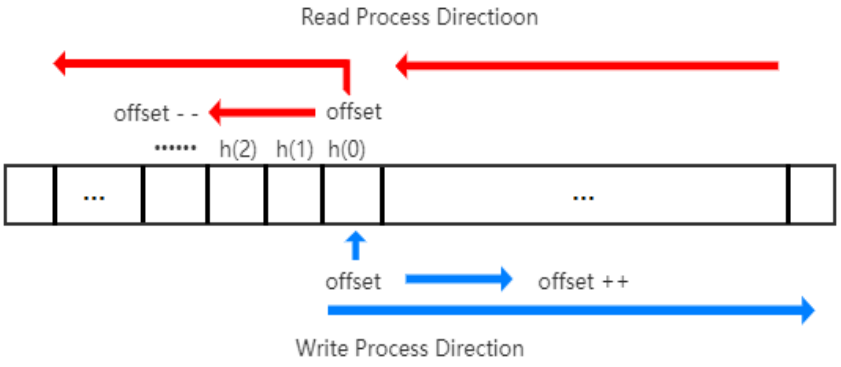

Figure 1: Illustration of Read and Write Process of Ring Buffer

1.2 Implementation of FIR Filter

The core idea of FIR filter is discrete convolution of input signal and coefficients, h(n). Below is the code for FIR filter.

```
class FIR_filter:
    def __init__(self, _coefficients):
        self.M = len(_coefficients)
        self.coefficients = _coefficients
        global R
        R= RingBuffer(self.M)

    def dofilter(self,v):
        R.write(v)
        result = 0

        for i in range(self.M):
            result = R.read() * self.coefficients[i] + result

        return result
```

The input of FIR filter is the array of coefficients, h(n). In *__init__* function, the filter stores coefficients and defines a ring buffer of the same size as h(n).
The *dofilter* function takes one sample as input and generate one sample at output, hence the FIR filter could be employed in real time system. For each sample inputted, it is stored into the ring buffer. Then, the convolution of the data in ring buffer and the coefficients - h(n) is calculated in a *for* loop, and the result is returned as the output.

1.3 Unit Test

A unit test is written to test the FIR filter. A delta impulse signal is input to an averaging FIR filter.

```
def unittest():
    #input signal
    x = [1,0,0,0,0,0,0,0,0,0]
    #coefficient
    h = [0.5, 0.5]

    F = FIR_filter(h)

    y = [None]*len(x)
    for i in range(len(x)):
        y[i] = F.dofilter(x[i])

    print(y)

    plt.title('unittest')
    plt.plot(y)
    plt.show()

if __name__ == "__main__":
    unittest()
```

The desired output should be [0.5,0.5,0,0,0,0,0,0,0,0]. From Fig.2, the output of unit test is identical to desired output.

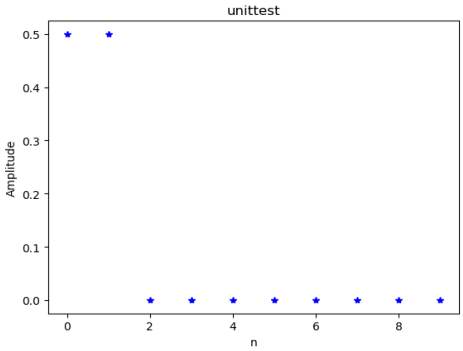

Figure 2: Result of Unit Test

1.4 ECG Filtering

1.4.1 Determine Ideal Frequency Response

Loaded raw ECG signal is shown in Fig.3 in frequency domain and in dB, while details around 1Hz is presented in Fig.4

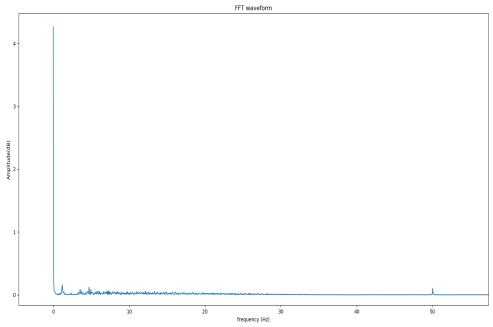

Figure 3: Raw ECG in frequency domain in dB

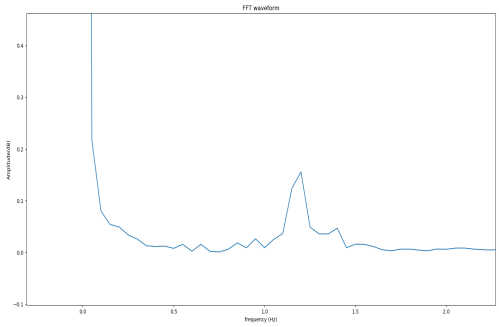

Figure 4: Detailed ECG around 0-2HZ

From Fig.3, the major noises are DC and 50Hz interference. In addition, a normal heart rate for adults ranges from 60 to 100 beats per minute, which implies signals higher than 1Hz is valid in this case of heart beat detection. According to Fig.4, DC interference ranges from 0-0.5Hz. Hence the FIR filter should remove signal ranges from 0 - 0.5 Hz and 45 - 55 Hz.Therefore, the ideal frequency response of this ECG filter is shown in Fig.5.

