Assignment 2, Digital Signal Processing: FIR Filters University of Glasgow School of Engineering

Mustafa Biyikli & Katarzyna Lenard

Introduction

Electrocardiogram (ECG) is a continuous signal representing the electrical activity of the heart muscle typically possessing peak amplitudes below 1mV. Therefore, it needs to be amplified and discretised for any digital signal processing application. An ECG signal needs to be processed as it is contaminated with mains interference consisting of a 50Hz sinusoid, its harmonics and electromyography (EMG) noise from the neighbouring muscle groups due to patient movement (Tayel et al., 2017). In most applications concerned with the well-being of a patient, ECG signals need to be processed in real-time proving non-causal methods of signal processing to be ineffective for this purpose. Therefore, this study investigates a method of causal signal processing to filter undesired frequency ranges from the ECG spectrum and detect the ECG R-peaks to calculate the momentary heart rate. The objectives were to design a finite impulse response (FIR) band-stop filter using an efficient ring buffer, written in Python, to process M number of samples at a time with a delay of M/2 samples.

Methods

Data Acquisition

The ECG of a 23-year-old female was recorded, following a short running exercise intended to increase the heart rate. The self-adhesive electrodes were placed on the shoulders and hips while the subject was laying on a flat surface to reduce undesired muscle noise. An amplifier with a gain value of 500 and a 24-bit analog-to-digital (A/D) converter with an input range of -1.325V to 1.325V were used. The recording consisted of 3 channels performed at a sampling rate of 1kHz. Channel Einthoven II was chosen as it had the highest amplitude.

Frequency Resolution & Coefficients

Initial step was to determine the number of taps required for filtering which determined the frequency resolution of the filter. Given an average heart rate of 1 beat per second, the fundamental frequency of the ECG was determined to be 1Hz. Therefore, the number of taps were set to 2000 to achieve a frequency resolution of 0.5Hz as shown in equation 1.

$$\Delta F = \frac{Fs}{M} \tag{1}$$

Where, ΔF is the frequency resolution, Fs is the sampling rate and M is the number of taps. To create the coefficients, an ideal frequency response of the filter needed to be designed according to the frequency ranges of the undesired noise. Therefore, the frequency spectrum of the recorded data was analysed by plotting the Fourier Transform of the time domain. Figure 1 (a) and (b) show the real part of the mains interference and its harmonics to be filtered, respectively.

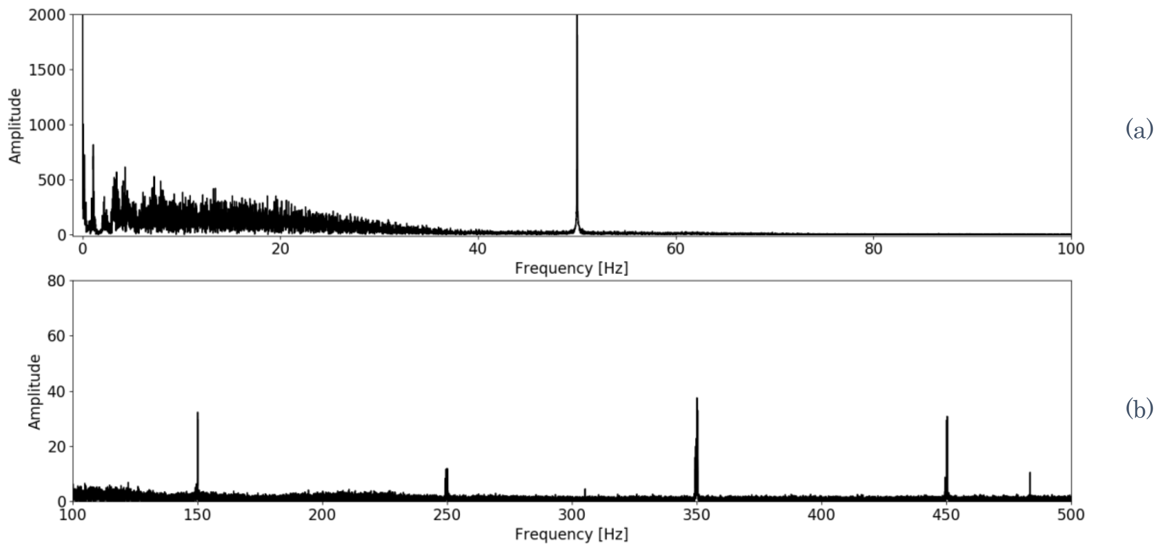


Figure 1, Mains interference and its harmonics to be filtered

Therefore, the ideal frequency response of the filter was defined using amplitudes of 1 and 0 for pass and stop bands respectively. 50Hz and its harmonics up to Nyquist frequency (Fs/2) were removed using a moving range of 10Hz. Additionally, frequencies from 0Hz to 0.5Hz were removed to get rid of the DC noise. Complex conjugates of the real part were treated equally in all operations following equation 2 where N is the total number of samples.

$$x^*(k) = x(-k) = x(N-k) \tag{2}$$

The ideal frequency response of the filter is shown in Figure 2.

