VRIJE
UNIVERSITEIT
BRUSSEL

Bachelor of Science in Engineering Technology:

PHOTOPLETHYSMOGRAPHY
Engineering Programming task

Samuel Gérin

2023-2024

ENGINEERING SCIENCES

Declaration

During the preparation of this work, the author(s) used the following tool(s) and service(s):

name of tool/service		reason
ChatGPT	in order to:	*debug code*
Microsoft Copilot	in order to:	*debug code*
DeepL Write	in order to:	*correct spelling and grammar errors*

After using this tool(s) and/or service(s), the author(s) reviewed and edited the content as needed. The author(s) take(s) full responsibility for the content of this work.

Contents

Disclosure on the use of AI i

Contents ii

1 Photoplethysmography 1
- 1.1 Introduction . . . 1
- 1.2 Experiment details . . . 1
- 1.3 Video data . . . 2
- 1.4 Preprocess . . . 4
- 1.5 PPG waveform and processing . . . 5
  - 1.5.1 Butterworth band pass filter . . . 5
  - 1.5.2 Fast Fourier transform . . . 6
  - 1.5.3 Power spectral density analysis . . . 7
  - 1.5.4 Second iteration . . . 9
- 1.6 Percentile ranking . . . 11
- 1.7 BPM over time . . . 13
  - 1.7.1 Rest . . . 13
  - 1.7.2 Active . . . 15
    - 1.7.2.1 30 seconds . . . 16
    - 1.7.2.2 1 minute . . . 17
    - 1.7.2.3 2 minutes . . . 18
    - 1.7.2.4 5 minutes . . . 19
- 1.8 Program . . . 20
  - 1.8.1 Introduction . . . 20
  - 1.8.2 Features . . . 20
  - 1.8.3 Installation . . . 20
  - 1.8.4 Usage . . . 20
    - 1.8.4.1 Calculate the average BPM over the entire video . . . 20
    - 1.8.4.2 Calculate the BPM over 5 windows . . . 21

Chapter 1

Photoplethysmography

1.1 Introduction

Photoplethysmography (PPG) combines the Greek words for light ("photos"), enlargement ("plethysmos"), and writing ("graphein") to describe a technology that uses light to measure changes in blood volume, commonly applied in heart rate monitors like watches and pulse oximeters. These devices emit light and capture its reflection, with blood absorbing more light during a heartbeat, allowing heart rate calculation.

This project involves implementing a heartbeat monitor in Python. Modern smartphones, equipped with a camera and white LED, can also perform PPG by illuminating tissue and detecting reflected light. PPG, a non-invasive technique since the 1930s, monitors blood volume changes during cardiac cycles, aiding in the measurement of blood pressure, heart rate, respiration, and blood oxygen levels. Reflection mode PPG, with the light source and detector side-by-side, is particularly effective due to shallow light penetration, and replacing the detector with a camera enables real-time, large-area PPG imaging. This project leverages smartphone technology for reflection photoplethysmographic imaging.

1.2 Experiment details

A consumer grade (Iphone SE 2020) cellular phone was used in this study.