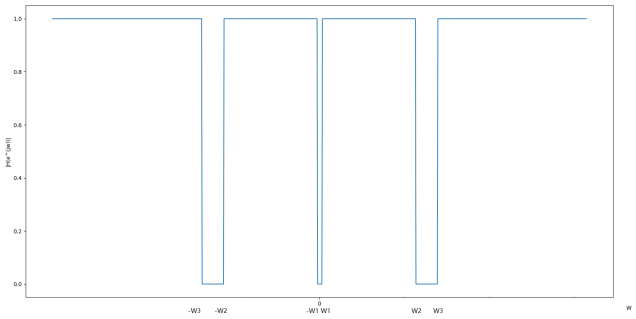

Figure 5: Ideal Frequency Response of ECG Filter

However, in Python, it is easier to deal with array in positive x-axis. Together with the mirror property, the ideal frequency response in negative x-axis is mirrored to positive x-axis in $[\pi, 2\pi]$. (Fig.6)

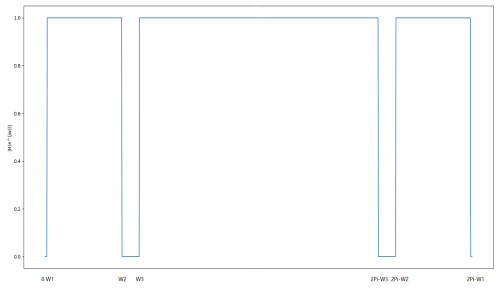

Figure 6: Mirrored Ideal Frequency Response

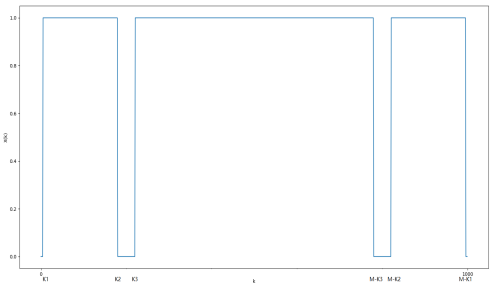

Figure 7: Array of Frequency Coefficients

Implementation of Fig.6 in python is creating a discrete array of frequency coefficients, which is displayed in Fig.7. In Fig.7, M is the number of taps of the filter, which will be discussed later.
Fig.7 is identical to Fig.6, the only difference of the two is the X-axis, but they are correspondent.Therefore, the formula to calculate the according X-axis in Fig.7 is derived as follow:
In Fig.6, an arbitrary value in X-axis, w_1:

$$w_1 = 2\pi f_1 \tag{1}$$

$$f_1 = \frac{F}{F_s} \tag{2}$$

, where f_1 is the normalized frequency, F is is the signal frequency and F_s is the sampling frequency, which is 250Hz in this lab.
In Fig.7, the w_1 is correspondent to k_1:

$$\frac{k_1}{M} = \frac{w_1}{2\pi} \tag{3}$$

According to Eq.(1)-(3), the X value in Fig.7 is:

$$k_1 = \frac{F}{F_s * M} \tag{4}$$

The index of an array should be an integer, thus in Python, k_1 should be changed into an integer. To remove DC and 50Hz interference, the Python code that calculating the index in coefficient array is:

```
k1 = int(0.5/Fs * M)
k2 = int(45/Fs * M)
k3 = int(55/Fs * M)
```

Then, the number of taps, or parameter M, should be determined. M is finite to limit the number of coefficients to M taps, which will affect the frequency resolution. In theory, the resolution needed here is 0.5Hz, due to the 0-0.5Hz removal. From Eq.(5), the number of taps should be 500, because F_s is 250Hz.

$$\Delta F = \frac{F_s}{M} \tag{5}$$

,where F_s is sampling rate, M is the number of taps and Δ F is the frequency resolution(Fig.E). However, in practice, the transition width is finite, which will influence the filtering performance, the practice M is usually two to three times of theoretical M. Here, to achieve better performance, M is set to 1000.
The Python code below creates the discrete array of frequency coefficients, that removes 0-0.5 and 45-55Hz.

```
H = np.ones(M)
H[0:k1+1] = 0
H[k2:k3+1] = 0
H[M-k1:M] = 0
H[M-k3:M-k2] = 0
```

1.4.2 Inverse Fourier Transform

Next step is to get the impulse response of created $H(n)$, which is executed by the Python command that does IFFT and takes only real part:

```
h = np.real(np.fft.ifft(H))
```

The plot of $h(n)$ in time domain is shown in Fig.8.