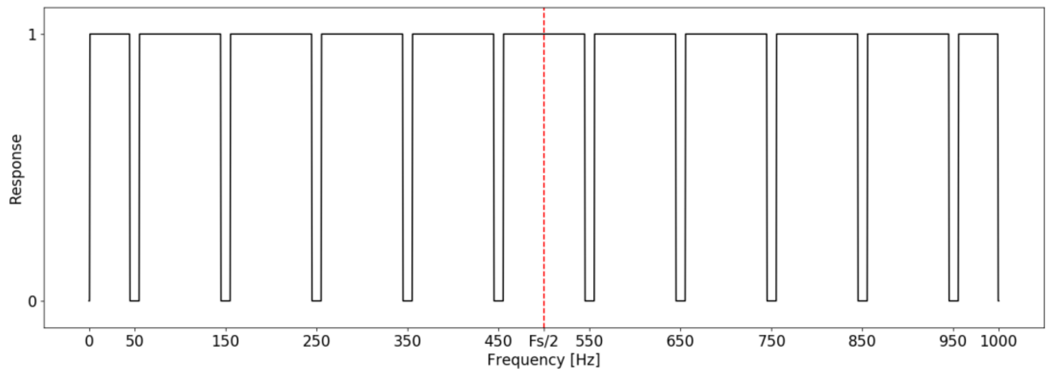


Figure 2, Ideal frequency response of the filter

The coefficients were obtained from the ideal frequency response and the performance was improved by applying an appropriate window function to reduce the ripples. Hamming, Hanning, Bartlett and Blackman window functions were applied to the impulse response and their effects were examined. The objectives were to remove the ripples whilst keeping a narrow transition width and good stop-band rejection. Figure 3 shows a performance comparison of the previously mentioned window functions on the 50Hz rejection band.

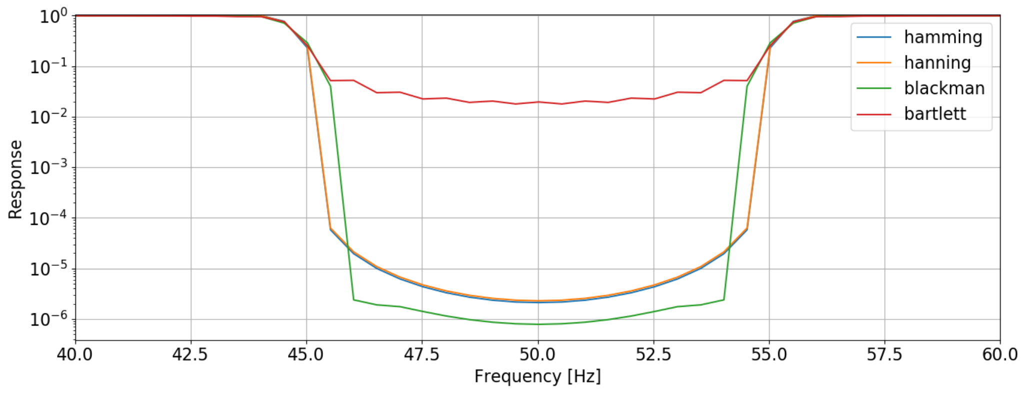


Figure 3, Performance comparison of window functions

Having the narrowest transition width and a good stop band rejection, Hanning window was selected. The impulse response was multiplied with the window function and the new coefficients were obtained following equation 3.

$$h(n)_{new} = h(n).w(n) \tag{3}$$

DC removal was reapplied after windowing as the transition width of the window function was larger than 0.5Hz. Therefore, the coefficients were transformed back to the frequency domain and frequencies from 0Hz to 0.5Hz were removed. The resulting response was transformed back to time domain, mirrored and shifted into positive time and truncated by the window function. This method ensured a sharp response for the DC removal while eliminating the ripples for the band-stop frequencies. Figure 4 shows the impulse response in time domain.

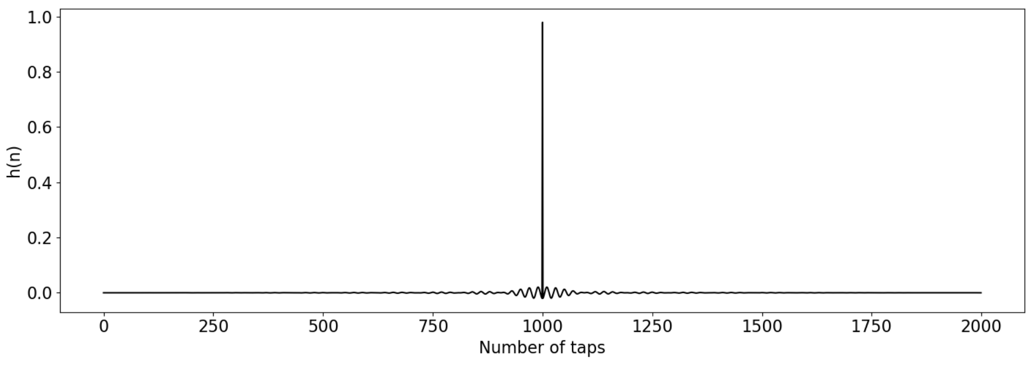


Figure 4, Impulse response of the filter; time domain

FIR filter class

A FIR filter class was created in Python, using a ring buffer and python slicing operations. The ring buffer is an empty array of the same length as the coefficients and it adds values one by one to the array. Once the buffer is full, it overwrites the last value with a new one. In the FIR filter class, each time a new value was added, the buffer array was multiplied with the coefficients in the correct order. Figure 5 presents how the FIR filter works.

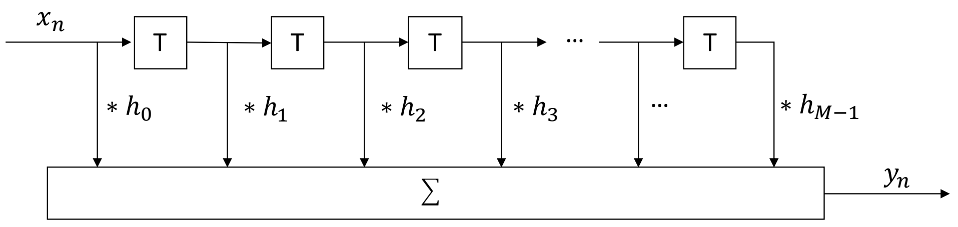


Figure 5 FIR filter

Where x_n is input signal, T represents delay, h are the coefficients, M is the total length of the array and y_n is the output value.

