Charlie Maclean

Synthesis of Heart-Rate Detection Methods

Computer Science Tripos – Part II

King's College

April 16, 2020

Proforma

Name:	**Charlie Maclean**
College:	**King's College**
Project Title:	**Synthesis of Heart-Rate Detection Methods**
Examination:	**Computer Science Tripos – Part II, July 2020**
Word Count:	**TODO**[1]
Project Originator:	Dr Robert Harle
Supervisor:	Dr Robert Harle

Original Aims of the Project

To research and implement the detection of heart rate from smartwatch sensors. To investigate the effectiveness of a selection of filters and peak finding algorithms. To use accelerometer data to find motion artifacts within the data, and compare methods of removing these artifacts.

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

Learning how to incorporate encapulated postscript into a LaTeX document on both Ubuntu Linux and OS X.

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Charlie Maclean of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

1 Introduction **9**

2 Preparation **11**
- 2.1 Photoplethysmography (PPG) 11
- 2.2 Techniques 12
- 2.3 Filtering 12
- 2.4 Spectrum 13
- 2.5 Current Algorithms 14
  - 2.5.1 TROIKA 14
  - 2.5.2 Finite State Machine 15
  - 2.5.3 JOSS 15
  - 2.5.4 PARHELIA 16

3 Implementation **17**
- 3.1 Gathering Data 17
  - 3.1.1 Wear OS Development 17
  - 3.1.2 Uploading Recordings 19
  - 3.1.3 Raspberry Pi Access Point 19
  - 3.1.4 Synchronising Signals 21
  - 3.1.5 Informed Synchronisation 22
- 3.2 Filtering 24
  - 3.2.1 Butterworth Filter 24
  - 3.2.2 Chebyshev Filter 24
- 3.3 Motion Artefact Reduction 25
  - 3.3.1 Adaptive Noise Cancellation 25
- 3.4 Heart Rate Calculation 27
  - 3.4.1 Local Maxima 28
  - 3.4.2 Peak-Peak Standard Deviation Minimization 28
- 3.5 Joint Sparse Spectrum Reconstruction 29
  - 3.5.1 Joint Sparse Spectrum Reconstruction 29
  - 3.5.2 Spectral Peak Tracking 30
- 3.6 Earbuds 31
  - 3.6.1 Gathering data 32
  - 3.6.2 Sensor Fusion 34
  - 3.6.3 Kalman Filter 34

4 Evaluation **37**
4.1 Gathering Data . . . 37
4.1.1 Synchronising Signals . . . 37
4.2 Filtering . . . 38
4.2.1 Butterworth Filter . . . 39
4.2.2 Chebyshev Type II Filter . . . 41
4.3 Motion Artefact Reduction . . . 42
4.3.1 LMS . . . 42
4.3.2 NLMS . . . 43
4.4 Heart Rate Calculation . . . 45
4.5 Joint Sparse Spectrum Reconstruction . . . 47
4.6 Earbuds . . . 48
4.6.1 Problem . . . 49
4.6.2 Sensor Fusion . . . 49

5 Conclusion **51**

Bibliography **51**

A Project Proposal **55**

List of Figures

2.1 Finger pulse oximeter. Image source [4]. 11
2.2 Example PPG signal. Image and annotations from [6]. 12
2.3 Diagram showing filter characteristics 13
2.4 An example spectrum. 14

3.1 Watch Interface. 18
3.2 The overall process of uploading files. 19
3.3 Absolute Cross Correlation over time. 22
3.4 Comparison of Butterworth filter orders 24
3.5 Comparison of Chebyshev filter rs values 25
3.6 ANC overall flow 26
3.7 Positioning of the PPG sensor on Jabra elite sport earbuds. 31
3.8 Retrieving Data from Earbuds. 33

4.1 Simulating a noisy heart-beat. 39
4.2 Butterworth Filter Power Spectrums. 40
4.3 The effect of Butterworth filter order on execution time. 40
4.4 Chebyshev Filter Power Spectrums. 41
4.5 The effect of Chebyshev filter order on execution time. 42
4.6 Comparison of LMS Error at Different Tap Numbers. 43
4.7 Comparison of LMS Error at Different Step Sizes. 44
4.8 Comparing Numbers of Taps for NLMS. 45
4.9 Comparing Step Size for NLMS. 45
4.10 Comparison of NLMS and LMS Validity. 46
4.11 Comparison of Peak Finding Algorithms. 46
4.12 Comparison of Heart Rate Peak Calculations. 47
4.13 Spectrum Reconstruction. 47
4.14 Comparing Spectrum Reconstruction to NLMS. 48
4.15 Plot of Earbud Heart-rate against ECG. 49
4.16 Kalman Filter Demonstration. 49
4.17 Sensor Fusion Error. 50

Acknowledgements

Chapter 1

Introduction

Elite runners have historically used heart rate to provide an accurate measure of fitness, and allow them to train more effectively. Previously, Electrocardiography (ECG) chest straps have been used to measure heart rate, by detecting the electrical signals controlling the expansion and contraction of the heart. They are accurate devices however often prohibitively expensive, and hence inaccessible to casual runners.

In recent years, a new technology has emerged - Photoplethysmogram (PPG) - light is directed at the skin, and sensors measure how much blood vessels scatter it. PPG sensors are cheaper than ECG sensors, and hence are available in a variety of products, particularly smartwatches. This innovation has bought a new wave of advanced training and monitoring onto the wrists of any runner.

Switching from ECG to PPG is not without flaws though - ECG sensors return a clean signal, as opposed to PPG signals which are contaminated with noise.

In particular when running, the motion can cause blood velocity to change, and the sensor can slip across the skin [12], [13]. These result in distortions to the PPG signal, known as a motion artifacts (MAs). Fortunately, smartwatches contain other sensors, such as accelerometers and gyroscopes which can be used to predict the presence of MAs, and hence compensate for them.

My task is to research and develop a heart rate detection algorithm for smartwatches worn during running.

Chapter 2

Preparation

This chapter details the steps I took to determine how to develop my implementation. It first describes photoplethysmography, the method with which heart-beats are detected on watches. Next, I give an overview of the algorithms I will use to extract the heart-rate from the PPG signal: filters, peak-detectors and motion artefact removers. Finally, I outline my development methodology.

2.1 Photoplethysmography (PPG)

To develop an algorithm to track heart-rate on wrist-watches it is first important to understand how watches track heart activity. PPG is a technique where light is used to detect the volume of blood in veins. In hospitals, this is used in finger pulse oximeters (Figure 2.1) to record the heart-beat of patients. These work by transmitting light on one side of the finger, and then measuring how much light is received on the other side of the finger. The amount of light which permeates through the finger is related to how much blood is currently in the veins.

Figure 2.1: Finger pulse oximeter. Image source [4].

With watches, we cannot transmit light on one side, and receive light on the other side, as the wrist is far too large. Hence, instead of monitoring the absorption of light by the skin, we monitor the reflection of light by the skin. A light is shone into the wrist, and sensors nearby monitor how much is reflected back. When the wrist is full of blood, more light is reflected back as blood scatters lights.

We know now how the signals are recorded, but we have not yet explored the signal that is actually received from this technique. In Figure 2.2 we see a clean PPG recording. There are two peaks in the data - a sytolic peak and a diastolic peak. The systolic peak represents the point at which the heart has beat - and hence pushed blood through the body. The challenge is to find this peak.

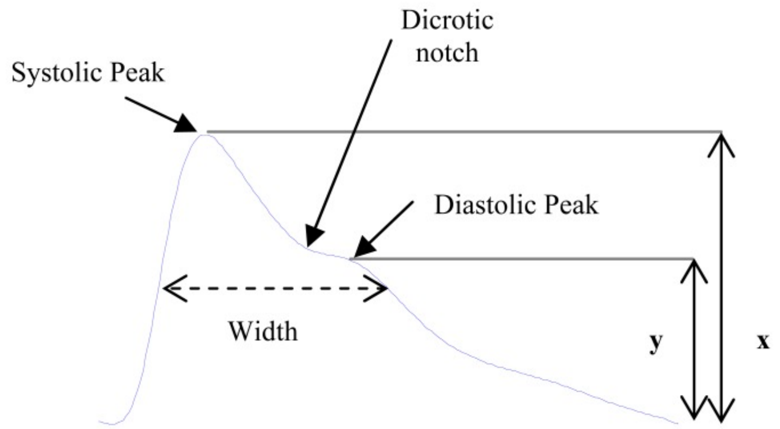


Figure 2.2: Example PPG signal. Image and annotations from [6].

2.2 Techniques

In this section, I detail the techniques which I may find useful in my implementation.

2.3 Filtering

PPGs produce a large amount of noise from various sources:

- Light pollution. Both DC and AC lights in the environment can infiltrate the sensor, affecting the PPG signal [8].
- Temperature. As temperature increases, so does the volume of blood, which will be picked up by the PPG sensor [10].
- Breathing. The change in pressure accosiated with respiration causes variations in the flow of blood, and hence can be seen by the PPG sensor [1].

In order to remove much of this noise, we can use filters to remove frequencies we know are irrelevant. We know the heartbeat can vary from 30 to 220 beats per minute, and hence we would like to disregard any noise outside of this range. In this section, I will introduce the concept of filters.

Figure 2.3 displays an example filter magnitude response diagram. This plots the amount of gain applied to each frequency. A gain of zero means the signal is removed, a gain of one means the signal is unchanged. The characteristics displayed are as follows.

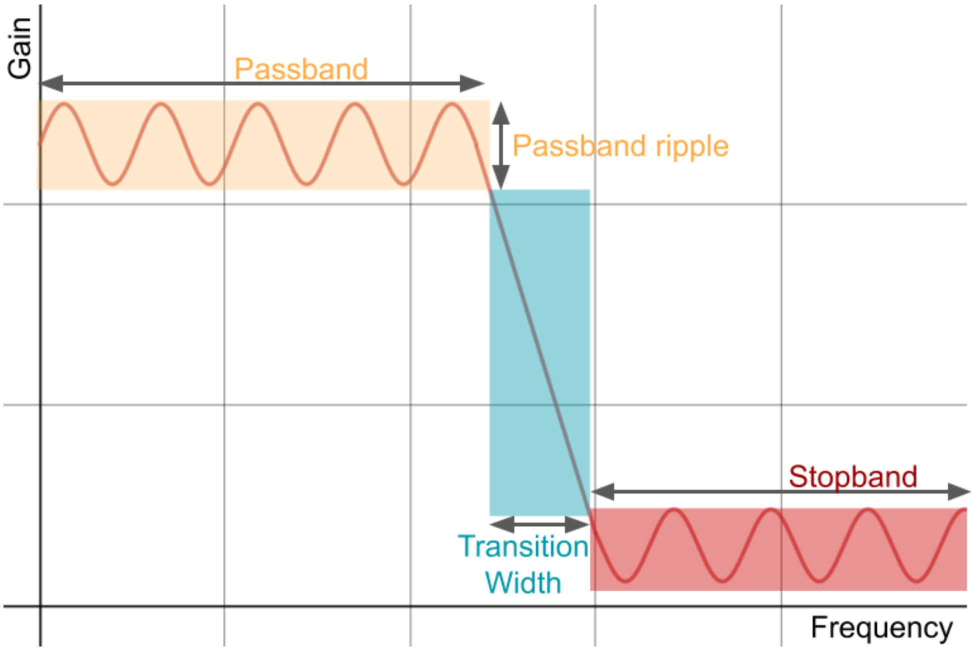


Figure 2.3: Diagram showing filter characteristics

The passband is the range of frequencies we would like to remain. In an ideal filter, there is no loss within the passband.

Passband ripple describes the variation in amplitude *within* the passband. An ideal filter will have no passband filter, such that all frequencies within the passband are permitted equally.

The transition width is the frequency range between the start and stop band. Ideally, this is zero, such that frequencies outside of the passband are instantly reduced.

The stopband is the range of frequencies we wish to remove. An ideal filter completely removes all frequencies within the stopband.

There are four different filter types, which describe the frequencies we remove, as follows:

- Lowpass - allow frequencies below the critical frequency.
- Highpass - allow frequencies above the critical frequency.
- Bandpass - combination of lowpass and highpass - given two frequencies, we want the passband to be between those frequencies.
- Bandstop - given two frequencies, put the stopband between them, leaving the passband outside those frequencies.

2.4 Spectrum

A spectrum (or power spectrum) describes the power of each frequency present within a signal. This is a useful tool for analysing signals, giving us the ability to detect where the frequency corresponding to heart-rate is, and additionally detect what frequencies are noise in a signal.

To demonstrate a spectrum, I produced a signal of two sine waves, with frequency 10 hz and 30 hz, and some random noise. The spectrum this produced is shown in Figure 2.4, it shows the presence of the two sine waves, and the random noise is shown by the random baseline variation.

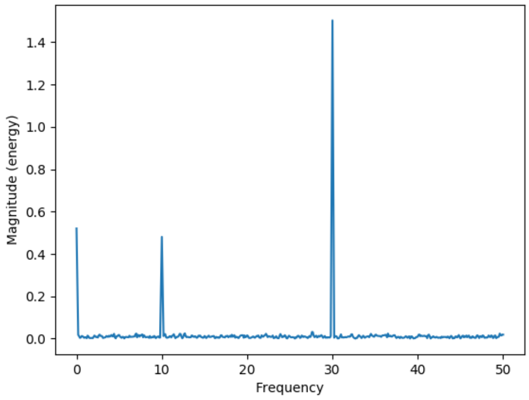


Figure 2.4: An example spectrum.