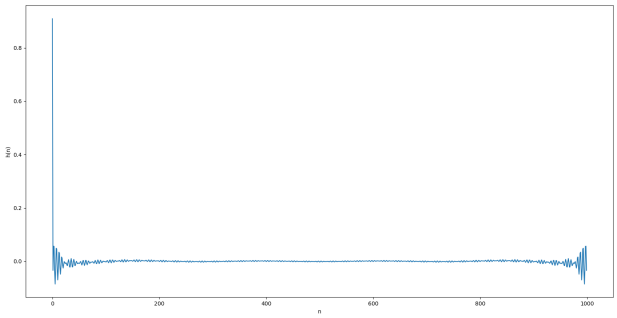

Figure 8: $h(n)$ in time domain

1.4.3 Mirror and Shuffle $h(n)$

From Fig.8, $h(n)$ that should display in negative time domain is mirrored to $[\frac{M}{2},M-1]$. To restore $h(n)$, signals in $[\frac{M}{2},M-1]$ should be mirrored into $[-\frac{M}{2},0]$. However, due to the non-zero values in negative X-axis, $h(n)$ is non-causal after mirror process. The premise of convolution is signals are causal. Hence, to ensure that condition, entire $h(n)$ is shifted from $[-\frac{M}{2},\frac{M}{2}]$,to $[0,M-1]$. The code implementing mirror and shuffle process is listed below:

```
h_new = np.zeros(M)
h_new[0:int(M/2)] = h[int(M/2):M]
h_new[int(M/2):M-1] = h[0:int(M/2)-1]
```

Fig.9 shows the $h(n)$ after shuffle, which is a causal signal now.

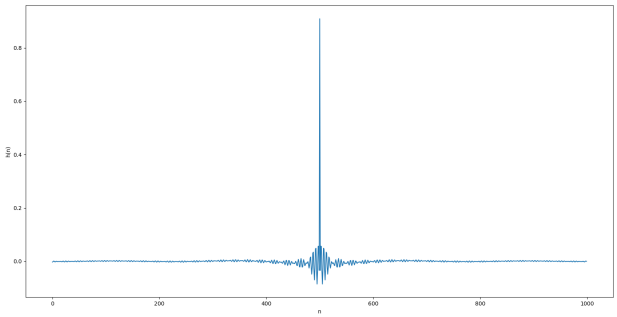

Figure 9: $h(n)$ after Mirror and Shuffle in time domain

1.4.4 Window Function

Window function is applied to improve the $h(n)$ for the sake of fixing ripples and band stop rejection. To discuss which window function is more suitable in this ECG filter, a couple of comparison are carried out. Fig.10-12 explain performance of different window function in terms of ripples, transition width and band stop rejection in frequency domain.