In simple terms, the newest added value in the buffer, called head, had to be multiplied with first coefficient, the second value with the second coefficient and so on, until the oldest value (called tail) was multiplied with the last coefficient. The problem with programming the buffer was that virtually it cannot have a circular shape. Instead it is presented as an array, in which the index of the head is moving in the right direction and wraps around the array after reaching the last index. This means that, for the correct and fast coefficient multiplication two operations had to be performed on the buffer to avoid applying a loop function. The buffer values had to be sliced and swapped, in such a way that buffer starts with the tail and ends with the head. Then the values were inverted, so the two arrays can be multiplied fully in one command. The whole process is shown in a block diagram in Figure 6.

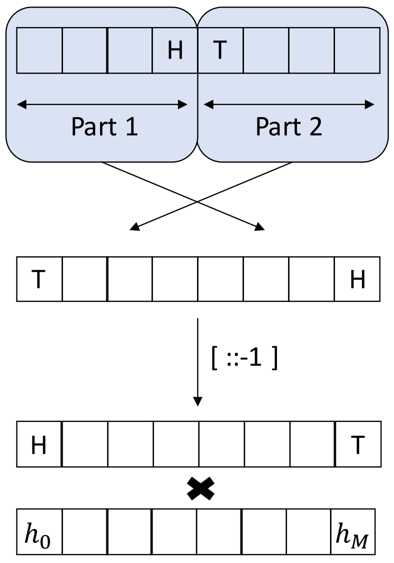


Figure 6 Process undertaken to achieve FIR filter properties. H and T represent head and tail of the ring buffer, respectively, and the bottom array with h 0 to M represents coefficients.

The filter was fed with one sample at a time to imitate real-time processing. The output, after filtering the entire signal, resulted in a delay of M/2 samples, which is 1 second in this case.

Methods – Match Filter

Template

Signal detection can be accomplished by a template matching process. Match filter is using a previously created template in a similar manner as the FIR filter does with its coefficients – using a ring buffer and slicing operations. The signal is sample by sample multiplied with coefficients of the template. When the signal matches with the template, a pulse is generated and the signal is amplified, making it easier to detect using the same threshold for all R-waves.

The coefficients of the template are calculated analytically from a mathematical formula. Several functions were tested in this study, and only one, most resembling the heartbeat was chosen. All templates had the same length of around 1/4 of the sampling frequency, as 1 heartbeat takes around 1 second and the R-wave can easily fit within 1/4 of that time.

Tested functions included Morlet Wavelet, Mexican Hat Wavelet and Gaussian Derivative Wavelet. The equations for each function are shown in equations 4, 5 and 6, respectively.

$$\Psi(t) = e^{-\frac{t^2}{2}}\cos(5t) \tag{4}$$

$$\Psi(t) = \frac{2}{\sqrt{3}\sqrt[4]{\pi}}e^{-\frac{t^2}{2}}(1-t^2) \tag{5}$$

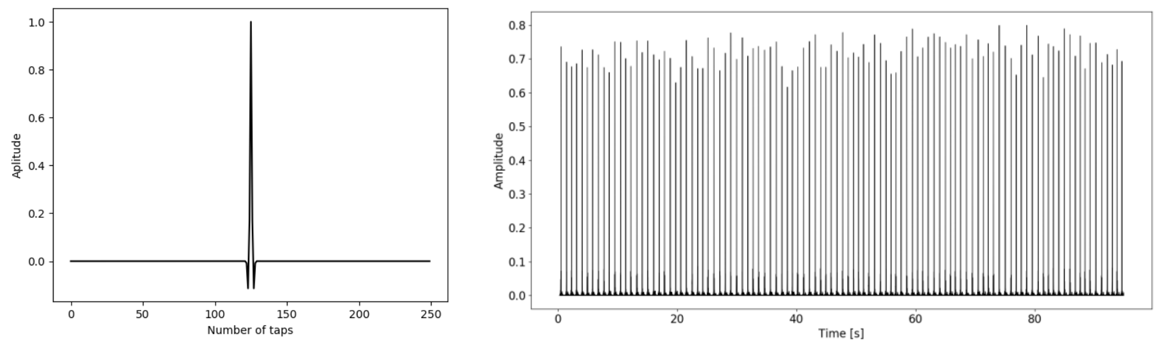
$$\Psi(t) = Ce^{-t^2} \tag{6}$$

Where C in equation 3 is an order-dependant normalisation constant. For this study C was chosen to be 1.

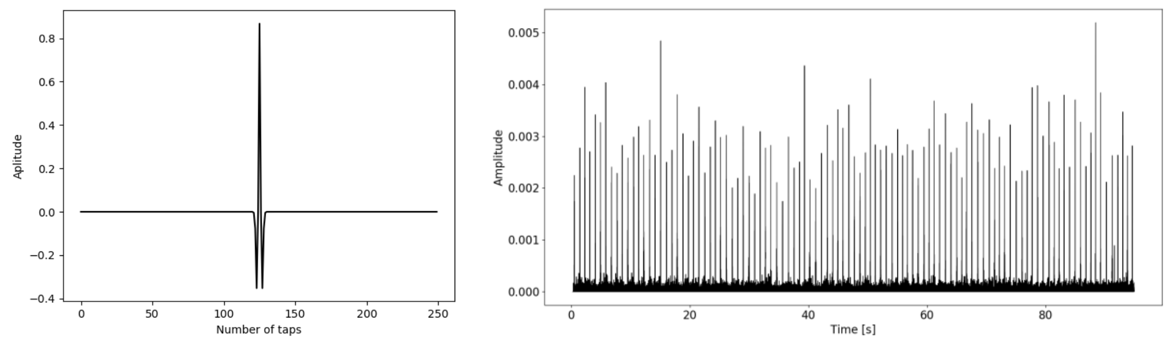
Every function starts from 0 point and is mirrored on the negative axis. Therefore, to make the coefficients causal, the entire function had to be shifted into positive time. After that procedure the coefficients were ready to be used in the matched filter.