2.5 Current Algorithms

In this section, I look at the literature already produced on this topic, discussing the algorithms which have been proposed in various papers.

2.5.1 TROIKA

TROIKA [14] is a general framework for removing motion artifacts from PPG signals taken when exercising. It consists of signal decomposiTion , sparse signal RecOnstruction and spectral peaK trAcking, each of which I will explain here.

Signal decomposition is where a signal s is split into Q components such that $s = \sum_1^Q s_i$. TROIKA recommends using singular spectrum analysis (SSA) which decomposes the signal into oscillatory components and noise. It determines what components are likely noise based on analysis of the frequency spectrum of accelerometer data. The dominant frequencies in the accelerometer signals are determined to be noise in the PPG signal.

Sparse signal reconstruction (SSR) is used to generate the frequency spectrum of the now filtered PPG signal. As opposed to a periodogram, the spectrum from SSR gives higher resolution frequency information, allowing more precise identification of peaks.

Spectral peak tracking uses the current SSR spectrum, and the last known heart-rate. In the paper, they use an 8s window, with 2 s shifts, which means the algorithm is run on one 8 s segment, before being shifted 2 s forwards then run again. The windows of the signal overlap by 6 s, meaning that the heart-rate between two calculations will be very similar. Hence, TROIKA recommends selecting a peak in a small range around the last known heart-rate.

2.5.2 Finite State Machine

The issue with the TROIKA framework is that it is prone to repeating errors when they happen. For example, if the algorithm accidentally locks on to a frequency related to motion it is very difficult to recover. A technique suggested to avoid these errors makes use of finite state machines (FSM) [3].

This approach uses a FSM with four states as follows:

- Stable - the heart-rate estimate is likely to be very accurate.
- Recovery - the heart-rate estimate is somewhat accurate.
- Alert - the heart-rate estimate is possibly wrong.
- Uncertain - the heart-rate estimate is probably wrong.

To transition between states we need some sort of measure which tells us how confident we are in the heart-rate estimation - in the FSM we use crest factor (CF). The CF is the ratio of the peak value to the root-mean-squared value. The lower the CF, the less likely the HR estimate is to be accurate. Hence, when the CF value falls below some threshold, we go into a state with less confidence in the heart-rate value. Additionally we are guided by heart-rate change - if the heart-rate remains constant that is a sign we are locked on the wrong signal.

Now, we have a FSM which will describe our confidence in a heart-rate measurement - the question is how we use it. In the paper, an estimation is only valid if it was taken in the stable state. Hence, we sacrifice some of the data, but as a result we can be much more confident in the results we do get.

2.5.3 JOSS

JOSS [15], or JOint Sparse Spectrum reconstruction consists of two stages, joint sparse spectrum reconstruction and spectral peak tracking. I will explain these here.

Joint sparse spectrum reconstruction aims to produce a spectrum of the raw PPG as though there were no MAs. It does this by simultaneously calculating the power spectrums of the PPG signal and all the accelerometer channels. It then subtracts the accelerometer channels spectrums from the PPG spectrum in such a way that retains only the frequencies associated with heart-rate.

Spectral peak tracking is much the same as described in TROIKA, initially finding the highest peak in the PPG spectrum. It then follows that peak using the assumption that in two successive windows the heart-rate will be very similar.

JOSS improves on TROIKA, providing a lower average absolute estimation error and additionally works better when the PPG is sampled at a lower rate.

2.5.4 PARHELIA

PARHELIA [7] is a PARticle filter-based algorithm for HEart rate estimation using photopLethymographIc signAls. It consists of two stages: spectral analysis, and then particle filtering, which I will delve into now.

The spectral analysis computes the spectrum of the PPG and each acceleration signal, by running FFT on each.

A particle filter can be used to represent uncertainty in a system, by having multiple particles representing estimates of the current state. Each particle is assigned a weight which represents our confidence in that particle being correct. In PARHELIA, there are five steps, as follows:

1. Initialisation - generate particles uniformly in the range 60 bpm and 170 bpm.
2. Prediction - every iteration particles are moved randomly based on a normal distribution with standard deviation 6 bpm.
3. Weight calculation - the weight of a particle is increased if it's bpm is a peak in the PPG signal, and reduced if it's a peak in the accelerometer data.
4. BPM Estimation - we estimate current BPM by looking at the weighted average of particles.
5. Resampling - particles with the lowest weights are redistributed to more likely states.

This approach improved on JOSS, claiming to provide 8.6% better estimation accuracy, and 20 times faster performance. However, it does require a higher sampling rate than JOSS.

Chapter 3

Implementation

3.1 Gathering Data

In this section I detail the process of developing an application to record PPG data on a Wear OS watch. This turned out to be more complicated than I presumed due to interesting issues inherent with developing for a small wearable with limited power.

3.1.1 Wear OS Development

Language Choices The watch application was developed using the official IDE for Android development - Android Studio. I programmed in a language I'd never used before - Kotlin. Kotlin is an open source language based on Java, which aims to reduce boilerplate code, adds null-safety, and remains interoperable with Java, allowing libraries written for Java to be used in Kotlin.

Structure The program was split into classes as described here:

- *MainActivity* - central class responsible for starting, stopping and saving the recordings.
- *SensorListener* - interface which overrides the Android *SensorEventListener*. Contains methods which are called by the WearOS system whenever new sensor data comes in. Additionally contain methods which save the received data. We need one child for each sensor as each different sensor provides different data, and must be saved in a different format. So we have the following children:
  - *PpgListener*
  - *AccelerometerListener*
  - *RotationListener*

When the user presses start recording, *MainActivity* registers each *SensorListner* with the OS so they receive any sensor changes.

When the user presses stop recording, *MainActivity* unregisters the listners and gets each one to save their recording.

Storing Recordings It was critical that I could easily import the recordings into any program, and hence I chose to export the data into a CSV text file. Each *Listener* class uses a *CSVWriter* object from library *opencsv* to write to a unique file for each sensor, within a directory chosen by *MainActivity*. For example, at the end of the recording in the directory *YYYY-MM-DD/HH.MM.SS/* we have files *ppg.csv*, *accelerometer.csv* and *rotation.csv*.

Interface I chose a simple yet functional interface to allow the user to start and stop a recording. The interface changes colour when recording, which makes it very easy to verify the recording is in progress. See the interface in figure 3.1.

Figure 3.1: Watch Interface.

Problems After I'd developed the application and was testing it out I found two key issues which I will explain here.

Power Saving Power is an enormous issue on wearable devices, as the form factor drastically constrains the size of the battery. Hence developers have explicitly designed Wear OS to limit power usage of any application as much as possible. As part of these optimisations Wear OS automatically suspends any application if it thinks they are not being used. While this is useful for most users, my application needs to run continuously without interruption.

Initially in testing there were long periods of time where no sensor values were being recorded, due to the application being suspended.

To fix this issue, a wake lock was included. Once a wake lock is acquired, the OS does not suspend the application. I added one which is acquired every time a recording is started, and released when the recording ends.

Sampling Rate When I subscribe to values from the PPG sensor and accelerometer sensors, the API asks for a parameter - sampling period, which is the amount of delay we want between sensor readings being delivered. In practise, Android only uses this number as a hint - values may be provided at a higher or lower rate.

Additionally, Android does not expose the actual rate at which sensor values are being provided to the OS. The sensors which are placed within devices always have a constant sampling rate. Sensors write to a very short buffer, which Android then reads from at a rate it decides. This means although we have a reliable sensor, Android occasionally misses readings.

I record the data with a sampling period of 0 - which signifies that I want updates as soon as they are available. However, Android is not guaranteed to provide this. I mitigate this by noting the time each recordings is provided to the application. Then, I define sampling rate to be equal to the time difference between the first and last sample, divided by the number of samples.

So, I have created an application which allows the user to record accelerometer and PPG values at a constant sampling rate, then save them to the watch.

3.1.2 Uploading Recordings

Now data has been recorded, the values must be uploaded somewhere to enable later analysis. I created a server which would run on a Raspberry Pi that the watch could connect to over the local network. The files would be passed to the server, and from there the files are uploaded to Google Drive where they can be accessed anywhere. In this section I go over the details of this implementation. See figure 3.2 for the overall uploading process.

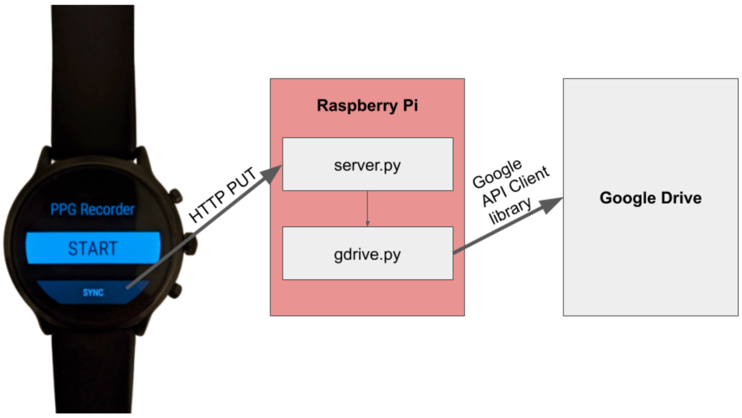


Figure 3.2: The overall process of uploading files.

Changes to Wear OS Application

First a 'sync' button was included in the interface, which users can press to trigger the sending of files to the server.

Then I added the logic to be triggered when the button is pressed. This logic iterates through every file in the recordings directory and performs a HTTP PUT request with them. I used the *OkHttpClient* library in order to send these PUT requests.

3.1.3 Raspberry Pi Access Point

To connect the watch to the server I needed to create a local access point on the Raspberry Pi. The packages I used to do this were *hostapd* and *dnsmasq*.

The package *hostapd* is used to setup the access point, so I used it to configure the name you can find the network from, the password of the network and the channel it can be accessed on.

The package *dnsmasq* is used to setup the DHCP service, assigning IP addresses to devices. I set it up to assign the range of adresses from 192.168.0.1 to 192.168.0.255 on the same interface as *hostapd* is broadcasting over.

Flask Web Application on Raspberry Pi

Now, to setup the server I used Flask, a Python Web App framework. I chose this over other frameworks like Django as it provides a flexible framework which is easy to understand and deploy.

I created an application *server.py* which accepts PUT requests containing a file from URL */upload/path* where *path* is the location we want to place the file. Then, the server makes the directories up to *path* if they don't exist, placing the file in this directory. Unlike in a normal server implementation, I do not secure this file upload process. I am aware there are a number of security issues concerned with leaving a file upload with arbitrary path options, for example a user might be able to upload a file in the parent direction by placing a PUT request for */upload/../../example*. However, I do not worry about these attacks as the access point is secured by *hostapd* and only the watch will be connected.

Once the file has been saved, I upload the new file to google drive, by calling *gdrive.py*, which I detail below.

Python Scripts to Upload to Google Drive

To manage the process of uploading files to Google Drive, I created a new application *gdrive.py*. To upload to Google Drive using Python, I used the library *googleapiclient*, which was a surprisingly complicated API, as I will explain here.

First, to connect to Google servers, the API must authenticate to ensure that it has permission to access a user's Drive. The service requires a credential token which expires periodically. Hence, I store the credentials in a binary file using library *pickle*, and when I need to access them, I refresh them if they have expired.

Then, I must actually upload a file. This process is complicated as Google Drive does not use a path system to store files. Instead, it is a set of files each with a unique IDs. A folder is simply a file with a *folder* file type, and a collection of files associated with it. Furthermore, as files are not referenced by filename, it is possible to have multiple files in the same folder with exactly the same filename.

The Google Drive API for Python only contains two methods which are useful for my application:

- *list(parent)* which lists the files within folder *parent*.
- *create(file, metadata)* which creates a *file* with *metadata*. Note that as folders are just files, this is also how folders are created, by specifying the folder type in *metadata*.

Using these methods, I define three methods within *gdrive.py*, as follows:

- *createFolder(name, parent)* creates folder by creating a file with metadata specifying the type as a folder.
- *getOrCreatePath(path)* is used to obtain a 'path' from Google Drive. It works by iterating through each level of the path, creating the necessary folders if they don't already exist, then entering them.
- *uploadFile(filepath)* created the path up to the file given by filepath, and then upload the file to the folder.

Now, I have created a system which allows the user to upload recordings from the watch to Google Drive via my server. And thus, I have completed the data recording process. The data is recorded on the Wear OS app, and then with a click transmitted to a Google Drive so it can be analysed.

3.1.4 Synchronising Signals

Once I have collected the PPG signal from the watch, and have recorded the ECG from the chest, I need a method to synchronise the signals, such that they start at the same time and we can compare them accurately.

I set out to produce a solution with the following properties:

- Able to synchronize signals with ±0.3s accuracy. With a heartbeat of 200 bpm this represents being within a heartbeat. This is an acceptable level of delay, as heart rate is averaged over several beats anyway.
- Can synchronize signals given the two recordings are started within two minutes of each other. This constraint is helpful as it prevents wasted time searching through the signal.
- Does not use the clocks built into the device. We must assume the clock within the ECG is unreliable. Additionally, the two devices may be synchronized to different time, and hence could be out by any amount of time.

Given I know both devices have an accelerometer, I realised I could ask the wearer to move in some motion which is picked up by both devices. Then, I could compare the two accelerometer signals in order to discover the movement. Then comparing the starting times of the two signals would enable calculation of the time difference between them.

The type of motion I chose was important, as it had to be easy to explain to the wearer, but also provide sufficient motion for it to be identifiable against normal motion. I decided jumping was appropriate - I would ask the wearer to hold the watch close to their chest and jump three to five times.