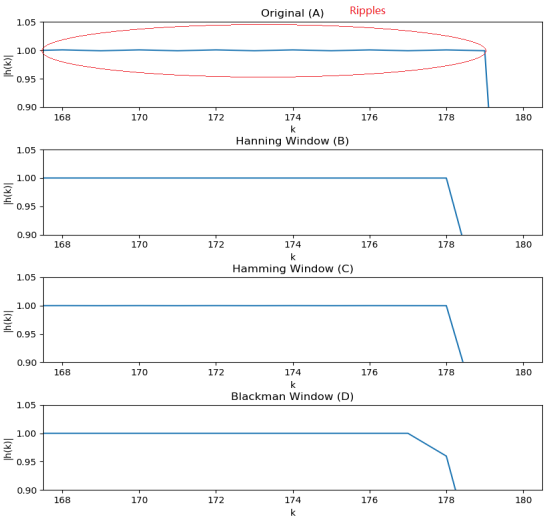

Figure 10: Ripples of Window Functions

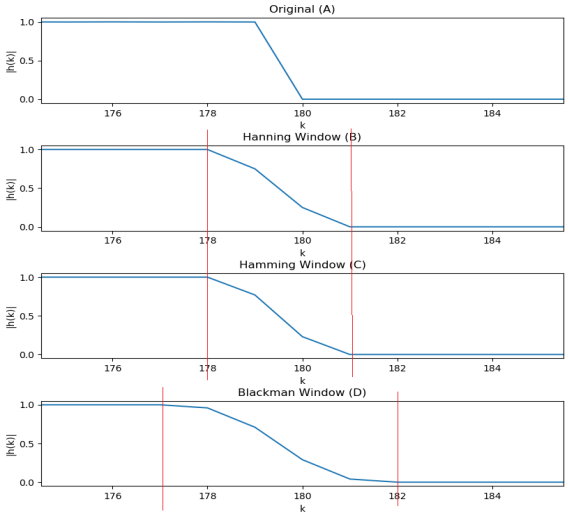

Figure 11: Transition Width of Window Functions

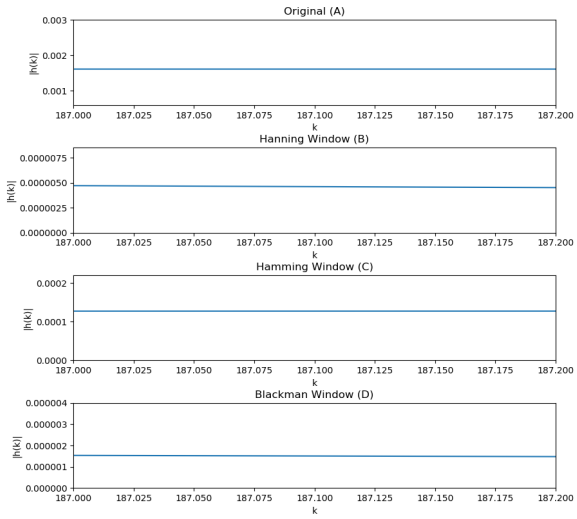

Figure 12: Band Stop Rejection of Window Functions

From Fig.10, sub figures (B)-(D) show similar performance of ripple fixing of Hanning, Hamming and Blackman functions. From Fig.11, the Blackman function results in wider transition width than other two. From Fig.12, the stop band performance of Hunning and Blackman are almost 100 times greater than that of Humming.
In all, considering smaller transition width and greater stop band rejection, Hamming function is applied in this ECG filtering task.

1.4.5 ECG Filtering

Till now, the preparation of FIR filter setting is finished. Final step is read out the coefficients and put them into an FIR filter. The according Python code is:

```
fir_filter = FIR_filter(coefficients)
output = np.zeros(N)
for i in range(N):
    output[i] = fir_filter.dofilter(data[i])
```

```
    return output
```

All previous code in Sec.1.4, which performs preparation and application of FIR filter is packaged in a function:

```
def signal_processing(Fs, data):
    ...
    ...
    return output
```

The input of *signal_processing* is F_s, the sampling rate, and data need to be filtered. The output of this function is filtered signal. Therefore, the ECG is filtered by commands:

```
ECG = np.loadtxt("ECG_ugrad_matric_7.dat")
Fs = 250
result = signal_processing(Fs= Fs, data= ECG)
```

As a result, Fig.13 is the filtered result of ECG signal. A single ECG heart beat after filtering is presented in Fig.14, where PQRST waves are clearly identified.

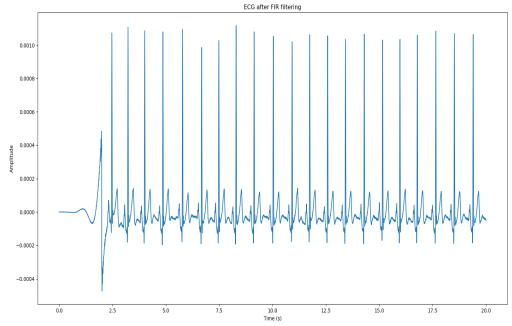

Figure 13: ECG Signal after FIR Filtering

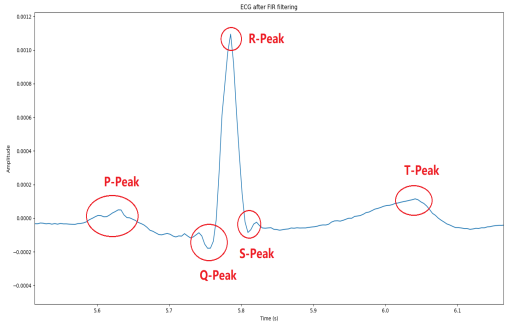

Figure 14: Single Heart Beat Signal

2 Task Two: ECG Heart Rate Detection

In this task, a matching filter will be created to detect the heartbeat in ECG signal, then the momentary heart rate will be calculated.