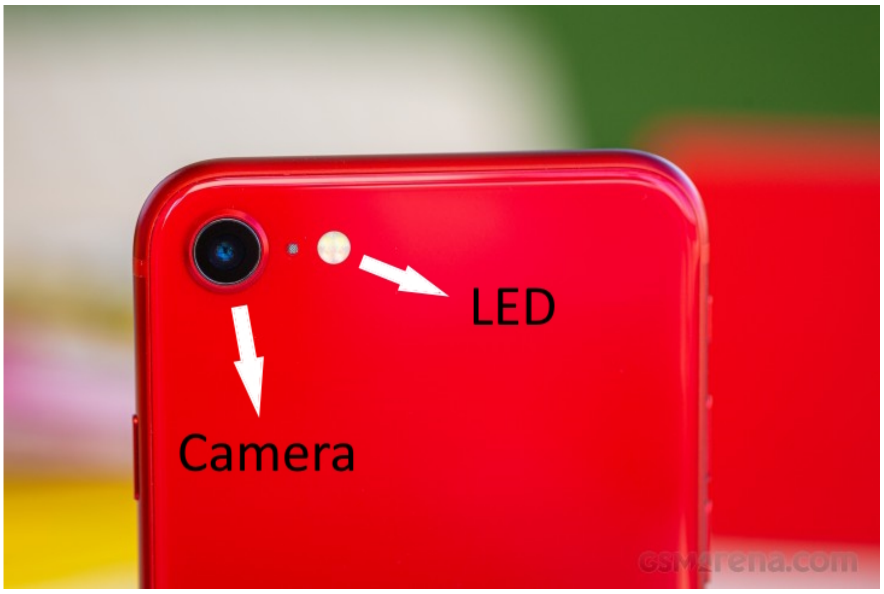

Figure 1.1: Used camera and LED

The unit (fig. 1.1) consists of a WLED as the illumination source next to a 7.0 megapixel camera at a centre-to-centre separation of around 15 mm. The phone supports colour video recording at about 30 and 60 frames per second at two user-selectable resolutions of 1920 × 1080 and 3840 x 2160 pixels. For this study a video of 30 fps and a resolution of 1920 x 1080 (full HD) will be chosen. The created program also works on the higher resolution with 60 fps but will tremendously increase the process time. Since there is no need for such accuracy, it is recommended to use lower resolutions with lower frame rates in the program.

A colour video of the volunteer's index finger placed across both the WLED and camera aperture was recorded in H.264 MOV format, the standard format for iPhone, onto the phone memory. In addition to recording movies while at rest, the post-exercise heart rate will also be determined. Furthermore, the two calculated heart rates will be incorporated into the Pulse dataset and their percentile ranking will be determined.

1.3 Video data

The video is taken from a fellow student's finger after doing 3 minutes of exercises. The data was extracted out of the video using the OpenCV library. OpenCV makes operation on each individual frame of the video possible. This is because OpenCV turns the frame in a three dimensional NumPy array (faster than Python lists for numerical operations but have more restrictions compared to them). This is 3D because we are using color images (RGB). The 3D array can be split in three 2D array each representing a individual color channel: red, green and blue .

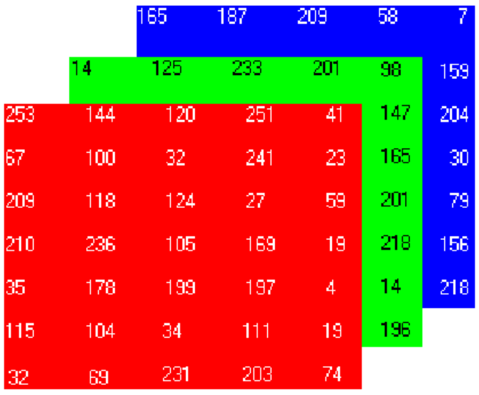

Figure 1.2: A three-dimensional RGB matrix

The red channel of the video can be extracted, returning values in a grid-like format representing the pixel values of the frame, ranging from 0 to 255 (8-bit) with 255 representing the reddest value. The NumPy library can be employed to take the mean of these values, which corresponds to the relative blood volume in the veins per frame.

In the Python script, it is preferable to utilise NumPy arrays in preference to regular lists. However, since the peaks list has a length of $length\_video \cdot fps$, they can become quite lengthy, which may result in a slow process time. NumPy arrays are known to be faster than lists for computing, but this is achieved at the expense of certain trade-offs. These include a fixed size, homogeneous data type requirements, reduced flexibility for non-numeric data, potential integration issues, and so on. In the event that the operations cannot be performed with NumPy arrays, lists will be employed. The code is presented below for reference.

```python
class VideoProcessor():
    def __init__(self, filepath):
        self.filepath = filepath
        self.cap = cv.VideoCapture(self.filepath)
        self.fps = self.cap.get(cv.CAP_PROP_FPS)

    def get_data(self):
        peaks = []

        while True:
            isTrue, frame = self.cap.read() # returns boolean that says if frame is successfully read, and the frame

            if not isTrue:
                break

            b, g, r = cv.split(frame) # split three channels, opencv uses BGR format
            onlyred = cv.merge([b*0, g*0, r])
            peaks.append(np.mean(onlyred))

        self.cap.release() # stop capturing frames

        peaks = np.array(peaks) # convert to NumPy array for faster iterations

return peaks
```

Plotting the peaks over time gives us the following graph (fig. 1.3).