The cross correlation of two signals f and g is a measure of similarity as a function of the displacement between them. It is simple to calculate, as a sum of the products between the samples of f and of g displaced by n.

To synchronise the accelerometer, I developed the following three step algorithm:

1. Normalize signals, by moving to zero mean,
2. Crop PPG signals to two minutes, crop ECG signals to four minutes, compute cross correlation with f = PPG acceleration, g = ECG acceleration.
3. Compute the other way - crop ECG to two minutes, PPG to four minutes, compute cross correlation with f = ECG acceleration, g = PPG acceleration. I compute both ways, to ensure that we can synchronise regardless of which device started recording first.
4. Find the maximum cross correlation across each of the calculations.

3.1.5 Informed Synchronisation

Running cross correlation on two signals is an effective way to compare their similarity, however it is not efficient. Cross correlation has complexity $O(n^3)$. In this section I develop a new algorithm which uses knowledge of the problem to speed up the cross correlation calculation.

By graphing the cross correlation of the acceleration signals, we can draw some useful features that help us to work out a more optimal solution. See figure 3.3 for an example of absolute cross correlation plotted between the two accelerometer signals over time. It is clear that there is one peak which is above all the others, where the jumps line up perfectly. However, in addition we can see that there is a significant rise in cross correlation around the peak, where we are seeing peaks due to one or two of the jumps aligning. This is useful as we hence do not need to scan at each possible time difference, instead we can to a lower resolution initial scan. After this initial scan is done, we find the peak, and check each possible time difference around the lower resolution peak to find the actual peak.

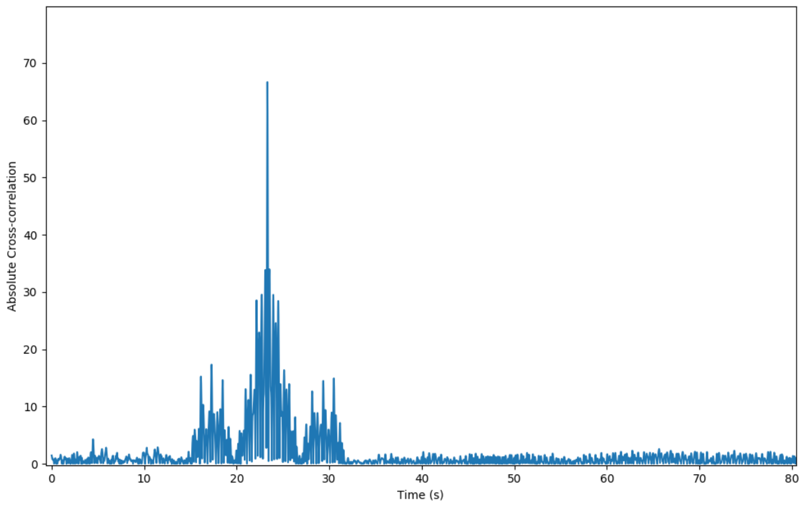


Figure 3.3: Absolute Cross Correlation over time.

The algorithm I designed takes two parameters: step size (g) and correct range (r). Step size is the number of samples we skip in the initial scan, so the lower it is the more likely we are to find the peak but the longer the algorithm takes. Correct range gives the range of the time period we want to scan once we have found an initial peak, in order to find the actual peak. The code is included here:

```
def fastCorrelate(f, g, freq, initTimeGap = 0.25, searchBound = 3):
    """
    Calculate cross correlation c of two numpy arrays, as defined by c[k] = sum_n (f[
    n+k] * g[n])
    using optimized method which does an initial correlation at a lower resolution
    and then hones in on the solution.

    Parameters
    ----------
    f, g : numpy arrays
        Input signals.

    freq : float
        The frequency of the signals.

    initTimeGap : float
        Interval at which to perform correlation intially in seconds.

    searchBound : float
        Bound wihtin which we search for the solution once we have made an initial
    pass.


    Returns
    -------
    k : int
        k is the time at which we think the signals are synced, given in terms of the
     number of samples through f we are.
        i.e. the value of k which maximizes c[k]

    v : float
        the value of cross correlation at that point (c[k])
    """
    if f.size < g.size:
        raise ValueError("Array f must be larger than g")

    length = f.size - g.size

    gap = int(initTimeGap * freq)

    c = np.zeros((length))
    for k in range(0, length, gap):
        c[k] = np.sum(f[k : k + g.size] * g)

    maxPos = np.argmax(c)
    bound = searchBound * freq
    lowerBound = max(maxPos - bound, 0)
    upperBound = min(maxPos + bound, length)
    for k in range(lowerBound, upperBound, 1):
        c[k] = np.sum(f[k : k + g.size] * g)


    k = np.argmax(c)
    v = c[k]

    return (k, v)
```

In Section 4.1.1 I found that a value of $g = 0.25$s provides significantly better performance, but still finds the correct time difference for all recorded data.

3.2 Filtering

In this section, I discuss the implementation of filters, as introduced in section 2.3. The ideal filter, as described above, is impossible - and hence we have a variety of filters which compromise between the desirable characteristics. Following this, I detail two of these compromises - the Butterworth filter, and the Chebyshev filter.

3.2.1 Butterworth Filter

The Butterworth filter aims to minimize passband ripple, at the expense of a larger transition width. To define a Butterworth filter, multiple parameters are used, which I will describe further here.

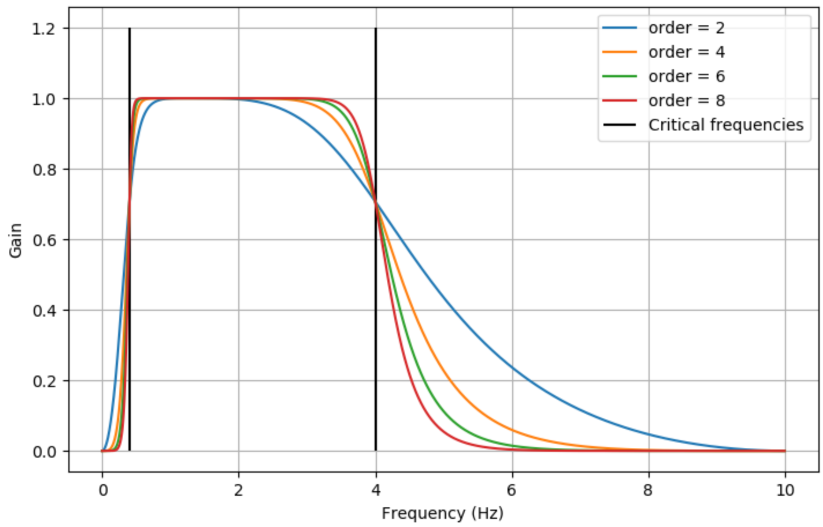


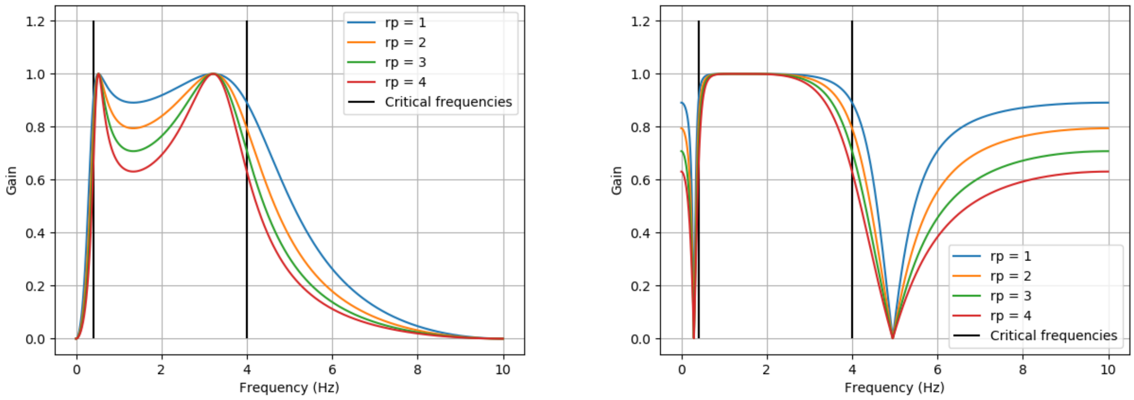
Figure 3.4: Comparison of Butterworth filter orders

Order (N) controls the transition width - the higher the order, the lower the transition width. This is desirable - we want to keep the passband as large as possible, and transition quickly to the stopband. However, it comes at the cost of computation complexity, as higher order filters take longer to apply. See figure 3.4 for a plot displaying the effect of different orders on a filter's response.

Critical frequency gives the frequency at which we want gain to drop below $1/\sqrt{2}$ from the passband. Hence we use this parameter to define the point from which we wish to cut out noise. Figure 3.4 uses critical frequencies 0.4 and 4 hz, which correspond to 24 and 240 bpm, a reasonable heart rate range.

3.2.2 Chebyshev Filter

The Chebyshev filter aims to reduce the transition width as much as possible, but to do this it introduces increased ripple.



(a) Chebyshev Type 1

(b) Chebyshev Type 2

Figure 3.5: Comparison of Chebyshev filter rs values

The filter is defined with similar parameters to a Butterworth filter - order, critical frequencies, and type all have very similar meanings. However, there are type 1 and 2 Chebyshev filters, with different aims. Additionally, there is a new parameter rs. I will explain these differences here.

Type 1 Chebyshev filters introduce passband ripple, whereas type 2 Chebyshev filters introduce stopband ripple. Figure 3.5 shows how these differences manifest in the response of a second order Chebyshev filter.

The parameter rs is defined as the maximum ripple permitted below unity gain. Higher rs results in more ripple for a type 1 filter, but less ripple for a part 2 filter, as is demonstrated in figure 3.5. Introducing more ripple is undesirable as leads to more signal distortion, however can be advantageous as it reduces the transition width.

3.3 Motion Artefact Reduction

In this section I explore a method which can be used to reduce the effect of Motion Artefacts (MAs) on the PPG Signal.

3.3.1 Adaptive Noise Cancellation

Adaptive Noise Cancellation (ANC), as described by Widrow et al. [11], is a technique which allows us to remove noise from a signal, given we have another signal correlated to the noise in some way. In our situation, we know MAs are correlated to motion, so we can use the accelerometer to remove MAs. This technique is extremely useful, as it can still produce good results when the MAs are at the same frequency as the heart beat, even if the MAs have a larger amplitude.

An overview of the algorithm follows. We have signal from heartbeat s that we want to figure out, but this is contaminated by noise from MAs n_0. We assume the PPG sensor reading p is such that $p = s + n_0$. Additionally, we have accelerometer sensor readings

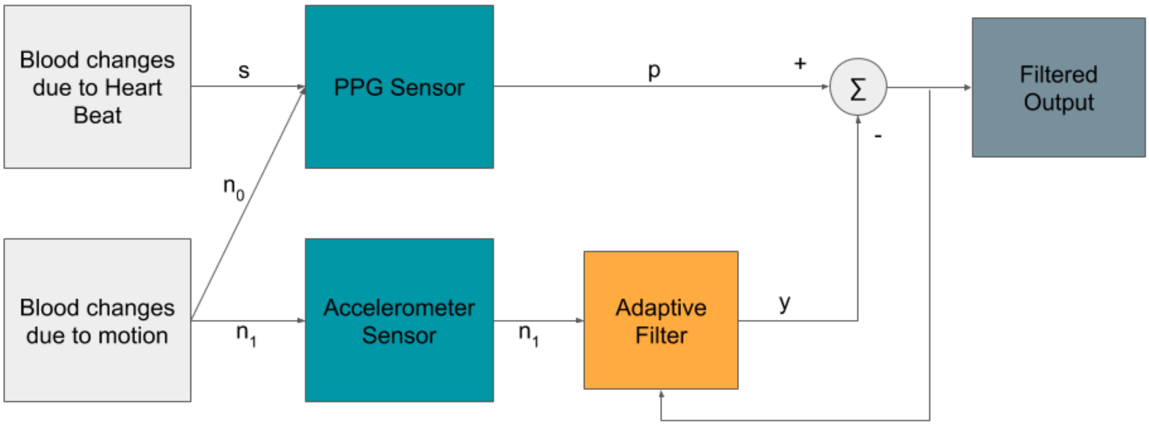


Figure 3.6: ANC overall flow

n_1. We filter n_1, producing signal y and subtract this from p, to produce $z = p - y$. Our aim is to adjust the filter, such that y is as close to n_0 as possible.

The type of filter we adjust is known as a Finite Impulse Response (FIR) filter. An Nth order filter is defined by N weights. These weights are convolved with the signal, hence filtering it.

The adaptive filter aims to choose a filter such that the power of the output $E[z^2]$ is minimized. Given s is uncorrelated with n_0 and n_1, and n_0 is correlated with n_1, it can be proven that minimizing the output power is equivalent to finding $z = s$.

There are multiple algorithms which exist in order to adaptively filter a signal. I'll look at two, first describing the least mean squares (LMS) algorithm from Widrow and Hoff. LMS first initializes all of the filter weights to 0. At each step, it finds the gradient of the mean square output $E(z^2)$. Each weight's gradient describes what would happen to the signal if we kept using that weight value. So if the gradient of a weight is positive we know that using that weight again would increase the mean squared output. Hence, at the end of each step we shift the weights down if the gradient is positive, and visa versa if the gradient is negative.

LMS takes a parameter - step size μ, which scales how much we move the weights by. The equation used to update weights is

$$w_{n+1} = w_n - \mu \nabla E(y^2)[n]$$

where w_n is the nth filter weight, and $\nabla E(y^2)[n]$ is the nth weight gradient.