A prefiltered signal, with DC removed, was fed one sample at a time into the matched filter.

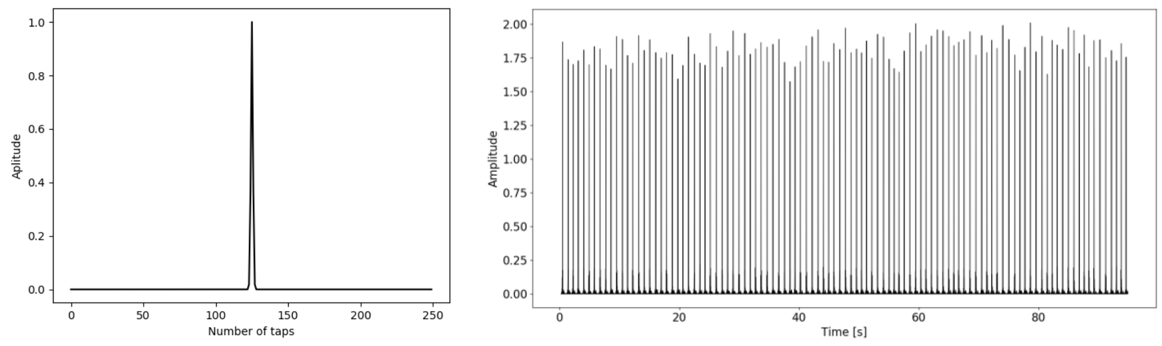
In order to choose one of the functions, they were plotted and fed to the matched filter for a performance comparison. Figure 7 (a) shows Morlet function, Figure 7 (b) Mexican Hat and Figure 7 (c) a Gaussian Derivative.



a) Morlet Wavelet with matched filtered signal



b) Mexican Hat Wavelet with matched filtered signal



c) Gaussian Derivative Wavelet with matched filtered signal

Figure 7 Plotted wavelet functions for template coefficients along with their performance.

As it can be seen from Figure 7, Mexican Hat decreased the amplitude and made the signal not usable, whereas the Gaussian Derivative achieved the highest amplitude, which means it was the closest to resemble the ECG R-wave from all tested functions. Morlet function was not chosen due to the amplitude not reaching 1. Therefore, having achieved the best performance, Gaussian Derivative was chosen as the template.

To justify the choice of function, it was found that Madeiro et al. (2012) also used a Gaussian function to generate a mathematical model of an ECG and according to Tlili et al. (2016), the Gaussian function is the best technique to generate a realistic ECG signal.

Momentary Heartrate

In this stage, the momentary heartrate will be calculated from the time intervals between adjacent heartbeats.

In order to make sure that one threshold could cover all R-peaks and exclude bogus ones, the signal-to-noise ratio (SNR) had to increase. This was accomplished by squaring the already amplified matched filter output. Usually, the noise should be less than an amplitude of 1, which becomes even smaller after the squaring operation, whereas, the desired signal characteristics above an amplitude of 1 become greater. This provides an ideal SNR for choosing a threshold.

The heartbeats were detected, by employing an adaptive threshold, which should be around ½ of the highest peak. In this case, the threshold was set to an amplitude of 1.25. The beats were detected from the rising side of the R-wave instead of the peak, as the maximum value of the R-wave could have been the result of noise. The detections were performed within 10% of the threshold, as the samples were not equally distributed among an exact threshold value. Thanks to increased SNR and correctly employed threshold, bogus detections were avoided and the momentary heartrate was calculated with no issues throughout the entire signal. This was done using equation 7.

$$momentary_{hr} = \frac{60}{\Delta t} = \frac{60}{\frac{n}{Fs}} \tag{7}$$

Where Δt is the time interval between the detected beats, which corresponds to the number of samples, n, divided by the sampling rate, Fs. The momentary heartrate in beats-per-minute against beat index was plotted. The block diagram of the process is shown in Figure 8.

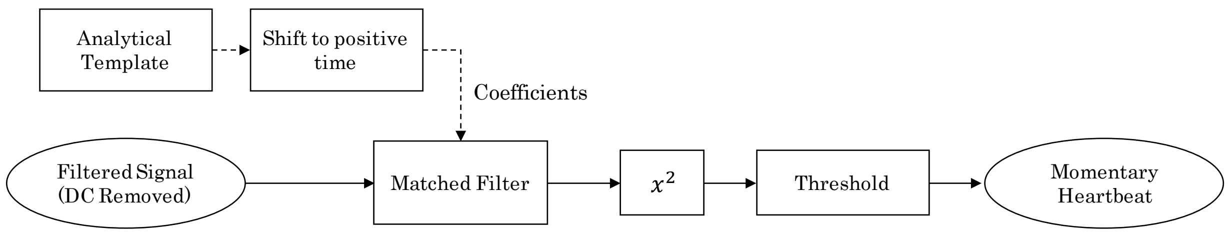
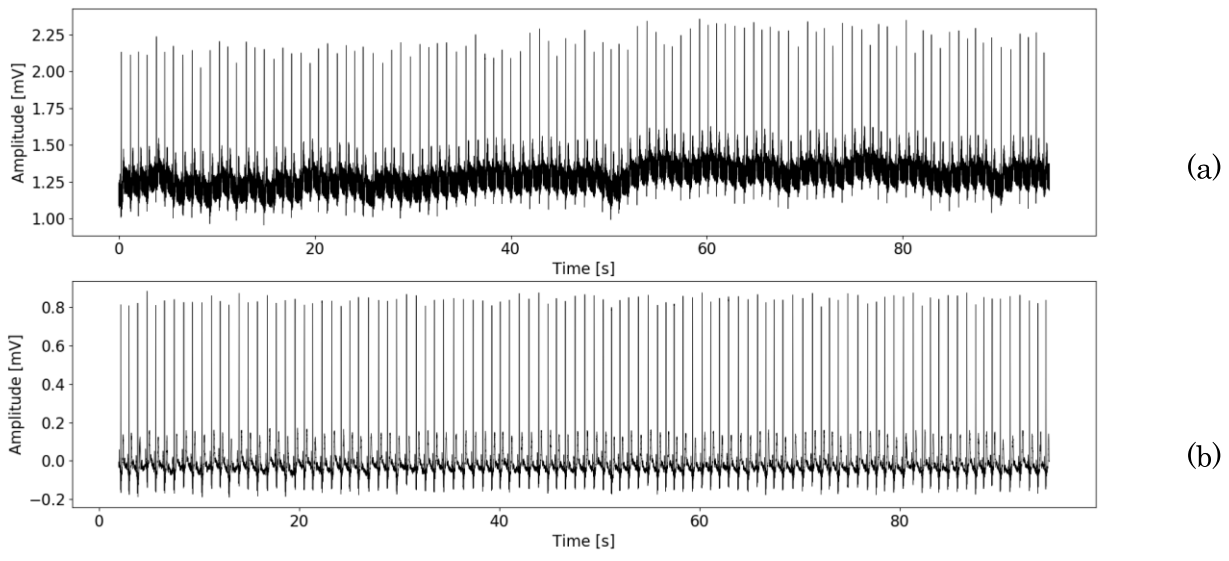