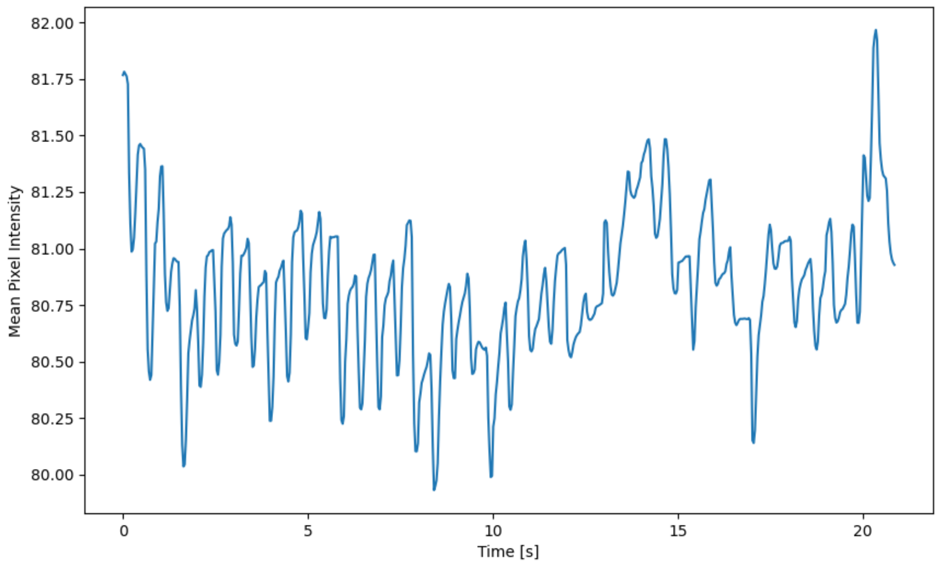

Figure 1.3: Red channel mean pixel intensity

1.4 Preprocess

If the average BPM over a full video needs to be calculated we can use the data that can be seen in the previous figure (fig. 1.3). But if we want to see the change in BPM over time. We must divide the data in seperate windows. These windows, for ease of use, must be of the same length. This is why we are gonna preprocess the data and crop it so that the video will always be a multiple of 10 seconds. This ensures that we can always equally divide the video into 5 parts. In my program, the user can choose between one or five windows. One window gives the average BPM over the whole video, while five windows allow the user to see the change in BPM over five windows of the full video. The cropping has been implemented in the code below.

```
class PreProcesser():
    @staticmethod
    def crop(lst, fps):
        length_video = round(len(lst) / fps)
        shortend_video = length_video - (length_video%10)

        new_lst = lst[:(shortend_video*round(fps))]

        return new_lst, int(len(new_lst)/fps)
```

This will turn our original 22 second video in a 20 second video which can be seen below. (fig. 1.4)

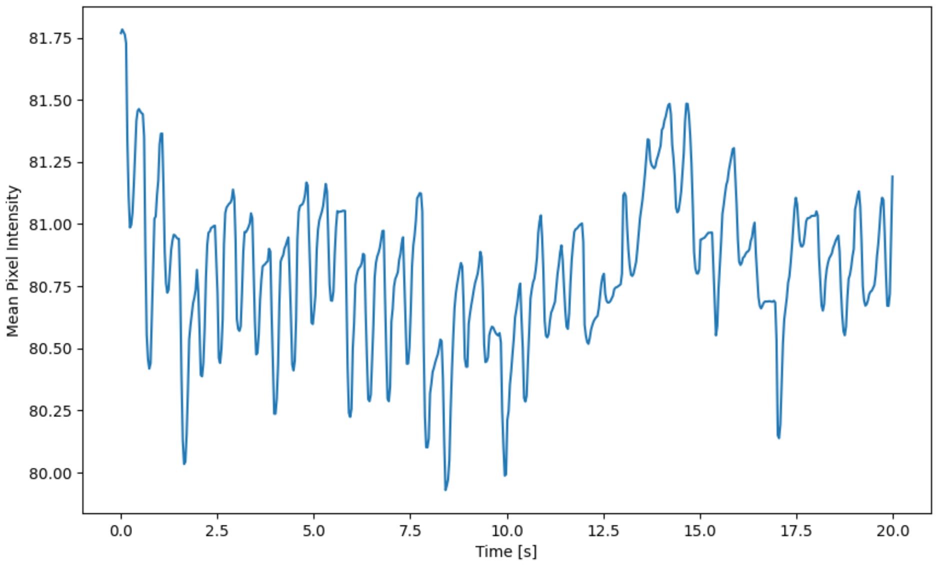

Figure 1.4: Red channel mean pixel intensity

With this data further analysis will be done.