2.1 Pre-Filtering

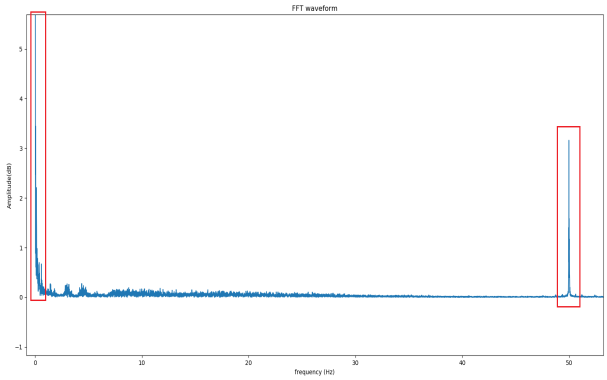

Figure 15: Frequency Spectrum of ECG from Einthoven Database

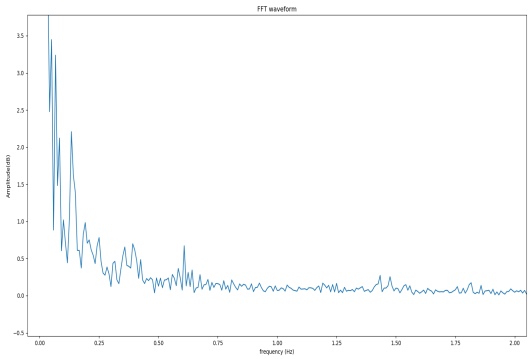

Figure 16: Detailed ECG around 0-2HZ

The first step is pre-filierting. From Fig.15, the frequency spectrum of ECG signal, it can be shown that the noise is mainly distributed in DC and 50Hz, which causes dramatic influence on the heartbeat detection. As discussed before in Sec.1.4.1, the heart rate of a normal person is usually higher than 60 bmp, which is 1Hz. According to Fig.16, which is the zoom in of Fig.15 around 0 - 2Hz, the DC noise ranges from 0 - 1Hz, instead of 0 - 0.5Hz in task 1. Therefore, in task 2, the cut-off frequency of DC removal is set to 1Hz. The same as task 1, the packaged function of FIR filter preparation and application is used with different DC cut-off frequency. Thus, the command of this step is:

```
ECG_new = signal_processing(Fs, ECG)
```

, and another alteration within that function is:

```
k1 = int(1/Fs * M)
```

As the FIR filter is semi-realtime, which contains a delay of $\frac{M}{2}$, the first 500 outputs is set to zero to avoid the undesired data. The ECG signal without DC and 50Hz interference in time domain is shown in Fig.17.

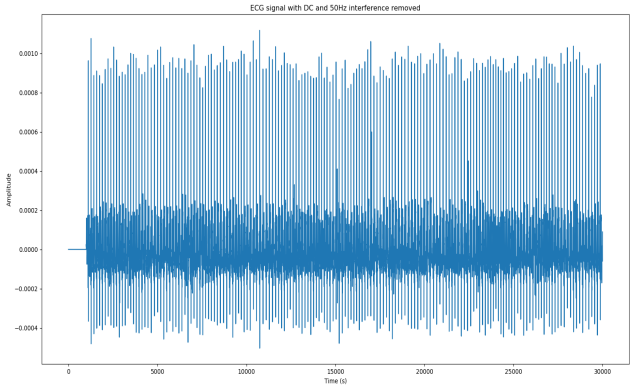

Figure 17: ECG Signal with DC and 50Hz Interference Removed

2.2 Template Creation

A template is a single heart beat with QRST complex. There are two choices of a template. One is using heart beat in Fig.14, which is extracted from the signal in Moodle. Another one is slicing from signal shown in Fig.17, that is from an online database.
To implement the second choice, signal from time interval (40.08, 40.40) is sliced by the Python command:

```
temp = ECG_new[10020:10100]
```

, where *ECG_new* is the cleaned ECG signal, and index = time * sampling rate

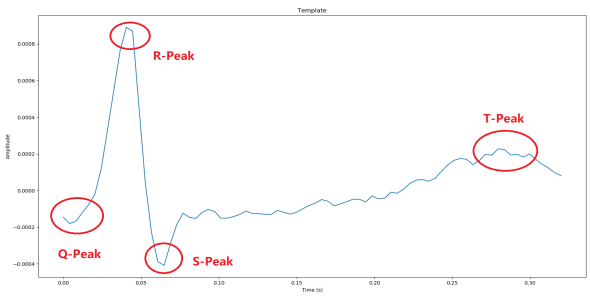

Figure 18: Template

Fig.18 displays the sliced heart beat.