The LMS code I produced is as follows:

```
def lms(ppg, accel):
    """
    Run a least mean squares adaptive filter to remove noise from the signal
    that we know is correlation to the reference signal.
```

```
Inputs
------------
 - ppg - the signal we want to remove noise from
 - accel - a signal we think is correlated with the noise

Outputs
------------
 - filtered - the signal with noise removed
"""

N = ppg.size          # Size of PPG signal
K = 15                # Number of filter taps
step = 0.001          # Step size for LMS

# Preprocessing
freq = ppg.getFrequency()
accel = accel.resample(freq)
accel = accel.crop(N)

w = np.zeros(K)      # Initial filter
e = np.zeros(N-K)    # Initial error

for n in range(0, N-K):
  acceln = accel[n+K:n:-1]
  en = ppg[n+K] - np.dot(acceln, w)
  w = w + step * en * acceln
  e[n] = en


motion_noise = signal.convolve(accel, w)
estimated_heart = ppg - motionNoise

return estimated_heart
```

Step size must be chosen very carefully, as if it is too large there is a risk of LMS missing the solution as it overcompensates for the gradient. If it is too small, then the algorithm becomes too expensive as we need many more iterations. The situation is made worse as the ideal step size changes with respect to the power of the input, making it very difficult to choose a step size which gives stability.

To solve these stability issues, NLMS was introduced. It is the same as LMS, except you normalize the input power each iteration. This makes choosing a step size which gives stability a lot easier.

So now we have a stable adaptive filter, which will allow us to tune a filter that will remove motion noise from our PPG signal.

3.4 Heart Rate Calculation

Now we have removed noise from the signal, we should have a signal representing the volume of blood in veins within the arm. As the heart beats, it sends a wave of blood through the body, forcing more blood through these veins. Hence, peaks in the PPG signal correspond to heart beats.

There are multiple approaches to finding heart-rate from our filtered signals, and I will detail a few here.

3.4.1 Local Maxima

If we can find the peaks in the signal, we can count them up and divide by the time period we count those peaks within, giving us a heart rate. One way we can define peaks is as local maxima.

The local maxima in a signal are defined as samples which are larger than both neighbouring samples. Finding local maxima is an easy problem to solve - intuitively, we can create an algorithm which iterates through the samples, comparing each sample to it's neighbours, which has complexity $O(n)$.

3.4.2 Peak-Peak Standard Deviation Minimization

Finding the local maxima is a naive approach that doesn't adapt to any problem specific knowledge. For example, we know heart-rate will not climb above 220 bpm, however the prior solution could report 300 bpm. Additionally, it might find multiple local peaks around a single heart-beat.

Ideally, we want to include knowledge about what a normal heart-beat looks like in our detection algorithm. One thing we can do is look at peak-peak intervals - the time that passes between heart beats. Surprisingly, this is normally not consistent - a healthy heart actually beats with some variation. However, when running it has been found that the heart beats with very little variation [9], and hence in our case we can assume that the heart beats regularly. The aim of this method is to use this assumption, along with minimum and maximum heart-rate to better detect heart-beats.

The method begins by calculating an initial set of peaks, by calculating the rolling mean of the signal, and marking points above that mean as peaks. Then, we compute the standard deviation of the time between these peaks (peak-peak interval). Then, we iteratively increase the rolling mean and again calculate the peaks, along with the standard deviation of their intervals. The idea is every time we increase the rolling mean we exclude more peaks. Then at the end, the peaks we actually select are the peaks which give a reasonable heart-rate (in the range 20-240 bpm), and have the smallest standard deviation.

The code I produced to do this is below:

```
percs = [0, 5, 10, 15, 20, 25, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 150, 200,
    300]
mov_ave = moving_average(signal, window_size)
mov_ave = mov_ave.getValues()
signal_vals = signal.getValues()

min_sd = np.inf
current_peaks = []

for perc in percs:
    current_mov_ave = mov_ave + mov_ave * perc / 100

    #Find points above current moving average
    x_peaks = (signal_vals > current_mov_ave).nonzero()[0]
    y_peaks = signal_vals[x_peaks]
    peak_edges = (np.diff(x_peaks) > 1).nonzero()[0] + 1

    # find the maximum between the peak edges.
    peaks = []
```

```
    for i in range(len(peak_edges) - 1):
        ys = y_peaks[peak_edges[i]:peak_edges[i+1]].tolist()
        if len(ys) > 0:
      peaks.append(x_peaks[peak_edges[i] + ys.index(max(ys))])

    # Calculate standard deviation of the peak peak intervals
    intervals = np.diff(peaks) / signal.getFrequency()
    sd = np.std(intervals)

    if sd < min_sd:
        min_sd = sd
        current_peaks = peaks

return np.array(current_peaks)
```

3.5 Joint Sparse Spectrum Reconstruction

So far all techniques I used to calculate heart-rate have worked in the time domain, so now I'll look at using the frequency domain. I developed a technique based on JOSS [15], which I discussed in the Section 2.5.3. The algorithm is composed of two main stages, joint sparse spectrum reconstruction and spectral peak tracking.

3.5.1 Joint Sparse Spectrum Reconstruction

Joint sparse spectrum reconstruction is a method to estimate the spectrums of multichannel signals. The problem can be summarised as solving the equation

$$Y = \Phi X + V$$

Where $Y \in \mathbb{R}^{M\times 4}$ is the matrix which has columns consisting of measurement vectors - the first column contains M PPG samples, and the second third and forth contain the x, y and z acceleration samples. Where N is the wanted resolution of the Fourier transform, $\Phi \in \mathbb{C}^{N\times M}$ is the redundant discrete Fourier transform basis matrix, as follows:

$$\Phi_{m,n} = e^{i\frac{2\pi}{N}mn},\ m = 0, \dots, M-1;\ n = 0, \dots, N-1$$

$V \in \mathbb{R}^{M\times 4}$ models noise, and $X \in \mathbb{C}^{N\times 4}$ is the desired solution matrix. There are multiple solutions for X, finding one is equivalent to finding the spectrum of each of the input signals.

To calculate X, I make use of the MFOCUSS algorithm [5]. MFOCUSS is designed to solve undetermined systems of measurement vectors, such as what we have. It is optimised to provide solutions where X is sparse, . I used the algorithm provided by Zhang, written in MATLAB, so I made use of the MATLAB Engine API for Python in order to call the algorithm from my existing code. The following code shows my SSR function:

```
def ssr(y, freq, N, eng):
    """
    Run sparse spectrum reconstruction using y sampled
    at frequency freq, and a Fourier matrix to produce N
    frequency taps. eng is the MATLAB engine used
    to run the MFOCUSS algorithm.
    """
```

```
    M = np.max(y.shape)

    # Make the Fourier matrix
    phi = np.zeros((M, N), dtype=complex)
    complex_factor = 1j * 2 * np.pi / N
    for m in range(0, M):
        for n in range(0, N):
            phi[m,n] = np.exp(complex_factor * m * n)

    Phi = matlab.double(phi.tolist(), is_complex=True)
    Y = matlab.double(y.tolist())

    X = eng.MFOCUSS(Phi, Y, 1e-10, 'MAX_ITERS', 4)
    x = np.array(X)
    x = np.abs(x) ** 2


    return x
```

Now we have calculated the spectrums of the PPG signal and the noise, we can remove noise from the signal, which is much easier in the frequency domain - we simply subtract the spectrum of the motion noise from the spectrum of the PPG noise, in order to reconstruct a clean spectrum. The advantage of using the MFOCUSS algorithm and calculating the spectrums together is that the frequency bins are much more likely to overlap between the accelerometer and the PPG, where there is motion noise.

The paper which describes JOSS recommends, for each frequency in the spectrum, subtracting the maximum of all the accelerometer frequency bins from the PPG's frequency bin. Additionally, it reccommends settting all frequency bins below the maximum frequency bin divided by four to zero, in order to sparsify the array. This is the relevant code:

```
aggression = 0.99
accel_max = np.zeros((N))
signal_ssr = spectra[:,0]

# Modify the SSR signal by subtracting the maximum acceleration in each bin
for i in range(0, bpm.size):
    # Max of acceleration at this frequency
    accel_max[i] = np.max([spectra[i,1], spectra[i,2], spectra[i,3]])
    signal_ssr[i] = signal_ssr[i] - aggression * accel_max[i]

# Set all SSR bins lower than the maximum divided by 5 to 0
max_bin = np.max(signal_ssr)
signal_ssr[signal_ssr < max_bin / 4] = 0
```

So, now we have produced a spectrum of the PPG signal, with motion noise removed, by computing the spectrums simultaneously.

3.5.2 Spectral Peak Tracking

We now have a spectrum, so we need some way of calculating heart-rate from it. We could just select the largest peak in the spectrum, however this would be naive - some noise may not be removed, so we could get random peaks. Additionally, we know that heart-rate is relatively consistent so we can use the previous heart-rate calculation to estimate the next one. Spectral peak tracking helps to achieve this.

My spectral peak tracking initializes the heart-rate by looking for the largest peak, as there will be little noise when the recording starts. Then, it tries to find peaks, first peaks that are less than 15bpm away, then peaks less than 25 bpm away. If no peaks are found, then the previously reported heart-rate is used again as the heart-rate is likely lost in noise. If there are multiple peaks, the algorithm takes the largest.

To avoid the heart-rate getting stuck at one value, a validation step is included. If the bpm reported is the same for more than 3 succesive runs, then validation is triggered. To validate, we return the closest peak to the latest recorded run, the code that finds this is below:

```python
def discover_peak(spectrum, prev_loc):
    """
    Find the closest peak in spectrum to the peak at prev_loc.
    """
    rng = np.arange(40, 220)
    N = spectrum.shape[0]

    # find peaks in range
    mask = np.zeros((N,))
    mask[rng] = 1
    filtered = (spectrum * mask).flatten()
    locs, _ = scipy.signal.find_peaks(filtered)

    if locs.size == 0:
        return prev_loc

    # find closest peak to prev_loc
    dist = np.abs(prev_loc - locs)
    index = np.argmin(dist)
    loc = locs[index]

    return loc
```

3.6 Earbuds

As an extension, I wanted to collect data from a new kind of PPG-enabled device: earbuds. In theory, these devices are less effected by motion noise as they are not affected by arm movement. Additionally, the ear has a rich set of veins from which we can gather PPG. For the purposes of this project I used a set of Jabra elite sport earbuds, which have a PPG sensor under the right bud - see figure 3.7. I connected the earbuds to the watch, and the following details the process of collecting data from the earbuds.

Figure 3.7: Positioning of the PPG sensor on Jabra elite sport earbuds.

3.6.1 Gathering data

To collect data, I connected the earbuds to the phone, and then data is sent through the Bluetooth Light Energy (LE) protocol. There are a few key pieces of terminology, which I explain here:

Generic Attribute Profile (GATT) describes the overall methods with which data values are transported by Bluetooth LE.

GATT Server is a device which stores and distributes information. In our case the earbuds are a GATT server.

GATT Client is a device which requests information from a GATT server, in our case a smart watch.

Characteristics are single data points, for example in our case a heart-rate integer. Characteristics also have a number of descriptors describing the value.

Services are logical group of characteristics, for example the heart-rate service contains the heart-rate measurement characteristic along with sensor position characteristic.

Characteristic Value Notifications are sent from client to server to indicate that the client would like to receive updates when a certain characteristic is changed. This is how frequently changing values are passed over Bluetooth LE, as it is more power efficient than having the client periodically request updates.

Figure 3.8 shows the flow I aimed to introduce in my data collection program to run on the watch. In Kotlin, I iterate through the connected Bluetooth devices, until I find the earbuds, and then the app connects to the earbud's GATT server. Once connected to the GATT server, the app requests notifications for the heart-rate characteristic. These characteristics are then periodically sent to the watch, where they are recorded.

Android Services

In order to avoid the data being lost when the application leaves the screen, I implemented all of the Bluetooth functionality within a service. A service allows code to run when the application isn't in focus. It does add some complexity into development though, as I describe here.