Figure 8 Block diagram of heartbeat detection and momentary heartbeat calculation process.

Results

After the data was imported to Python, channel Einthoven II was selected as it possessed the highest amplitude and a graph of the raw data was plotted. Figure 9 (a) shows a plot of the raw data in time domain and Figure 9 (c) shows 5 full heart beats. Both plots show the effect of DC and mains interference noise. Figure 9 (b) and (d) are the plots after the signal was processed by the derived FIR filter class. Comparing Figure 9 (a) and (b), the effect of DC removal can be seen. Comparing Figure 9 (c) and (d), the effect of mains interference removal and a delay of M/2 taps (1sec) can be seen.



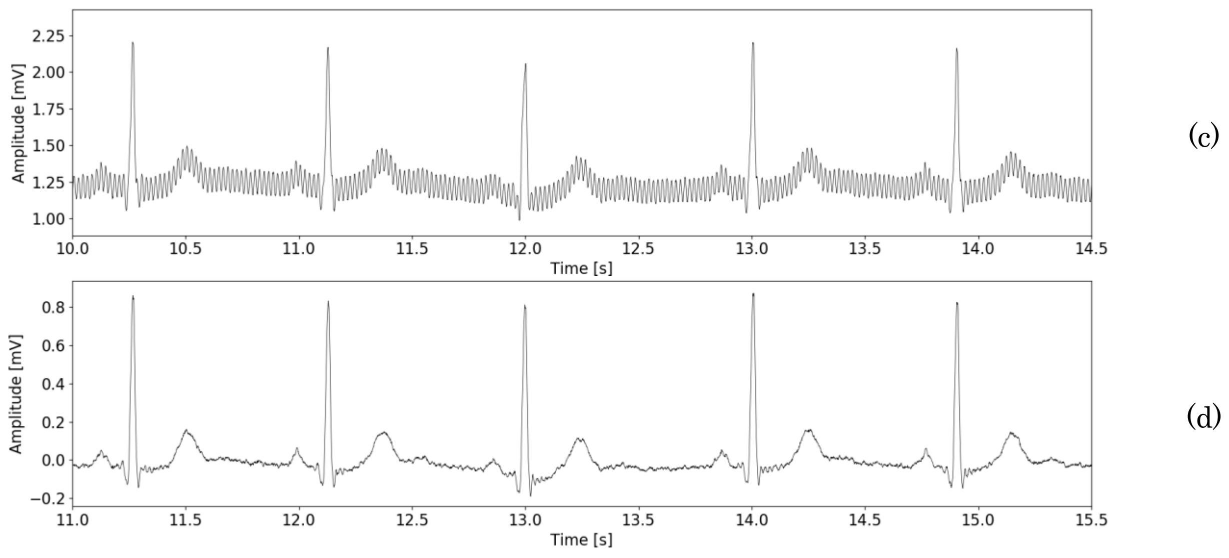


Figure 9, Einthoven II, Raw & Clean comparison

For heartrate detection, the signal was filtered using the matched filter class. A Gaussian wavelet was used as a template for amplifying the R-peaks and the matched filter output was squared for an improved SNR. An adaptive threshold was applied only on one side of the R-peak removing any bogus detections. Figure 10 shows the matched filter output with the detections.

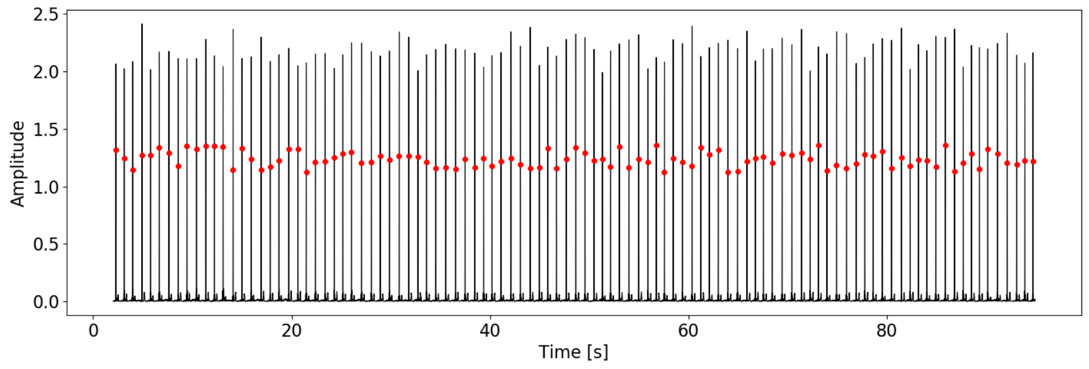


Figure 10, Matched filter output & R-peak detection

Taking the intervals between the detections, the momentary heartrate was calculated over the full signal. Figure 11 shows the momentary heartrate for each heartbeat.