1.5 PPG waveform and processing

1.5.1 Butterworth band pass filter

To further analyze this data waveform processing included filtering with a Butterworth band pass filter will be done. An order of 8 and a frequency pass band of 0.08 to 7 Hz will be employed. Band pass filtering simultaneously suppresses high frequency noise in the signal and quasi-DC signals, such as those caused by finger movement or changes in venous pressure. The code below applies the Butterworth band pass filter using the SciPy library.

```
    @staticmethod
    def filter(signal, fps, lstLCHC, name, fors):
        order = 8
        nyq = 0.5 * fps # calculate Nyquist frequency, half of sampling rate (fps)
        filtered_signal = []

        for lst in lstLCHC:
            lowcut = lst[0]
            highcut = lst[1]
            low = lowcut / nyq # normalized cutoff frequencies by dividing with nyq
            high = highcut / nyq # "
            sos = butter(order, [low, high], btype='band', analog=False, output='sos')  # design Butterworth bandpass filter
            filtered_signal.append(sosfiltfilt(sos, signal)) # apply filter to the signal

        return np.array(filtered_signal), fig
```

This will give us the following plot (fig. 1.5).

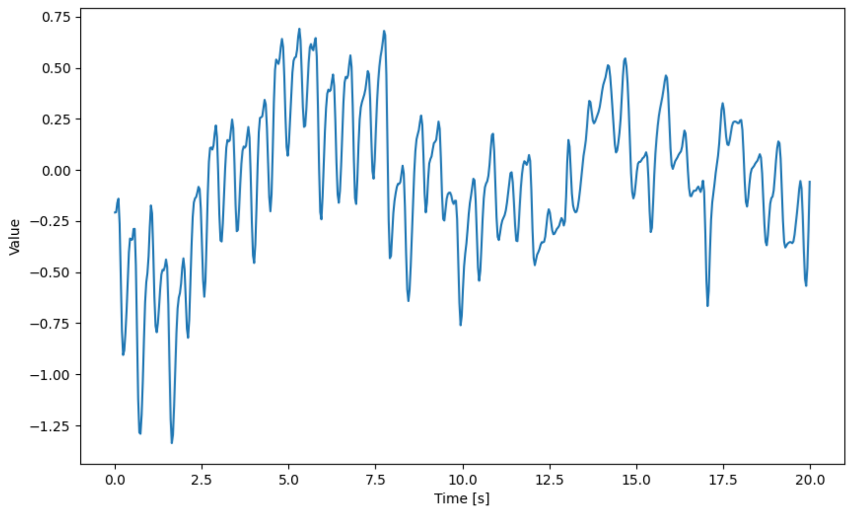

Figure 1.5: First analysis filtered

1.5.2 Fast Fourier transform

Subsequently, a fast Fourier transform (FFT) analysis was conducted on the data. The FFT is an efficient algorithm for computing the Discrete Fourier Transform (DFT) and its inverse. While the DFT transforms a sequence of values into components of different frequencies, it is computationally expensive with an order of complexity of $O(N^2)$. The FFT reduces this to an order of complexity of $O(N \log N)$, making it much faster for large datasets. The FFT is computed with the following code:

```python
    @staticmethod
    def fft(signal, windows, fps, name):
        fft_result = []
        frequency = []
        fft_resultm = []
        figs = []

        for i in range(windows):
            window_start = int(len(signal) / windows) * i
            window_end = int(len(signal) / windows) * (i + 1)
            window_signal = signal[window_start:window_end]

            fft_result.append(np.fft.fft(window_signal))
            freq, fft_resultm_temp, N = SignalProcessor.limit_frequency_range(np.abs(fft_result[i]), fps) # take abs since np.ftt.ftt returns complex number
            frequency.append(freq)
            fft_resultm.append(fft_resultm_temp)

            figs.append(Plot.plot(xvals = frequency[-1], signal = (np.abs(fft_resultm[-1])),
                                  xticks = np.arange(0, 4.0, 0.5),
                                  xlabel = "Frequency [Hz]", ylabel = "Magnitude",
                                  filename = name + "_ftt" + str(i), save = True))

        return np.array(fft_resultm), np.array(figs), np.array(frequency)
```