When a service is created, it is bound to the application which starts it. When a service is bound, the application can send requests, and receive updates from it - as I'll discuss further on. However, when an application is out of focus, services bound to it are destroyed, and hence our service must be unbound before this happens. Then, when our application comes back into focus, we rebind it, see the code below:

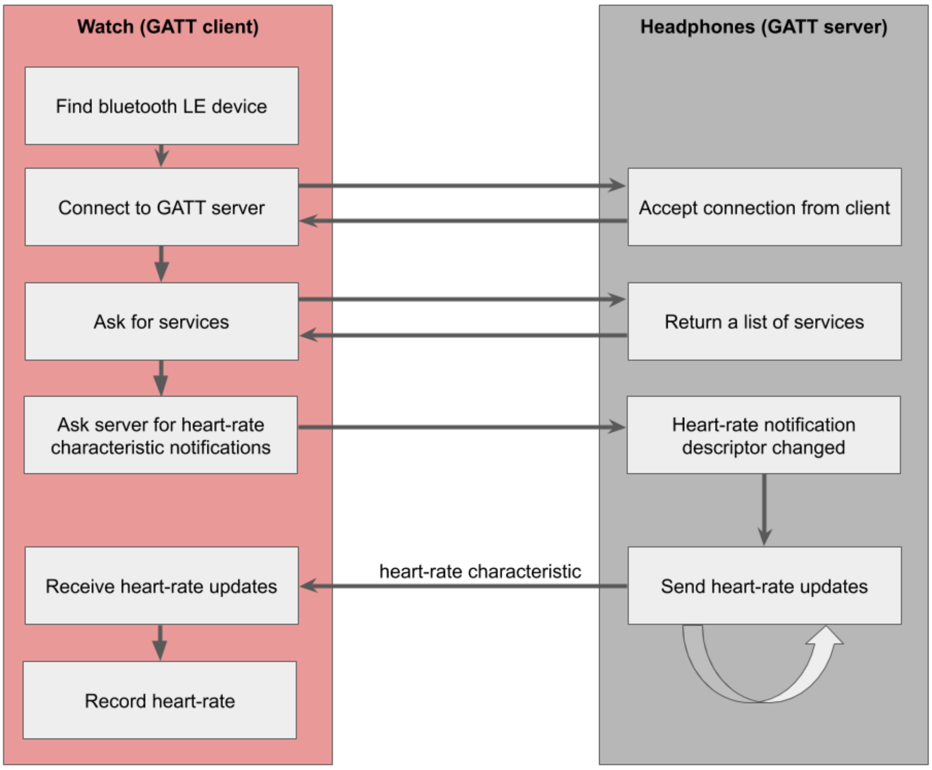


Figure 3.8: Retrieving Data from Earbuds.

```
override fun onStart() {
    doBindService()
    super.onStart()
}

override fun onStop() {
    doUnbindService()
    super.onStop()
}
```

In order to receive updates from the service, we must use intents, which are the way Android allows different applications to communicate. We create a *BroadcastReceiver* object in *MainActivity*, which deals with intents we might receive from the service - the intents it may send include device connected/disconnected and an update to say the recordings have successfully been saved.

```
// Handles various events fired by the Service.
// ACTION_GATT_CONNECTED: connected to a GATT server.
// ACTION_GATT_DISCONNECTED: disconnected from a GATT server.
// ACTION_SAVED: heart-rate data has been saved
private val mGattUpdateReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val action = intent.action
        when {
            HeartDeviceService.ACTION_GATT_CONNECTED.equals(action) -> {
                mConnected = true
                showConnected(mConnected)
            }
            HeartDeviceService.ACTION_GATT_DISCONNECTED.equals(action) -> {
                mConnected = false
                showConnected(mConnected)
```

```
            }
            HeartDeviceService.ACTION_SAVED.equals(action) -> {
                mRecordingStarted = false
                mSaveListener.onSave()
            }
        }
    }
}
```

3.6.2 Sensor Fusion

Once I had gathered data from the earbuds, I evaluated it and found a problem, see section 4.6.1 for details. In essence, the issue I found was that the earbuds would lose the heartbeat signal due to the sensor changing position during a run. Particularly annoyingly, the earbuds report a heart-rate of zero whenever this happens.

To rectify this, I will implement a sensor-fusion technique. This technique makes use of a Kalman filter. The idea is it takes measurements from the earbuds, which we know are accurate, and then in periods where the heart-beat signal drops out we guide the output using the less accurate PPG from the wrist-watch.

3.6.3 Kalman Filter

A Kalman filter can be used to estimate the value of an unknown variable, by producing a probability distribution for that variable that is updated over time. It consists of two phases:

1. Prediction step - predict the current heart rate based on the previous heart-rate, and the readings from the watch PPG.

2. Correct step - when we get a confirmed measurement from the earbuds, we move the heart-rate to the earbud's estimation.

The Kalman filter relies on modelling current heart-rate as a Gaussian distribution with it's standard deviation representing the certainty of the current estimate.

Kalman gain is the parameter used to weight the measurements against the prediction. A higher gain assigns more trust to the measurements, meaning the filter will respond quicker, a low gain produces more smoothing.

The code for my Kalman filter is as follows:

```
def predict(self, t, old_hr, old_var):
    if t < 1:
        raise ValueError("Predict step must start at time t > 0")

    # calculate hr difference between t-1 and t on the watch
    diff = self.watch[t] - self.watch[t-1]

    hr = old_hr + diff
    var = old_var + self.predict_var

    return hr, var
```

```
def correct(self, t, predicted_hr, predicted_var):
    # Calculate Kalman gain
    k = predicted_var  / (predicted_var + self.ear_noise)

    # Calculate new hr and variance
    hr = predicted_hr + k * (self.ear[t] - predicted_hr)
    var = (1 - k) * predicted_var

    return hr, var


def filter(self):
    hrs = np.zeros(self.size)

    # Initial step, wait until we get some data from the ear sensor before starting
    the filter
    t_min = 1
    while self.ear[t_min] == 0:
        t_min += 1

    hr = self.ear[t_min]
    var = self.ear_noise

    # Run the Kalman filter
    for t in range(t_min, self.size):
        hr, var = self.predict(t, hr, var)

        if self.ear[t] != 0:
            hr, var = self.correct(t, hr, var)

        hrs[t] = hr

    return data.Signal(hrs, 1)
```

Chapter 4

Evaluation

In this chapter, I describe the steps I took in order to evaluate both functionality and performance of my project. I go through all the implementation stages - gathering data, filtering data, motion artefact reduction, heart-rate calculation, post-processing and dealing with earbuds - for each commenting on their validity and performance.

4.1 Gathering Data

4.1.1 Synchronising Signals

In section 3.1.4, I detail an algorithm which allows me to calculate the difference in start-times between different recordings from a watch and from an ECG. Initially I use cross-correlation of accelerometer signals to determine the time at which the signals are the most similar, however this is quite ineficient, so I also develop an improved algorithm, referred to as informed cross correlation from now on.

In this section, I will evaulate both the validity and the performance of the solutions. To prove validity of imformed cross correlation, I compare the time difference it predicts to the actual time difference calculated by cross correlation, and then take the average absolute error over all recorded runs, in seconds.

To evaulate performance of informed cross correlation I run the algorithm on a sample set of accelerometer data, 500 times, using the python module timeit. This module avoids some easy mistakes that can be made when measuring execution times in python, for example it uses *time.perf_counter()* to measure time, which only measures execution time, not time spent sleeping.

Informed cross correlation takes an additional parameter, g which describes how many seconds to skip whilst calculating the initial cross correlation. I will test the algorithm with different values of this parameter, so I can determine which gives the best peformance whilst stil being valid.

The results of my test are displayed in Table 4.1.

The first result I draw from this table is that my informed algorithm leads to a massive improvement in performance. Even the slowest informed runtime is 6 times faster than the naive implementation. Next, it can be seen that the runtime of the informed algorithm

Table 4.1: Comparing Cross-Correlation Techniques.

Algorithm	Average Absolute Error (s)	Time for 500 iterations (s)
Cross correlation	0	112.227
Informed g = 0.1s	0	17.935
Informed g = 0.25s	0	11.089
Informed g = 0.5s	0	8.648
Informed g = 1s	0.137	7.069

increases as the size of g decreases. This makes sense as the smaller the size of g, the fewer cross-correlation values we calculate.

In terms of validity, the table shows that the total absolute error of the informed algorithm is 0, for all values of g up to $g = 1$s. As g increases, the likelihood that the initial scan misses the optimal solution increases because the step size is too large.

In conclusion, I have discovered that the informed algorithm is capable of giving a valid solution, although if the value of g rises too high the solution may no longer be valid. In addition, I have discovered that it gives a much quicker solution, which gets quicker as g rises. For safety, I would choose $g = 0.25$s, which ensures the result will be valid, whilst still providing a ten-fold performance increase.

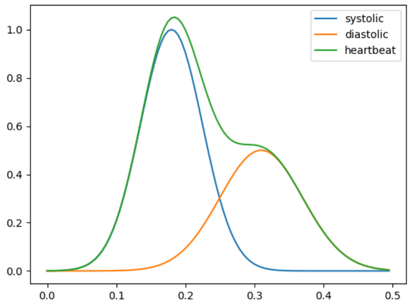
4.2 Filtering

To test filters, I will run two different tests. Firstly I will test how much noise the filter can remove alongside how much the information we want is actually retained. Additionally I will test how the speed of the filters varies, as in a heart-rate calculation algorithm we want to run the filters at speed.

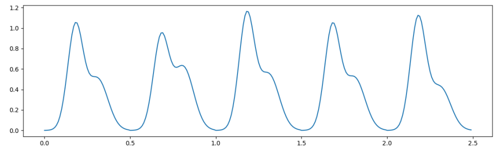
To test the quality of the filter I devised a test where I simulate a possible PPG signal without noise, then add noise. The noise I add will be random amount of high and low frequency noise, as we get in an actual PPG signal. Then to calcualte the quality of the filter, I filter the signal, and compare the power spectrums of the filtered signals to the clean signal.

To simulate PPG signals, I found that two Guassians could be used [2]. One Gaussian represents the higher systolic peak, and the other represents the lower diastolic peak. See figure 4.1a for an example of the clean signal my simulation produces. As heart-rate is relatively consistent during running, I can assume the peaks are in approximately the same position, with slight variation. I simulate a heart-rate of 120 bpm, by repeating the pattern heart-beat every 0.5 seconds. In each beat I randomly vary the amplitude of both peaks, as would happen with an actual heart-beat. See figure 4.1b for a simulated heart-beat of 5 beats.

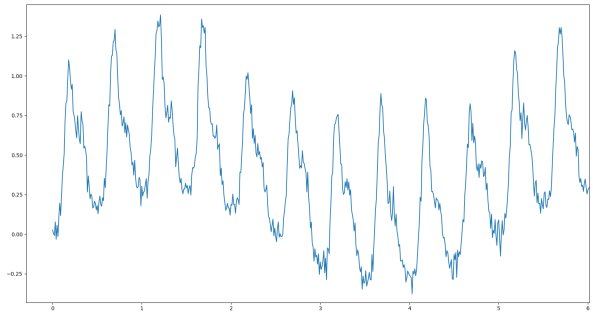
To represent the noise, I add a random Gaussian noise to each sample, to represent the measurement error, along with low frequency noise at 0.2 and 0.3 hz, representing the noise present for example due to temperature variation, and breathing. The results of adding this noise to a signal can be seen in figure 4.1c.



(a) A single simulated beat.



(b) Multiple simulated heart-beats.



(c) Simulated heart-beats with noise.

Figure 4.1: Simulating a noisy heart-beat.

Next, I must test the speed of the algorithms. To test this, I run the filter on a sample run from which I have collected 10 minutes of data. I run the test 500 times and take an average over the time taken.

I test the Chebyshev filter and the Butterworth filter with different parameters, in order to decide between the two of them, and additionally what parameters are optimal, in terms of both speed and validity.

4.2.1 Butterworth Filter

I test the validity of the filter by comparing the power spectrum produced by filtering the simulated heart-beat I detail above. The power spectrums are shown in Figure 4.2. For reference, at the top of the figure we plot the power spectrum of the clean signal and that signal with noise added. Ideally, we are looking to see the clean signal spectrum once we have filtered the noisy signal. The clean signal spectrum shows a clear peak at 2 hz, which corresponds exactly to the heart-rate we chose - 120 bpm. Additionally, there are peaks at 4 and 6 hz which are a feature of the power spectrum - it produces peaks at 2, 3, 4, and so on times the principle frequency. The noisy spectrum shows the additive Gaussian noise, which is present at every frequency, and additionally a small peak at around 0.2 hz, which shows the low frequency noise. I will compare how effective the filters are at removing both the random noise and the low frequency noise.

The results show that at all orders, the desired signal at 2 hz is present. However, as

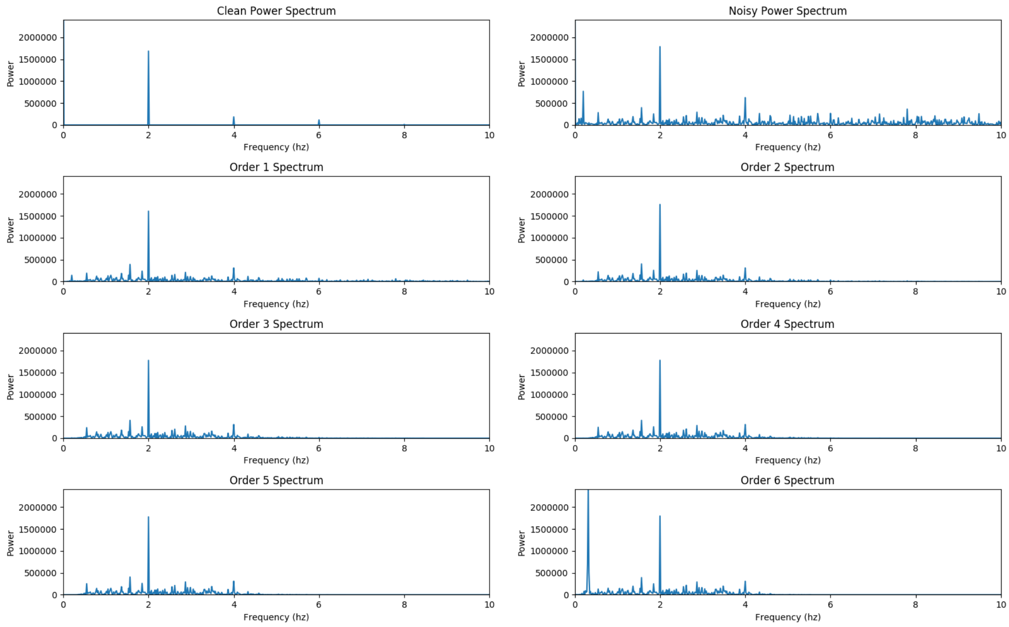


Figure 4.2: Butterworth Filter Power Spectrums.

the filter order increases, the amount of random noise present decreases. The 1st order filter has a wide range of random noise, whereas the 5th order filter removes almost all noise outside the range 0.4 to 4 hz. Interestingly, there is an artefact in the 6th order filter, where there is a spike at 0.3 hz, which is larger than the desired signal, so I will avoid using these.