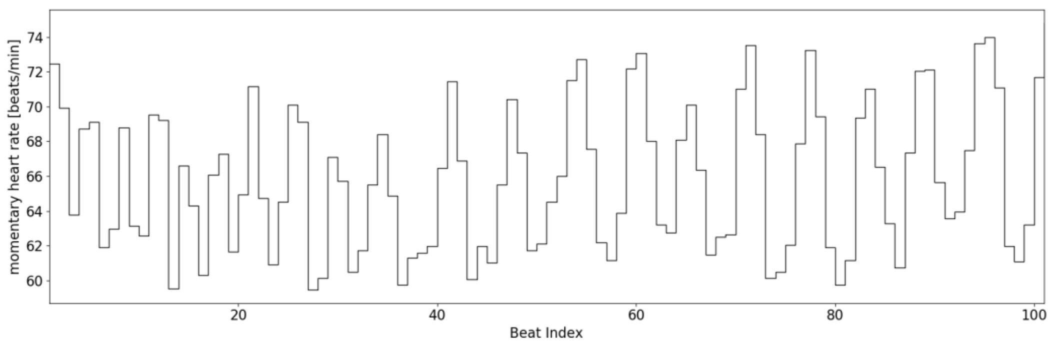


Figure 11, Momentary heartrate over the full signal

Discussion

As it can be seen in Figure 9 (a) and (c), the raw ECG wasn't saturated with excessive amounts of mains interference. This was due to selecting a location away from the mains cables for recording the ECG. Additionally, the EMG noise from the neighbouring muscles was minimised by asking the subject to lay comfortably. The electrodes were placed on the shoulders instead of the wrists ensuring the best possible raw data SNR and an accurate amplitude due to measurements being taken closer to the heart.

The proposed method of filtering written in Python took considerably longer than the C/C++ equivalent, nevertheless, an effective usage of a ring buffer was employed to ensure the best possible computational efficiency. The ring buffer was implemented to avoid shifting the data and iterating through loops. Instead, Python slicing operations were used for improved performance. As it can be seen from Figure 9 (c) and (d), FIR filter output was delayed by M/2 number of samples, in this case by 1 second. A satisfactory filter performance was achieved as all ECG characteristics could easily be determined from the clean signal. This is shown in Figure 12.

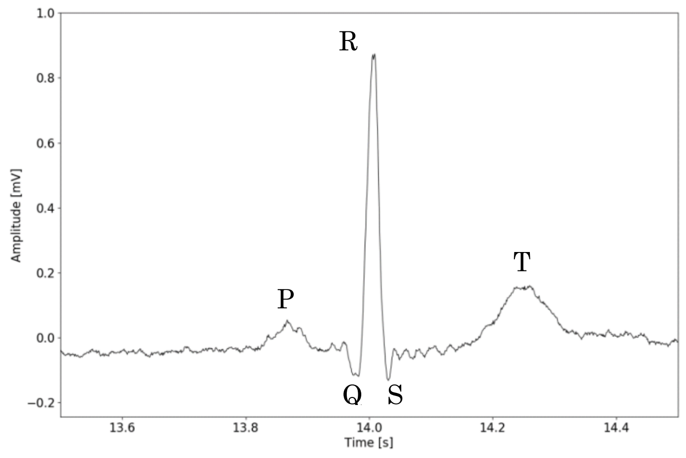


Figure 12, ECG characteristics

The matched filter output provided an exaggerated R-peak for an easy detection. The threshold applied worked very well neglecting any bogus detections. Having tested several wavelets with similar characteristics to an R-peak, the Gaussian derivative was chosen. There may be functions better suited for R-peak detection which would require further investigation, however, for the purposes of this study, a satisfactory matched filter output was obtained. The adaptive threshold, and successful detections can be seen in Figure 10.

Momentary heart rate for each beat index over the full ECG recording was derived from the time intervals between the detections. Although the subject had exercised prior to the recording, there was an upward trend visible in the momentary heartrates. This can be seen in Figure 11 and may be caused by the long set-up time for the recording during which the subject could have rested. The upward trendline could be explained by the changes in breathing or anxiety caused by the procedure.

Having the benefit of making linear phase possible, the FIR filter required a lot of memory. The proposed Python based FIR filter proved to be inefficient for any practical causal signal processing as it requires a significant amount of computational power. Larger number of taps are required for higher sampling rates which increases the computing time and introduces a larger delay.

Conclusion

In this study, DC and mains interference noise were filtered from an ECG signal by designing and implementing a Python based FIR filter. An ideal frequency response of the filter was produced and transformed into time domain for creating the FIR coefficients. An efficient ring buffer was designed, and slicing operations were used for efficient computing. Similarly, a Matched filter was designed and implemented for heartrate detection. A Gaussian wavelet was used for R-peak amplification and the filter output was squared for an improved SNR. An adaptive threshold was applied for detecting the R-peaks and bogus detections were avoided. Momentary heartrates were calculated by taking the detection intervals. The filter performance was slow compared to C/C++ implementations due to Python's method of handling data. Although the FIR filter provided a linear phase, it also required a large amount of memory with the increased number of taps. Therefore, this method of filtering proves to be inefficient for any practical real time filtering when implemented in Python. An infinite impulse response filter would provide a better computational efficiency and would need to be implemented for a better understanding of causal methods of ECG signal processing.

References

Madeiro, J. P. V., Nicolson, W. B., Cortez, P. C., Marques, J. A. L., Vázquez-Seisdedos, C. R., Elangovan, N., Ng, G. A. & Schlindwein, F. S. 2012. New approach for T-wave peak detection and T-wave end location in 12-lead paced ECG signals based on a mathematical model. *Medical Engineering and Physics,* 35**,** 1105-1115.