Comparing Fig.14 and Fig.18, or comparing using single heat beat from Moodle and online database, in Fig.18, the amplitude of S-peak is larger than that of Q-peak, but in Fig.14 the amplitude of S-peak is smaller than that of Q-peak. A short sum up, although Fig.14 is more smooth, Fig.18 is chosen to be the template because it is closer to an ideal heart beat.

2.3 Matching filter

The mathematical principle of the matching filter is correlation, while the operation done by FIR filter is convolution. Therefore, the coefficient of a matching filter should be the time inverted template rather than the original one. Therefore, the next step is inverting the template and input it as the coefficients by the Python command:

```
coefficients = temp[::-1]
```

Based on the coefficients generated here, the matching filter can be initialized.

```
fir = FIR_filter(coefficients)
```

2.4 R-Peak detection

The code listed below operates the process of R peak detection, wrong detection removing as well as momentary heart rate calculation, while the last two will be discussed in next section.

```
flag = 1
index = [0,1]
detection = np.zeros(len(ECG_new))
HeartRate = np.zeros(len(ECG_new))
for i in range(len(ECG_new)):

    detection[i] = fir.dofilter(ECG_new[i])

    #Step five squart the result
    detection[i] = detection[i] * detection[i]


    #Step six R-Peak detection
    if detection[i] > 1e-11 and flag == 1:
        index[0] = index[1]
        index[1] = i

        interval = (index[1] - index[0])/fs

        #Wrong detections removing && momentary calculating
        if 60/interval < 130 and 60/interval >60:
            HeartRate[i] = 60/interval
            HeartRate[index[0] : index[1]] = np.linspace(HeartRate[index[0]],
                                                         HeartRate[index[1]],
                                                         index[1] - index[0])

        flag = 0

    if detection[i] < 1e-11 and flag == 0:
        flag = 1
```

The cleaned ECG signal is fed into the filter and the result is squared so that the difference between the R Peak and other undesired peaks is more obvious.The code that operates filtering and squaring in the *for* loop is :

```
detection[i] = fir.dofilter(ECG_new[i])

#Step five squart the result
detection[i] = detection[i] * detection[i]
```

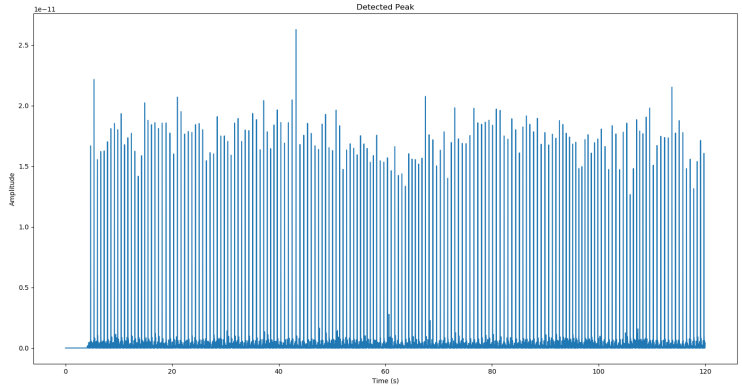

Figure 19: The Distribution of R-Peak

The distribution of R-Peak is shown in Fig.19, according to which, the threshold can be set to $1e-11$. A list of two elements is created to store the index of a detected heart beat and the previous one. And a *Flag* that helps to indicates previous data is initialized at 1.

```
flag = 1
index = [0,1]
```

In the process of traversing the data of the heartbeat detection result, a heart beat is detected if and only if amplitude exceeds the threshold and the *Flag* is 1. It is set to 1 when no data exceeds the threshold, therefore a heart beat is detected only at both conditions, that *Flag* is 1 and data exceeds threshold, are satisfied. When a heart beat is detected, the index [0] will store the old index, and the new index will be written to index [1]. In addition, the *Flag* is set to zero to indicates rising edge of a R peak is detected. The *Flag* is reset to 1 only when a data is less than threshold and *Flag* is 1, which indicates the flowing edge of a R peak. The code that implementing this R peak detection is listed below:

```
if detection[i] > 1e-11 and flag == 1:
    index[0] = index[1]
    index[1] = i

if detection[i] < 1e-11 and flag == 0:
    flag = 1
```

In this way, the index information of the last heartbeat is recorded at index [0] and the latest heartbeat position is stored at index [1].