It can be observed that in the FFT function, the function "limit_frequency_range" is called. This is a function that filters out all of the frequency values that would correspond to a BPM larger than 240, which is obviously not possible. The code for that filtering function is provided below:

```python
    @staticmethod
    def limit_frequency_range(signal, fps):
        N = len(signal)
        frequency_index = np.arange(N)

        frequency = (frequency_index * fps) / N
        mask = frequency <= 4  # mask +240bpm
        frequency = frequency[mask]
        signal = signal[mask]

        return frequency, signal, N
```

After applying FFT we get the following graph:

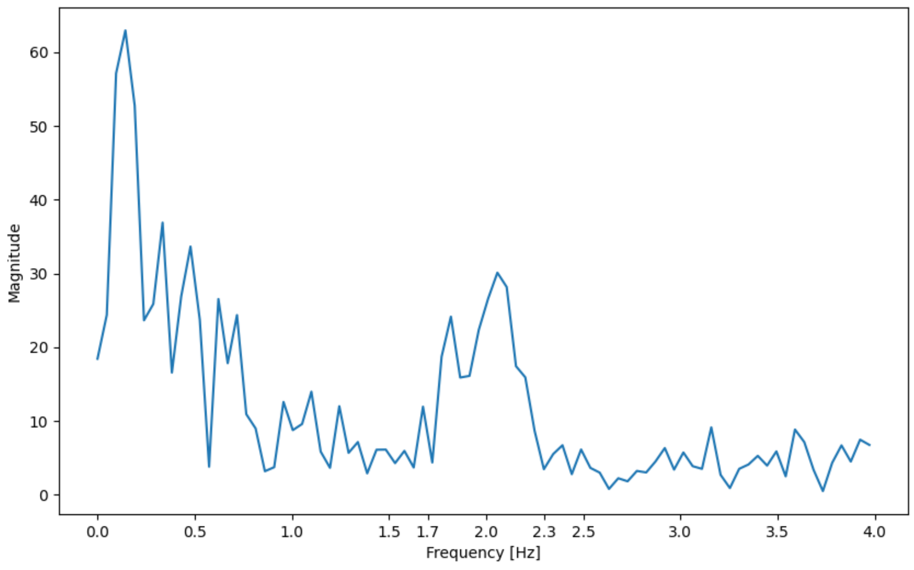

Figure 1.6: First analysis FTT

This figure illustrates the various frequencies present in the signal. It can be observed that almost every frequency is represented in the signal, indicating the presence of considerable noise. A spectral density analysis (PSD) would provide a more detailed representation, as it depicts the power associated with each frequency, rather than the magnitude.

1.5.3 Power spectral density analysis

PSD is a measure that describes the distribution of power across different frequencies in a signal or time series. It quantifies the signal's power content in terms of energy per unit frequency, providing insight into the dominant frequencies and the overall energy distribution within the signal. Since we have already calculated the FTT, calculating PSD is a relatively straightforward process. The following code is used to perform PSD analysis on our Fourier-transformed signal.

```
        @staticmethod
    def psd(signal, windows, frequency, fps, name):
        psd_result = []
        max_index = []
        figs = []
        frequency_max_indexes = []

```

```python
        for i in range(windows):
            window_start = int(len(signal) / windows) * i
            window_end = int(len(signal) / windows) * (i + 1)
            window_signal = signal[window_start:window_end]

            _, _, N = SignalProcessor.limit_frequency_range(signal[i], fps)
            psd_result.append(np.abs(window_signal) ** 2 / N)
            max_index.append(np.argmax(psd_result[-1])) # argmax returns max index

            figs.append(Plot.plot(xvals = frequency[i], signal = psd_result[-1][-1],
                                  xticks = np.arange(0, 4.0, 0.4),
                                  xlabel = "Frequency [Hz]", ylabel = "Power",
                                  filename = name + "_psd" + str(i), save = True))

        for i, index in enumerate(max_index):
            frequency_max_indexes = frequency[i][max_index]

        return frequency_max_indexes, np.array(figs)
```

Plotting our signal gives us the following graph (fig. 1.7).