Tayel, M. B. P., Eltrass, A. S. P. & Ammar, A. I. M. 2017. A new multi-stage combined kernel filtering approach for ECG noise removal. *Journal of Electrocardiology,* 51**,** 265-275.

Tlili, M., Maalej, A., Ben Romdhane, M., Rivet, F., Dallet, D. & Rebai, C. Mathematical modeling of clean and noisy ECG signals in a level-crossing sampling context. 2016 2016. IEEE, 359-363.

Appendix A – FIR Filter Class

```
import numpy as np
class FIR_filter:
    def __init__(self, coefficients):
        self.size = len(coefficients)
        self.coefficients = coefficients
        self.buffer = np.zeros(self.size)
        self.index = 0
    def dofilter(self, data):
        self.buffer[self.index] = data
        self.output = 0
        # Slice the ring buffer into two parts and invert them
        self.temporary = [*self.buffer[0:self.index][::-1],
*self.buffer[self.index:self.size][::-1]]
        # Multiply the full buffer with coefficients and add all = output
        self.output = np.sum(self.temporary * self.coefficients)

        if self.index == self.size - 1:
            self.index = 0
        else:
            self.index += 1
        return self.output
```

Appendix B – Matched Filter Class

```
import numpy as np

class Matched_filter:

    def __init__(self, coefficients):
        self.size = len(coefficients)
        self.coefficients = coefficients
        self.buffer = np.zeros(self.size)
        self.index = 0
        self.peaks = []; self.momentary_hr = []; self.thresholds = []
        self.first = 0; self.last = 0

    def hr_detector(self, data):
        self.buffer[self.index] = data
        self.output = 0
        # Slice the ring buffer into two parts and invert them
        self.temporary = [*self.buffer[0:self.index][::-1],
*self.buffer[self.index:self.size][::-1]]
        # Multiply the full buffer with coefficients and add all = output
        self.output = np.sum(self.temporary * self.coefficients)

        if self.index == self.size - 1:
            self.index = 0
        else:
            self.index += 1
        return self.output**2

    def threshold_finder(self, signal, Fs, threshold):
        self.count = 0
        for i in range(len(signal)):
            if (signal[i] > threshold-(0.10*threshold)) and (signal[i] <
threshold+(0.10*threshold)) and (self.count > Fs/4):
                self.thresholds.append(i)
                self.count = 0
            elif (signal[i] == threshold) and len(self.thresholds) == 0:
                self.thresholds.append(i)
                self.count = 0
            self.count += 1
        for i in range(1, len(self.thresholds)):
            self.momentary_hr.append(self.thresholds[i] - self.thresholds[i-1])
        return self.thresholds, self.momentary_hr

    def peak_finder(self, signal, threshold):
        for i in range(len(signal)):
            if signal[i] > threshold:
                self.last = i
            else:
                if (self.last == i-1) and (self.last > self.first):
                    for r in range(self.first, self.last):
                        if signal[r] == max(signal[self.first:self.last]):
                            self.max_index = r
                    self.peaks.append(self.max_index)
                self.first = i
        for i in range(1, len(self.peaks)):
            self.momentary_hr.append(self.peaks[i] - self.peaks[i-1])
        return self.peaks, self.momentary_hr
```

Appendix C – Wavelets Class

```
import numpy as np

class Wavelet:
    def __init__(self, number_of_taps):
        self.M = number_of_taps
    def Morlet(self):
        mor1 = np.empty(self.M)
        mor1_inversed = np.empty(self.M)
        for t in range(-int(self.M/2), int(self.M/2), 1):
            mor1[t] = np.exp(-(t**2)/2)*np.cos(5*t)
        mor1_inversed[0:int(int(self.M/2))] = mor1[int(int(self.M/2)):self.M]
        mor1_inversed[int(int(self.M/2)):self.M] = mor1[0:int(int(self.M/2))]
        return mor1_inversed
    def Mexican_Hat(self):
        mex_hat = np.empty(self.M)
        mex_hat_inversed = np.empty(self.M)
        for t in range(-int(self.M/2), int(self.M/2), 1):
            mex_hat[t] = (2/(3**(1/2)*(np.pi**(1/4))))*np.exp(-(t**2)/2)*(1-t**2)
        mex_hat_inversed[0:int(int(self.M/2))] =
mex_hat[int(int(self.M/2)):self.M]
        mex_hat_inversed[int(int(self.M/2)):self.M] =
mex_hat[0:int(int(self.M/2))]
        return mex_hat_inversed
    def Gaussian(self, C=1):
        gaus = np.empty(self.M)
        gaus_inversed = np.empty(self.M)
        for t in range(-int(self.M/2), int(self.M/2), 1):
            gaus[t] = C*np.exp(-(t**2))
        gaus_inversed[0:int(int(self.M/2))] = gaus[int(int(self.M/2)):self.M]
        gaus_inversed[int(int(self.M/2)):self.M] = gaus[0:int(int(self.M/2))]
        return gaus_inversed
```