I test the speed by running the filter 1000 times using python's *timeit.timeit()* function, which provides the accurate time the computer has spent evaluating a given piece of code. I run the test on the same 35 minute PPG I recorded on a run.

The results of the timing test can be seen in figure 4.3. It can be seen in the graph that generally higher order results in greater execution time. The effect is not linear though - there is a smaller increase in execution time between an odd order and the next order.

If choosing a Butterworth filter, I would opt for a 5th order filter, as it provides the most filtering, whilst not producing artefacts as seen in the 6th order filter.

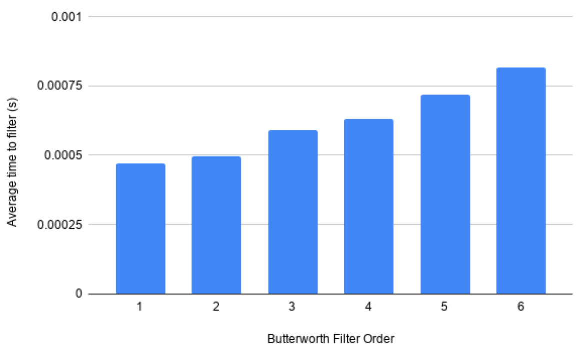


Figure 4.3: The effect of Butterworth filter order on execution time.

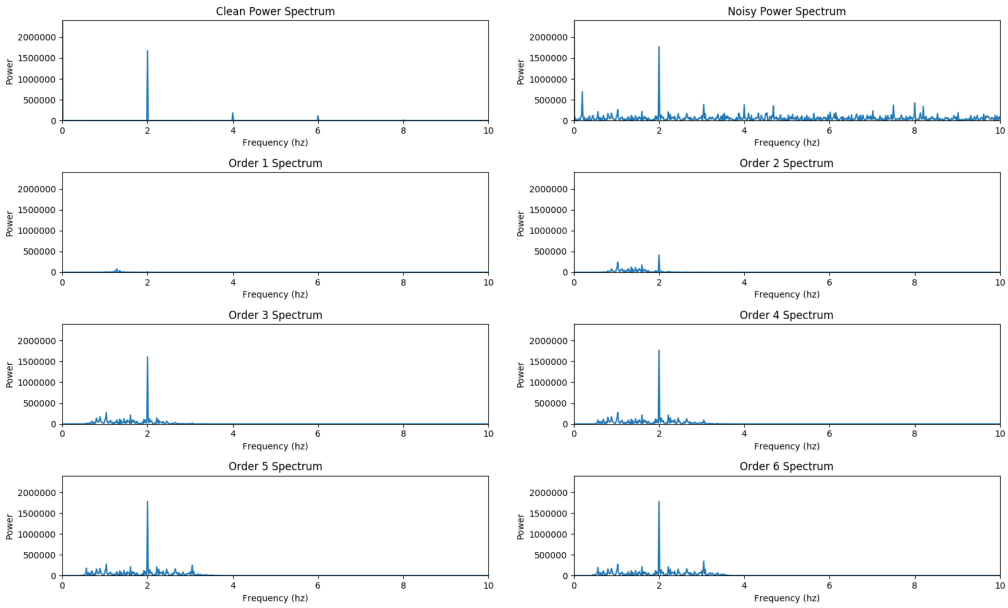


Figure 4.4: Chebyshev Filter Power Spectrums.

4.2.2 Chebyshev Type II Filter

I test the validity of the filter by again comparing the power spectrum produced by filtering the simulated heart-beat I detail above. The power spectrums are shown in Figure 4.4.

The results show that an order 1 filter is not useful, as there is almost no useful signal present. This is as a first order Chebyshev Type-2 filter provides a very narrow pass-band, resulting in almost no useful signal passing through. The order 2 filter is an improvement, and there is a clear peak at 2 hz, however it is about as powerful as the random noise, meaning we have not recovered the signal as accurately as we might want. Orders 3,4,5 and 6 are all very similar, with a very clear peak at 2 hz, and the random noise only present between 0.4 and 4 hz, as we wanted from the filter. The 2 hz peak is also much larger than the random noise, as desired. All of the filter orders effectively dealt with the low frequency noise, it is not featured within the power spectrums at all.

I run the timing test again on the Chebyshev type II filter, taking the average amount of time taken to filter the same 35 minute segment as before. The results are plotted in figure 4.5. Firstly, there is a clear linear correlation between increasing filter order and increased execution time. This implies we should choose the lowest filter order which is feasible. Secondly, note that the Chebyshev filter takes over double the time as the Butterworth filter for each order. If there is no significant filtering performance improvement to using the Chebyshev filter, we should therefore use the Butterworth filter.

If I was going to choose a Chebyshev Type II filter, I would choose an order 3 filter, as it provides sufficient filtering, without reducing the power of the useful signal too much. Additionally, any higher-order filters would take too much time.

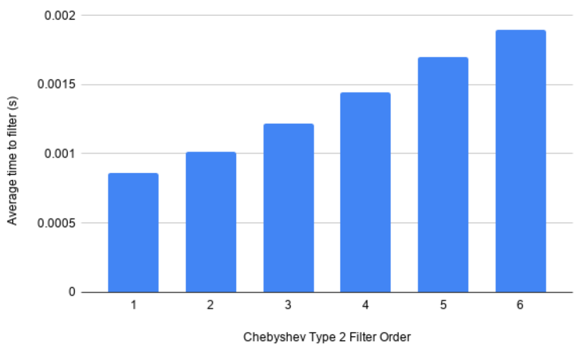


Figure 4.5: The effect of Chebyshev filter order on execution time.

4.3 Motion Artefact Reduction

In this section I talk about the work done to evaluate the effectiveness of my motion artefact reduction techniques. I first detail the tests I ran, followed by the results of these tests on the different algorithms.

I will test the algorithms on different levels of motion corruption, to determine what solution is most effective for each level. I divide the data into twenty second segments of PPG data, labelled with it's running speed. For testing motion artefact reduction, I only test data recorded while running at a speed greater than 3 km/h.

Now, to test my motion filtering algorithm, for each PPG data segment, I run the algorithm on it. With the results, I calculate the heart-rate that my heart-rate calculate algorithm would compute. I use my standard deviation minimisation algorithm from 3.4.2 and then compare this to the ground truth which is the ECG data associated with that segment, to get the error associated with it.

I test two of my motion filtering algorithm - Least Mean Squares (LMS) and Normalised Least Mean Squares (NLMS). Both of these have parameters step-size and number of taps, so I will compare both of them with a variety of step sizes and tap numbers to determine the optimal parameters. Once I have found the optimal parameters, I will compare LMS and NLMS in terms of execution time and validity.

4.3.1 LMS

Before I begin testing, I must determine which axis I should filter on. In other words, do I adaptively filter against acceleration from the x, y or z axis, or some combination of these axes. To test this, I ran the adaptive filters on all of the medium noise data I have, at various different step sizes and number of taps. For each tap number and step size, I try every possible combination of filtering using the x, y and z acceleration.

I condensed the information from these tests, by averaging the ordering of the errors. If a particular combination of axis produced the best filtering, based on average absolute error it is assigned number 1, the next best 2 and so on. Then, I take the average of this order over each segment of data, which is shown in Table 4.2.

The table shows that filtering using just the x-axis, is on average better than any other combination, with the closest average order to 1. As a result, in these tests I will just be

Table 4.2: Comparing LMS Filtering with Different Axes.

X Filtering	Y Filtering	Z Filtering	Average Order
Y	Y	Y	4.58
Y	Y	N	3.68
Y	N	Y	2.05
Y	N	N	1.15
N	Y	Y	6.33
N	Y	N	6.13
N	N	Y	4.10

filtering using the x-axis as the reference motion.

Next, I determine the number of taps which would be best for filtering the data, at low noise, medium noise and high noise. I plot the cumulative probability diagram of the error, and the magnitude error, in order to compare the error distributions. The results are in Figure 4.6.

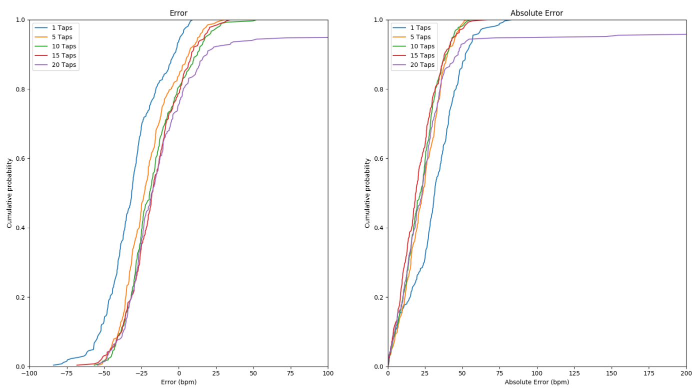


Figure 4.6: Comparison of LMS Error at Different Tap Numbers.

The results show that all of the errors are skewed towards negative values, regardless of number of taps. Looking at the absolute errors, we see that using 15 taps produces the lowest median absolute error, as the curve remains closest to error = 0. Having 1 tap produces a worse error, as we do not reduce enough of the motion noise, whereas more taps removes too much data, resulting in the error increasing.

Next, using 15 taps, I determine the best step value to use for LMS filtering. I plot the cumulative graphs again, see Figure 4.7. The results show all the step sizes under-approximate the heart-rate, however a step size of 1 provides the lowest absolute error, and hence should be used. Lower step sizes likely take too long to converge, and don't reach the correct solution, while larger step sizes likely miss the solution.

4.3.2 NLMS

To determine which axes to use, I run the same test as previously but using NLMS instead. The results are in Table 4.3. Again, filtering using just the x-axis has a much

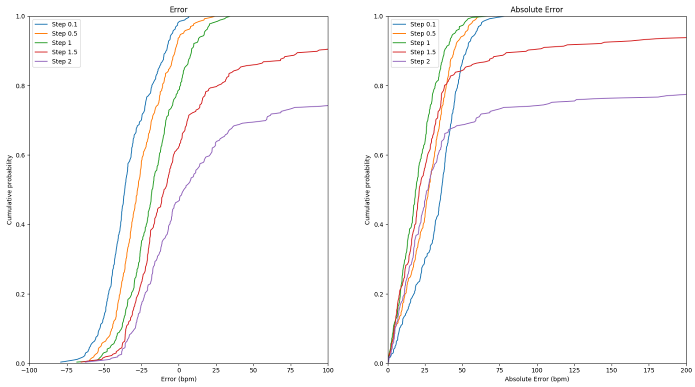


Figure 4.7: Comparison of LMS Error at Different Step Sizes.

Table 4.3: Comparing NLMS Filtering with Different Axes.

X Filtering	Y Filtering	Z Filtering	Average Order
Y	Y	Y	4.68
Y	Y	N	4.83
Y	N	Y	2.95
Y	N	N	1.83
N	Y	Y	5.20
N	Y	N	5.40
N	N	Y	3.13

lower average ordering than the other techniques, so I will filter using just the x-axis from here on.

Next I investigate the effect of the number of taps on NLMS. The results, in Figure 4.8, are very similar to what we saw for LMS, and again using 15 taps is optimal.

Now, I investigate the effect of step size on NLMS. The errors are plotted in Figure 4.9. We see that a step size of 1 is optimal, any larger and very quickly the error increases as we don't converge on the actual solution.

Now I have investigated LMS and NLMS, finding the optimal parameters. I will next look at the differences between them given their optimal parameters, both in terms of execution time and validity.

In figure 4.10 I plot the errors calculated by the algorithms. The plot shows that they both tend to underestimate the heart-rate. However, NLMS produces a slightly lower absolute error.

To test time, I use the Python module *timeit* to run the algorithms on thirty seconds of data, and time how long they take to run 10000 times over. The results are in table 4.4, showing that lms runs in less time than NLMS. Hence the choice between NLMS and LMS is a trade-off between need for quick results and need for lower errors.

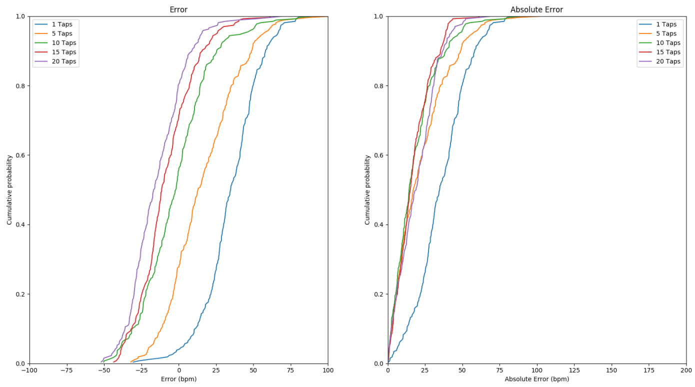


Figure 4.8: Comparing Numbers of Taps for NLMS.

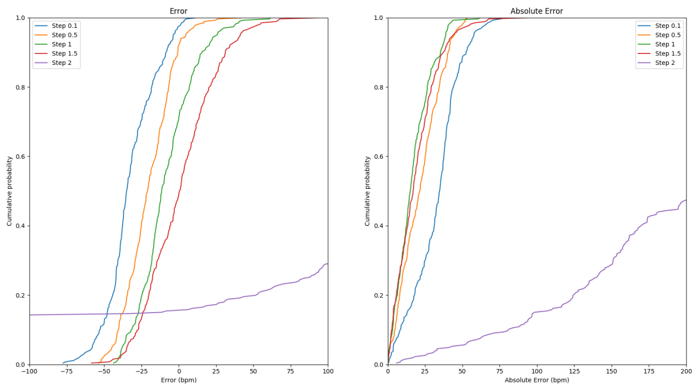


Figure 4.9: Comparing Step Size for NLMS.