2.5 Wrong Detection Removing and Momentary Calculation

Through the difference between the old and new indexes, the time interval between two heartbeats can be calculated by the command:

```
interval = (index[1] - index[0])/Fs
```

However,normal heartbeat of a walking human should lies within the range from 60 bmp to 130 bmp. Hence, the heart rate that not in that normal range is wrong detected. Hence, only valid heart beat rate will be displayed, which realized by a *if* statement. The code of this part is:

```
interval = (index[1] - index[0])/Fs

if 60/interval < 130 and 60/interval >60:
    HeartRate[i] = 60/interval
    HeartRate[index[0]:index[1]] = np.linspace(HeartRate[index[0]],
                                               HeartRate[index[1]],
                                               index[1] - index[0])
```

As a result, in Fig.20, except for the 0 bmp at the beginning due to the time delay and at the end due to no more peaks, this set of data has an momentary heart rate between 80 bmp and 100 bmp in the 120-second period. This is in line with the normal heart rate range.

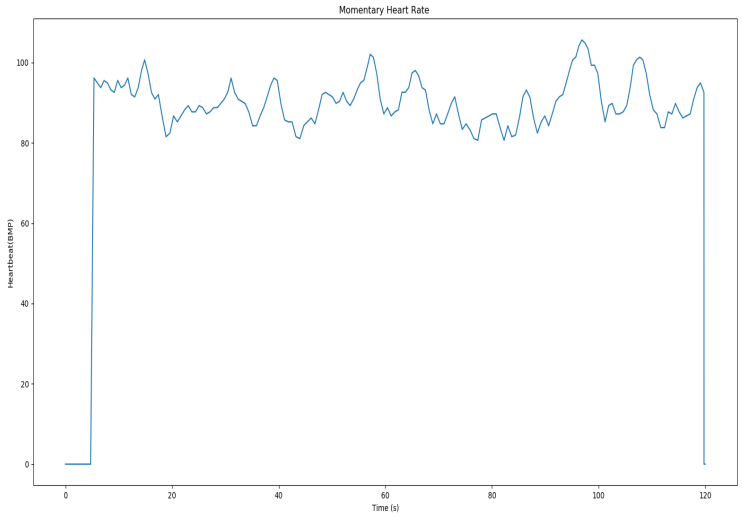

Figure 20: Momentary Heart Rate

Appendix

A Code of fir_filter.py

```
#=============================================================#
#Function of FIR filter
#=============================================================#

import numpy as np
import matplotlib.pyplot as plt

class RingBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [0]*capacity
        self.offset = -1

    def write(self, item):
        self.offset = (self.offset +1) % self.capacity
        self.queue[self.offset] = item

    def read(self):
        tmp = self.queue[self.offset]
        self.offset = (self.offset - 1)%self.capacity
        return tmp

    def display(self):
        print(self.queue)
        input()

class FIR_filter:
    def __init__(self, _coefficients):
        self.M = len(_coefficients)
        self.coefficients = _coefficients
        global R
        R= RingBuffer(self.M)

    def dofilter(self,v):
        R.write(v)
        result = 0

        for i in range(self.M):
            result = R.read() * self.coefficients[i] + result

        return result

def unittest():

    #input signal
    x = [1,0,0,0,0,0,0,0,0,0]

    #coefficient
    h = [0.5, 0.5]
```

```
    F = FIR_filter(h)

    y = [None]*len(x)
    for i in range(len(x)):
        y[i] = F.dofilter(x[i])

    print(y)

    plt.title('unittest')
    plt.plot(y,'b*')
    plt.xlabel('n')
    plt.ylabel('Amplitude')
    plt.show()



if __name__ == "__main__":

    unittest()
```

B Code of ecg_filter.py

```
#=========================================================================#
#Filter the ECG signal
#
#test file info : name : "ECG_ugrad_matric_7.dat"
#                 sample rate : 250 Hz
#=========================================================================#

import numpy as np
import matplotlib.pyplot as plt
from fir_filter import FIR_filter

def signal_processing(Fs, data):
    '''
    parameter Fs   : sample frequency
    parameter data : test data
    '''
    N = len(data)
    #step one design the proper ideal frequency response of filter
    M = 1000                #number of taps

    #resoulution of frequency is Fs/M = 0.25
    #remove the DC component and the 45Hz - 55Hz
    k1 = int(0.5/Fs * M)
    k2 = int(45/Fs * M)
    k3 = int(55/Fs * M)

    H = np.ones(M)
    H[0:k1+1] = 0
    H[k2:k3+1] = 0
    H[M-k1:M] = 0
    H[M-k3:M-k2] = 0

    #step two Inverse Fourier Transform
    h = np.real(np.fft.ifft(H))

    #step three mirror impulse response and shift to positive time
    h_new = np.zeros(M)
    h_new[0:int(M/2)] = h[int(M/2):M]
    h_new[int(M/2):M-1] = h[0:int(M/2)-1]

    #step four window the impulse response
    coefficients = h_new*np.hamming(M)

    #step five put the coefficients into FIR filter
    fir_filter = FIR_filter(coefficients)           #create the initail filter

    output = np.zeros(N)
    for i in range(N):
        output[i] = fir_filter.dofilter(data[i])

    return output
```

```
#=============================================================================#
#Main Function
#=============================================================================#

#Load data
ECG = np.loadtxt("ECG_ugrad_matric_7.dat")

#Basic Parameter
Fs = 250                                                    #sampling rate
N = len(ECG)                                                #sample number
period = N / Fs
time = np.linspace(0, period, N)


#input data into signal processing function
result = signal_processing(Fs= Fs, data= ECG)

np.savetxt('shortecg.dat', result)

#=============================================================================#
# ploting
plt.title('Original ECG')
plt.plot(time, ECG)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()

plt.title('ECG after FIR filtering')
plt.plot(time, result)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()
```