Appendix D – ECG Filter

```
import numpy as np; import matplotlib.pyplot as plt; import time
# Import our modules
import FIR_filter

time_total = time.time()    # This will be used to print the total execution time
# Load the data, select time & data arrays & set some variables
data = np.loadtxt('ecg.dat')
ecg = data[:, 2] * ((1.325*2)/2**24)    # Potential difference [mV]
T = data[:, 0] / 1000                   # in seconds [s]
# Plot the raw data
plt.subplots(); plt.plot(T, ecg, linewidth=0.8, color='k')
plt.title('Raw ECG Signal [Einthoven II]'); plt.ylabel('Amplitude [mV]');
plt.xlabel('Time [s]')

M = 2000        # Number of Taps (double Fs)
Fs = 1000       # Sampling rate [Hz]

# Ideal Frequency Response
X = np.ones(M)
for k in np.arange(45, int(Fs/2), 100):
    k1 = int(k/Fs*M); k2 = int((k+10)/Fs*M) # A moving frequency range of 10Hz
    X[k1:k2+1] = 0                          # Set the ranges in real part to 0
    X[M-k2:M-k1+1] = 0                      # Set the complex conjgate ranges to 0

x = np.fft.ifft(X)
x = np.real(x)

# Slice & Swap & Shift into positive time
h = np.zeros(M)
h[0:int(M/2)] = x[int(M/2):M]
h[int(M/2):M] = x[0:int(M/2)]

# Window function
h = h * np.hanning(M)   # Choice justified in the report

# DC Removal (sharp DC removal response)
X = np.fft.fft(h)
X[0:int(0.5/Fs*M)] = 0
h = np.real(np.fft.ifft(X))

time1 = time.time()     # Seconds passed since epoch (1/1/1970 00:00:00)

# FIR Filter usage
```

```
f = FIR_filter.FIR_filter(h)
y = np.empty(len(ecg))
for i in range(len(ecg)):
    y[i] = f.dofilter(ecg[i])

y = y[M:]
plt.subplots(); plt.plot(T[M:], y, linewidth=0.8, color='k')
plt.title('FIR Filter Output'); plt.ylabel('Amplitude [mV]'); plt.xlabel('Time
[s]')
# Time the FIR Filter
print('{} {} {}'.format('FIR Filter; took', time.time()-time1, 'seconds'))
np.savetxt('ecg_clean.dat', y)

print('{} {} {}'.format('Total Execution; took', time.time()-time_total,
'seconds'))
plt.show()
```

Appendix E – Heartrate Detection

```
import numpy as np; import matplotlib.pyplot as plt; import time
# Import our modules
import Matched_filter; import Wavelets
time_total = time.time()    # This will be used to print the total execution time

####    Testing the wavelets    ####

Fs = 1000;              # Sampling rate [Hz]
M = 2000;               # This was used for the FIR Filter and is needed here
M_template = int(Fs/4); # Number of taps for the template
y = np.loadtxt('ecg_clean.dat')
T = np.linspace(0,  len(y)/Fs, num=len(y))  # Time [s]

wavelet = Wavelets.Wavelet(M_template) # Our module, see Wavelets.py for
derivations

""" MORLET WAVELET """
morlet = wavelet.Morlet()
plt.subplots(); plt.subplot(231); plt.plot(morlet, linewidth=0.8, color='k');
plt.title("Morlet Wavelet")
plt.suptitle('Wavelet Performance Comparison')
plt.ylabel('Amplitude'); plt.xlabel('Number of taps')

d = Matched_filter.Matched_filter(morlet)
detector = np.empty(len(y))
for i in range(len(y)):
    detector[i] = d.hr_detector(y[i])
plt.subplot(234); plt.plot(T, detector, linewidth=0.8, color='k')
plt.ylabel('Amplitude'); plt.xlabel('Time [s]')

""" MEXICAN HAT WAVELET """
mexican_hat = wavelet.Mexican_Hat()
plt.subplot(232); plt.plot(mexican_hat, linewidth=0.8, color='k');
plt.title("Mexican Hat Wavelet")
plt.ylabel('Amplitude'); plt.xlabel('Number of taps')
```

```
d = Matched_filter.Matched_filter(mexican_hat)
detector = np.empty(len(y))
for i in range(len(y)):
    detector[i] = d.hr_detector(y[i])
plt.subplot(235); plt.plot(T, detector, linewidth=0.8, color='k')
plt.ylabel('Amplitude'); plt.xlabel('Time [s]')

""" GAUSSIAN DERIVATIVE """
gaussian = wavelet.Gaussian()
plt.subplot(233); plt.plot(gaussian, linewidth=0.8, color='k');
plt.title("Gaussian Derivative")
plt.ylabel('Amplitude'); plt.xlabel('Number of taps')

time1 = time.time()
d = Matched_filter.Matched_filter(gaussian)
detector = np.empty(len(y))
for i in range(len(y)):
    detector[i] = d.hr_detector(y[i])
plt.subplot(236); plt.plot(T, detector, linewidth=0.8, color='k')
plt.ylabel('Amplitude'); plt.xlabel('Time [s]')
plt.subplots_adjust(left=0.05, right=0.95, top=0.9, bottom=0.1, wspace=0.3,
hspace=0.3)
# Time the Matched Filter
print('{} {} {}'.format('Matched Filter; took', time.time()-time1, 'seconds'))

""" HEARTRATE DETECTION w/GAUSSIAN """
# Matched filter heartrate detection
thresholds, momentary_hr = d.threshold_finder(detector, Fs, threshold=1.25)
thresholds_time_domain = np.empty(len(thresholds))

# Get peak indexes in time domain
for i in range(len(thresholds)):
    thresholds_time_domain[i] = thresholds[i]/Fs

# Calculate the momentary heart rate in bpm
for i in range(len(momentary_hr)):
    momentary_hr[i] = 60/(momentary_hr[i]/Fs)

# Plot the mathced filter output w/detections
plt.subplots(); plt.plot(T, detector, linewidth=0.8, color='k');
plt.plot(thresholds_time_domain, detector[thresholds], 'ro', markersize=3)
plt.title('Matched Filter Output'); plt.ylabel('Amplitude'); plt.xlabel('Time
[s]')

# Plot the momentary heartrate over the entire signal
plt.subplots();plt.step(np.linspace(1, len(momentary_hr),
num=len(momentary_hr)),momentary_hr, linewidth=0.8, color='k')
plt.ylabel('momentary heart rate [beats/min]');plt.xlabel('Beat
Index');plt.xlim(1, len(momentary_hr))

print('{} {} {}'.format('Total Execution; took', time.time()-time_total,
'seconds'))
plt.show()
```