4.4 Heart Rate Calculation

To evaluate my Heart-Rate calculation, I first calculate the running time of both algorithms - local maxima, and peak-peak standard deviation minimization . I also establish the effectiveness of the algorithms, when given a clean PPG signal.

For the clean data, I have recorded an hour of data using the wristwatch and ECG where I do not make large movements. I then pass it through a Butterworth filter to remove noise out of the valid range. I do not attempt to remove motion noise, as there should be minimal noise in this data.

First I evaluate the time taken, by running the algorithms on a section of data, and repeating 1000 times. As can be seen in Table 4.5, the local maxima technique is much

Table 4.4: Comparing Adaptive Filtering Running Times.

Technique	Time for 10000 iterations (s)
NLMS	44.61
LMS	34.49

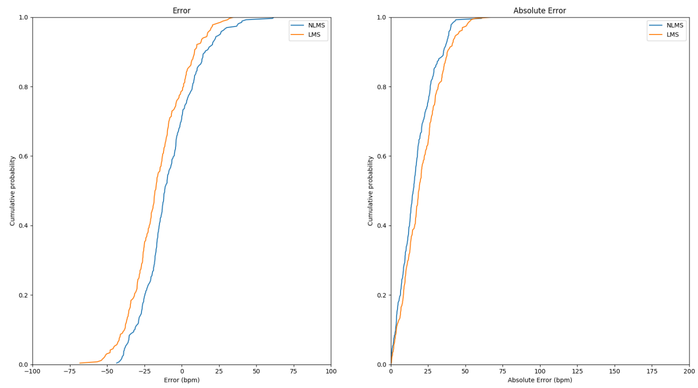


Figure 4.10: Comparison of NLMS and LMS Validity.

Table 4.5: Comparing Heart Rate Calculation Running Times.

Technique	Time for 1000 iterations (s)
Local Maxima	0.36
Standard Deviation Minimisation	76.54

less expensive. This is as it only needs to iterate through the signal once. Hence I should choose the local maxima technique if it can perform as well as standard deviation minimization.

Finally I compare the result of the heart-rate calculation to the actual heart-rate measure provided by the ECG. In Figure 4.11 I plot the error and absolute error of both methods. The data clearly shows that standard deviation minimization is superior, providing a much lower average absolute error. We see that 90% of the recorded errors are below 10bpm, whereas 90% of local maxima calculated heart-rates are below 35bpm. The error graph shows that the local maxima technique nearly always over-approximates the heart-rate, whereas the standard deviation technique tends to under-approximate.

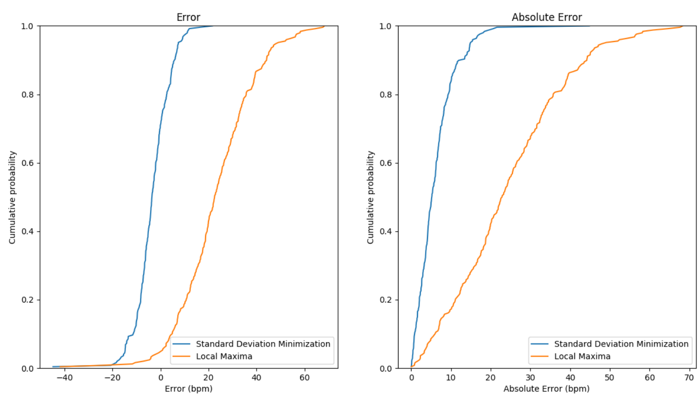


Figure 4.11: Comparison of Peak Finding Algorithms.

The reason the errors differ so much can be seen in Figure 4.12. It is clear that the local maxima technique over-approximates the number of peaks, as it often finds peaks which are either noise or not caused by the heart-rate.

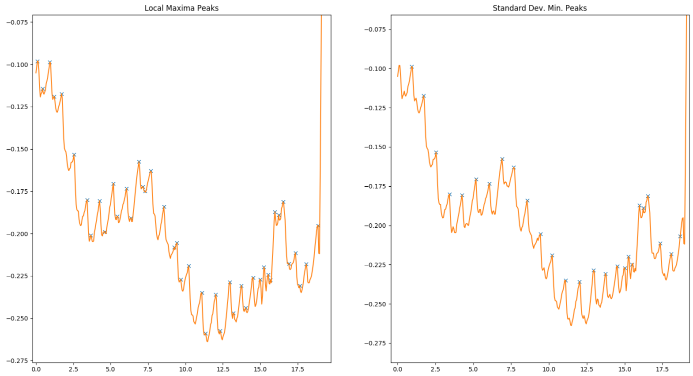
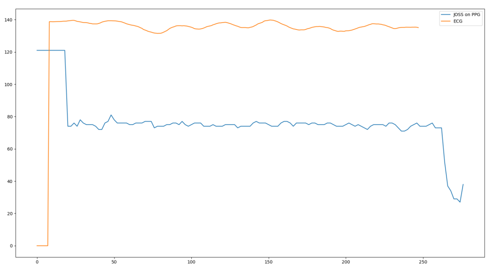


Figure 4.12: Comparison of Heart Rate Peak Calculations.

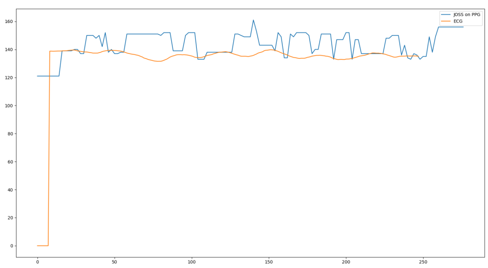
I will use the standard deviation minimization technique in my work, as it produces significantly lower errors. It does take longer, however it would still produce results in real-time on a smart-watch.

4.5 Joint Sparse Spectrum Reconstruction

To evaluate how effective spectral subtraction is for removing motion noise, I run the algorithm on a run recorded at 9 kph, with and without subtraction. The results are in Figure 4.13 and show that without subtraction, the algorithm locks into the wrong frequency, caused by a large motion artefact. However, with spectral subtraction we find that the right bpm is selected.



(a) Without Subtraction.



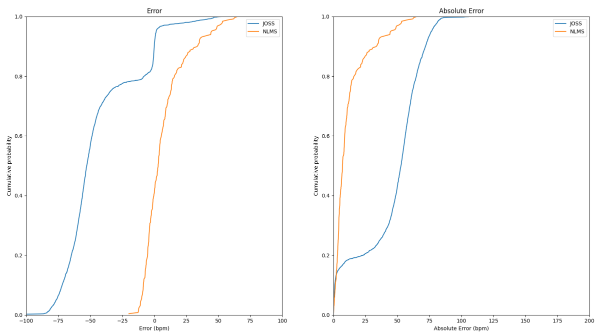
(b) With Subtraction.

Figure 4.13: Spectrum Reconstruction.

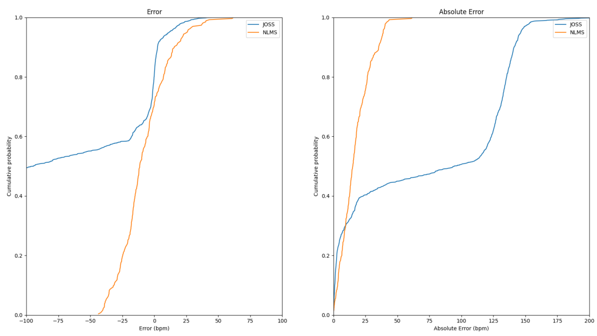
I next plot the errors, see Figure 4.14, comparing the spectrum reconstruction to NLMS, for samples taken with both high and low motion noise.

With low motion noise, NLMS is significantly better than spectrum reconstruction. I estimate this is due to the spectrum subtraction - small movements result in large and random fluctuations in the accelerometer's spectrum - which may then be subtracted from the PPG, removing the useful signal.

With approximately 30% of the high motion noise, spectrum reconstruction performs better than NLMS, these are the cases where the correct frequency is tracked. However, for the remaining data, NLMS is far better. The error plot shows that for about 50% of the data, spectrum methods produce a heart-rate between 125 and 150 bpm lower than the ECG. This is likely caused by the algorithm locking on to a lower frequency, such as motion noise - evidence that the spectrum subtraction is not completely effective.



(a) Low Motion Noise.



(b) High Motion Noise.

Figure 4.14: Comparing Spectrum Reconstruction to NLMS.

In conclusion, my NLMS method is preferable to the spectrum method, as it produces far better results. The spectrum method requires more optimisation to ensure that we never lock on to frequencies coming from noise.

4.6 Earbuds

In this section, I evaluate the performance of the earbuds alone, and then evaluate some of the fixes I propose to issues identified in the evaluation.

The first step I took when evaluating the earbuds was to go on a fast run, which would introduce much motion noise on the watch. You can see the heart-rate from the earbuds plotted against the heart-rate I received from the ECG in Figure 4.15.

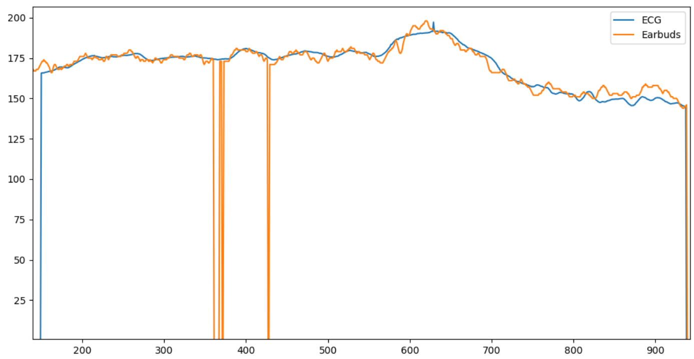


Figure 4.15: Plot of Earbud Heart-rate against ECG.

4.6.1 Problem

The problem with the earbuds is that they occasionally lose contact with the ear. This can happen as a combination of sweating and motion dislodges the earbud. In addition, I found when running I would often have to stop and adjust the earbud as I suspected it was going to fall out.

The earbuds are quite sensitive to location, so if they become dislodged or are rotated, the reported heart-rate drops to zero. This can be seen in Figure 4.15. Where the heart-rate is tracking accurately, but then suddenly just drops to zero.

4.6.2 Sensor Fusion

Figure 4.16 shows the effectiveness of the Kalman filter - when the earbud's signal drops out it follows the curve picked up by the watch sensor.

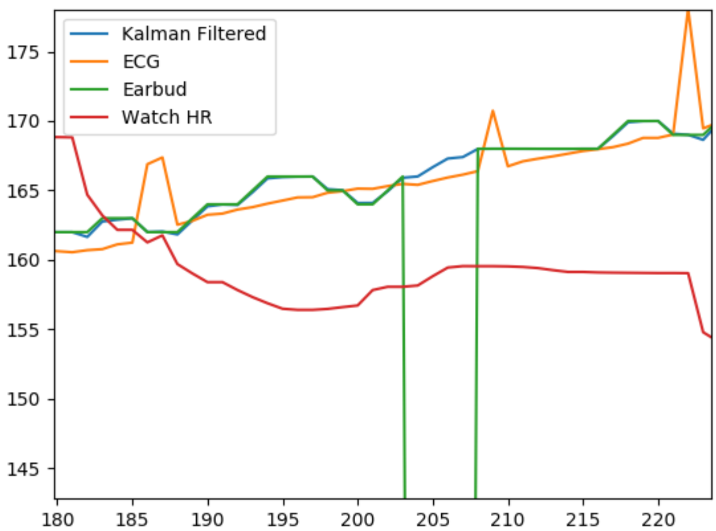


Figure 4.16: Kalman Filter Demonstration.

In Figure 4.17 we plot the errors for the watch sensor, the earbud sensor and the sensor fusion technique. The graph shows that the earbuds and hence the fusion technique produce a significantly lower error than the watch for 90% of the samples, as expected. And, for the 10% the watch is better - as the earbuds report 0 bpm. The sensor fusion technique is clearly better than the earbuds for these samples as well, using the watch to guide the output. Hence, the sensor fusion technique gets the best of both worlds.

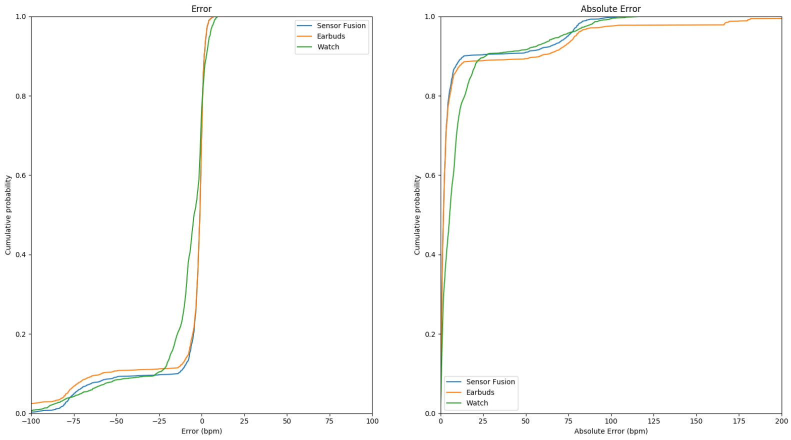


Figure 4.17: Sensor Fusion Error.

In this section I have shown that earbuds produce a very accurate signal, but require the sensor fusion method I developed in order to avoid issues where the rate drops to zero due to lack of contact.