C Code of hr_detect.py

```
#=========================================================================#
#Heartbeat detection and real-time heart rate calculation
#
#test file info : name : "ecg_class.einthoven_II"
#                 sample rate : 250 Hz
#=========================================================================#

import numpy as np
import matplotlib.pyplot as plt
from fir_filter import FIR_filter
from ecg_gudb_database import GUDb

def signal_processing(Fs, data):
    '''
    parameter Fs   : sample frequency
    parameter data : test data
    '''
    N = len(data)
    #step one design the proper ideal frequency response of filter
    M = 1000                #number of taps

    #resoulution of frequency is Fs/M = 0.25
    #remove the DC component and the 45Hz - 55Hz
    k1 = int(1/Fs * M)
    k2 = int(45/Fs * M)
    k3 = int(55/Fs * M)

    H = np.ones(M)
    H[0:k1+1] = 0
    H[k2:k3+1] = 0
    H[M-k1:M] = 0
    H[M-k3:M-k2] = 0

    #step two Inverse Fourier TransformZ
    h = np.real(np.fft.ifft(H))

    #step three mirror impulse response and shift to positive time
    h_new = np.zeros(M)
    h_new[0:int(M/2)] = h[int(M/2):M]
    h_new[int(M/2):M-1] = h[0:int(M/2)-1]

    #step four window the impulse response
    coefficients = h_new*np.hamming(M)

    #step five put the coefficients into FIR filter
    fir_filter = FIR_filter(coefficients)           #create the initail filter

    output = np.zeros(N)
    for i in range(N):
        output[i] = fir_filter.dofilter(data[i])
```

```
        if i < 1000:
            output[i] = 0


    return output


#=========================================================================#
#Main Function
#=========================================================================#

#Subject number to load
subject_number = 7

#Experiment to load
experiment = 'walking'

#Creating class which loads the experiment
ecg_class = GUDb(subject_number, experiment)

#Load the test data set
ECG = ecg_class.einthoven_II


#Basic Parameter
Fs = 250                                                    #sampling rate
N = len(ECG)                                                #sample number
period = N / Fs
time = np.linspace(0, period, N)


#Step one pre-filtering: Remove DC and 50Hz interference
ECG_new = signal_processing(Fs, ECG)

#Step two template
temp = ECG_new[10020:10100]

#Step three create FIR coefficients by the time inverted template
coefficients = temp[::-1]


#Step four filter the ECG signal
fir = FIR_filter(coefficients)

flag = 1
index = [0,1]
detection = np.zeros(len(ECG_new))
HeartRate = np.zeros(len(ECG_new))
for i in range(len(ECG_new)):

    detection[i] = fir.dofilter(ECG_new[i])

    #Step five squart the result
```

```
    detection[i] = detection[i] * detection[i]


    #Step six R-Peak detection
    if detection[i] > 1e-11 and flag == 1:
        index[0] = index[1]
        index[1] = i

        interval = (index[1] - index[0])/Fs

        #Wrong detection removing && momentary calculating
        if 60/interval < 130 and 60/interval >60:
            HeartRate[i] = 60/interval
            HeartRate[index[0] : index[1]] = np.linspace(HeartRate[index[0]],
                                                         HeartRate[index[1]],
                                                         index[1] - index[0])

        flag = 0

    if detection[i] < 1e-11 and flag == 0:
        flag = 1


#=============================================================================#
# ploting result of Heartbeat detecting and Momentary heart rate
plt.plot(time, detection)
plt.title("Detected Peak")
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()

plt.plot(time, HeartRate)
plt.title("Momentary Heart Rate ")
plt.xlabel('Time (s)')
plt.ylabel('Heartbeat(BMP)')
plt.show()
```