Chapter 5

Conclusion

I have met all the success criteria set out in my project proposal.

Summary In this project I explored a few techniques which calculate heart-rate from the noisy signals coming from a smart-watch worn while running. To remove random noise not caused by motion, I experimented with band-pass filters, finding the optimal parameters for my usage. To remove motion noise, I investigated and implemented two adaptive noise cancellation techniques. Then to calculate the heart-rate I looked into two different methods. I also looked at using joint spectrum calculations to estimate heart-rate, and remove motion noise.

Finally, I showed that putting heart-rate sensors into earbuds is a promising technique, although the signal can cut out. Hence, my developed sensor fusion technique which combines heart-rate from the earbuds and from a wrist-watch provides a better signal than either alone.

Issues I found that using the wrist watch sensors alone, I could not get a good estimate of heart-rate. This is in line with other algorithms suggested in papers.

Using spectrum reconstruction appears to be effective in some cases but can latch on to the wrong frequency and never recover the correct signal. In future work, some more research might be done into determining whether the frequency we're currently tracking is reasonable.

Using two sensors, I found I could estimate heart-rate with much more confidence. Currently the sensor-fusion algorithm is not very accessible to users, as the interface is very minimal, so if I had more time I would have developed an interface which allowed users to better track their workouts.

Bibliography

[1] John Allen, John R Frame, and Alan Murray. Microvascular blood flow and skin temperature changes in the fingers following a deep inspiratory gasp. *Physiological measurement*, 23(2):365, 2002.

[2] Rohan Banerjee, Avik Ghose, Anirban Dutta Choudhury, Aniruddha Sinha, and Arpan Pal. Noise cleaning and gaussian modeling of smart phone photoplethysmogram to improve blood pressure estimation. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 967–971, 2015.

[3] H. Chung, H. Lee, and J. Lee. Finite state machine framework for instantaneous heart rate validation using wearable photoplethysmography during intensive exercise. *IEEE Journal of Biomedical and Health Informatics*, 23(4):1595–1606, 2019.

[4] Wikimedia Commons. File:puls oxymeter.jpg — wikimedia commons, the free media repository, 2013. [Online; accessed 21-February-2020].

[5] S. F. Cotter, B. D. Rao, Kjersti Engan, and K. Kreutz-Delgado. Sparse solutions to linear inverse problems with multiple measurement vectors. *IEEE Transactions on Signal Processing*, 53(7):2477–2488, 2005.

[6] Mohamed Elgendi. On the analysis of fingertip photoplethysmogram signals. *Current cardiology reviews*, 8(1):14–25, 2012.

[7] Y. Fujita, M. Hiromoto, and T. Sato. Parhelia: Particle filter-based heart rate estimation from photoplethysmographic signals during physical exercise. *IEEE Transactions on Biomedical Engineering*, 65(1):189–198, 2018.

[8] J. Kim, T. Lee, J. Kim, and H. Ko. Ambient light cancellation in photoplethysmogram application using alternating sampling and charge redistribution technique. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 6441–6444, Aug 2015.

[9] Scott Michael, Kenneth S Graham, and Glen M Davis. Cardiac autonomic responses during exercise and post-exercise recovery using heart rate variability and systolic time intervalsa review. *Frontiers in physiology*, 8:301, 2017.

[10] Hangsik Shin. Ambient temperature effect on pulse rate variability as an alternative to heart rate variability in young adult. *Journal of clinical monitoring and computing*, 30(6):939–948, 2016.

[11] B. Widrow, J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, R. H. Hearn, J. R. Zeidler, J. Eugene Dong, and R. C. Goodlin. Adaptive noise cancelling: Principles and applications. *Proceedings of the IEEE*, 63(12):1692–1716, Dec 1975.

[12] R. W. C. G. R. Wijshoff, M. Mischi, and R. M. Aarts. Reduction of periodic motion artifacts in photoplethysmography. *IEEE Transactions on Biomedical Engineering*, 64(1):196–207, Jan 2017.

[13] L. B. Wood and H. H. Asada. Noise cancellation model validation for reduced motion artifact wearable ppg sensors using mems accelerometers. In *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 3525–3528, Aug 2006.

[14] Z. Zhang, Z. Pi, and B. Liu. Troika: A general framework for heart rate monitoring using wrist-type photoplethysmographic signals during intensive physical exercise. *IEEE Transactions on Biomedical Engineering*, 62(2):522–531, 2015.

[15] Zhilin Zhang. Photoplethysmography-based heart rate monitoring in physical activities via joint sparse spectrum reconstruction. *IEEE transactions on biomedical engineering*, 62(8):1902–1910, 2015.

Appendix A

Project Proposal

Charlie Maclean
King's
cm927

Diploma in Computer Science Project Proposal

Synthesis of Heart-Rate Detection Methods

December 18, 2019

Project Originator: Dr Robert Harle

Project Supervisor: *Dr Robert Harle*

Signature:

Director of Studies: *Dr Timothy Griffin*

Signature:

Overseers: *Dr Anil Madhavapeddy* and *Professor John Daugman*

Signatures: *<no need to obtain Overseers' signatures yourself>*

Introduction and Description of the Work

Heart-rate signals from watches are unreliable while exercising. Watches make use of photoplethysmography (PPG) sensors - sensors which detect the volume of blood in the skin and use variances in this to reconstruct a heart-rate. PPG sensors are preferred to the more accurate electrocardiogram (ECG) due to user comfort. However, the signals they provide are harder to process - I want to compare strategies to process these signals to extract heart rate.

There are several sources of noise within a PPG signal. There is often high frequency contamination caused by electrical interference or light from external sources. Additionally, there is a constant low frequency variation in the DC background of the signal, as a result of capillary density, blood volume and temperature variations.

In the context of running, motion caused by the arms swinging forward and back causes the sensor to slide along the skin, creating motion artifacts (MAs). These are particularly challenging as they can have a much larger amplitude than the pulse we are looking for. Additionally, they can be at the same frequency as the heart rate signal, making them challenging to filter out. Therefore, there is research [1][2] that suggests using accelerometer data in order to predict MAs. I will implement algorithms which find the MAs based on data from the accelerometer. Following this, I will look into the implementation of filters to remove these MAs.

In order to get accurate heart-rate measurements with which to compare the PPG signals, I will make use of an chest-mounted portable ECG. I will need to synchronize the signals received from the ECG and PPG, due to clock skew between the different devices. Potentially, there could be drift between the two clocks as well.

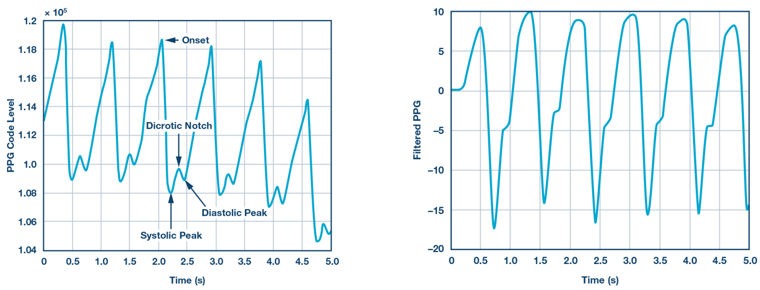


Figure 1: PPG signals before and after filtering [3]

Starting Point

Wearable Development

I will need to develop an application for the smart watch which will record PPG and motion signals without interruption. I will use Android Studio and Kotlin.

I have never used Kotlin before and have used Android Studio once before, but never to develop something for a wearable.

Digital Signal Processing

Manipulation of the PPG signals I receive will require much digital processing, and there are two languages I am considering using: MATLAB and Python.

I have never used MATLAB before, but I am familiar with Python.

Substance and Structure of the Project

Core

1. Developing a wearable application to capture PPG signals and motion data. I would like to develop this using Android Studio and Kotlin. I will need to make use of wake locks in order to ensure that the application can continuously record data.

2. Collecting data using a PPG-enabled watch and a portable ECG. I will record my heartbeat over the course of several runs.

3. Synchronizing signals. Implementation of an algorithm to synchronize the data output from the watch records to the data output from the portable ECG. I will have to develop a program to find the lag between the two signals.

4. Removing noise. Due to the various sources of noise, I will investigate potential low and high pass filters, to remove both high and low frequency disturbance.

5. Peak finding - implementing algorithms to find the actual beat given a clean PPG signal. Some potential algorithms are:

    (a) Adaptive threshold [4]

    (b) Wavelet transformation [5]

6. Finding MAs - implement an algorithm which uses the accelerometer in order to detect segments of the PPG which are likely to have been affected by motion.

7. Removal of MAs - implementation of filters to remove the previously detected MAs.

Possible Extensions

- Investigating PPG-enabled earbuds
  - Evaluation of the quality of heart-rate provided by earbuds
  - Exploring the potential to merge signals from a smartwatch and earbuds in order to provide a higher quality signal.
- There is research [6] to suggest gyroscope information is also helpful in filtering out MAs. I could include gyroscope data in my MA filtering technique.
- Comparing more heart-rate detection algorithms:
  - Digital filters
  - Adaptive filters
  - Singular value decomposition
  - Empirical mode decomposition
  - Spectrum analysis

References

[1] Z. Zhang. Photoplethysmography-based heart rate monitoring in physical activities via joint sparse spectrum reconstruction. *IEEE Transactions on Biomedical Engineering*, 62(8):1902–1910, Aug 2015.

[2] Z. Zhang, Z. Pi, and B. Liu. Troika: A general framework for heart rate monitoring using wrist-type photoplethysmographic signals during intensive physical exercise. *IEEE Transactions on Biomedical Engineering*, 62(2):522–531, Feb 2015.

[3] Foroohar Foroozan. Music-based algorithm for on-demand heart rate estimation using photoplethysmographic (ppg) signals on wrist, 2018.

[4] Ivaylo I. Christov. Real time electrocardiogram qrs detection using combined adaptive threshold. *BioMedical Engineering OnLine*, 3(1):28, 2004.

[5] Jake D. Campbell, Christopher G. Pretty, J. Geoffrey Chase, and Phillip J. Bones. Near-real-time detection of pulse oximeter ppg peaks using wavelet decomposition. *IFAC-PapersOnLine*, 51(27):146 – 151, 2018. 10th IFAC Symposium on Biological and Medical Systems BMS 2018.

[6] Alexander J. Casson, Arturo Vazquez Galvez, and Delaram Jarchi. Gyroscope vs. accelerometer measurements of motion from wrist ppg during physical exercise. *ICT Express*, 2(4):175 – 179, 2016. Special Issue on Emerging Technologies for Medical Diagnostics.

Success Criteria

The following should be achieved:

- Develop an application which records and stores PPG signals on a watch.
- Create program which synchronises ECG signals with PPG signals.
- Implement at least two filtering algorithms, demonstrate filtering works by displaying signals before and after filtering.
- Implement at least two peak finding algorithms, demonstrate they work by comparing peaks on the PPG signal to peaks on the ECG signal.
- Implement a MA detection algorithm.
- Be able to remove MAs, demonstrating that the peak finding algorithm is not affected by signals caused by motion.

Resources Required

I will use my own laptop. I will regularly backup my project to GitHub and an external HDD, so that I can recover data in the event of hardware failure. I accept full responsibility for this machine and and I have made contingency plans to protect myself against hardware and/or software failure.

I will use a smart watch, and a chest-mounted ECG sensor in order to collect data.

Timetable and Milestones

Weeks 1-2 (28/10/19 - 10/11/19)

- Project set-up:
  - Installation and setup of Android Studio.

  - Investigate and install an IDE for Python or MATLAB.
  - Use GitHub to set up a backup system for the project.
  - Learn to use Kotlin for the wearable app.
- Create a wearable app to record and store PPG and motion data, using Android Studio and Kotlin. Research into wake logs and interface design.
- Additionally, test application by recording data while running.

Weeks 3-4 (11/11/19 - 24/11/19)

- Collect data from runs.
- Build program to synchronize data between ECG and PPG (based on either heart rate variation or motion information).
- Researching and then implementing filters to remove both high and low frequency noise from the PPG signal.

Weeks 5-6 (25/11/19 - 08/12/19)

- Implement program to detect peaks in the signal.
- Begin writing dissertation document.

I will take the next week off.

Weeks 7-8 (16/12/19 - 29/12/19)

- Implement MA detection algorithm.
- Continue writing dissertation.

Weeks 9-10 (30/12/19 - 12/01/20)

- Implement MA removal algorithm.
- Continue writing dissertation.

Weeks 11-12 (13/01/20 - 26/01/20)

- Begin extension work, testing earbud HR detection against wrist-watches and portable ECGs.
- Continue writing dissertation.

Friday 31st January - Progress Report due

Weeks 13-14 (27/01/20 - 09/02/20)

- Extension work, testing whether earbud HR can be merged with wristwatch signal to create a more accurate output.
- Continue writing dissertation.

Weeks 15-16 (10/02/2020 - 23/02/2020)

- Extension work, including gyroscope feedback into the MA detection algorithm.
- Continue writing dissertation.

Weeks 17-18 (24/02/2020 - 08/03/2020)

- Extension work, including gyroscope feedback into the MA detection algorithm.
- Continue writing dissertation.

Weeks 19-20 (09/03/2020 - 22/03/2020)

- Writing dissertation.

Weeks 20-21 (23/03/2020 - 05/04/2020)

- Writing dissertation, aim to have first draft by second week of Easter holiday.

Weeks 22-26 (06/04/2020 - 08/05/2020)

- Continue improving dissertation.
- Leaving time for exam preparation.

Final deadline for dissertation 08/05/2020