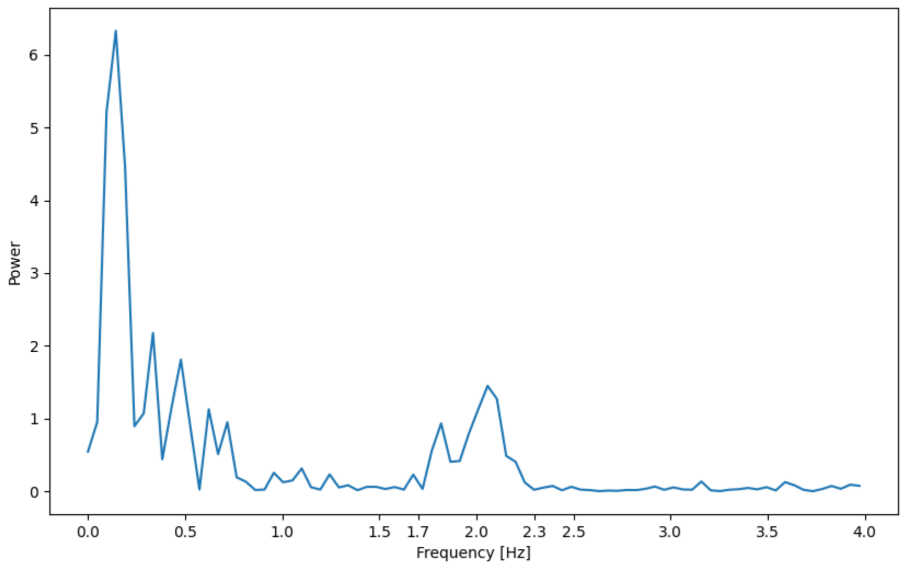

Figure 1.7: First analysis PSD

It is evident that a significant proportion of the frequencies observed in the FTT have been eliminated, leaving only the dominant frequencies. Upon analysis of the graph (fig. 1.7), it becomes apparent that there are pronounced peaks below the 0.5 Hz mark. However, it is important to note that these peaks do not align with the heart rate, as a BPM below 30 is implausible. The frequencies observed in PPG signals below 0.5 Hz typically represent slower physiological phenomena related to cardiovascular and respiratory control. These include respiratory rate variations, vasomotor oscillations and baroreflex activity.

Upon further analysis, another increase in power is observed between 1.7 Hz and 2.3 Hz. This is likely to correspond to the frequency of the heartbeat, given that the video was taken post-exercise.

Having established this, we can now apply a low-cut of 1.7 Hz and a high-cut of 2.3 Hz to our data, thereby enabling us to focus on the frequencies of the heartbeat and obtain a more accurate result.

1.5.4 Second iteration

Following the reprocessing of the data for a second iteration, in which a precise low-cut and high-cut were defined following the analysis of the graph, the following results were obtained:

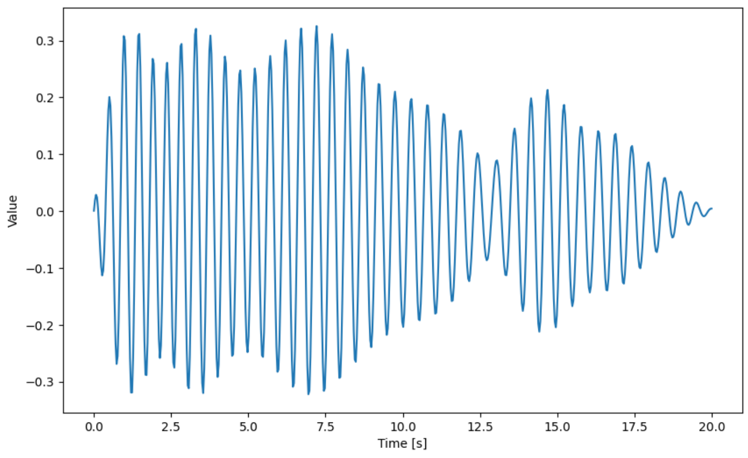

Figure 1.8: Second analysis filtered

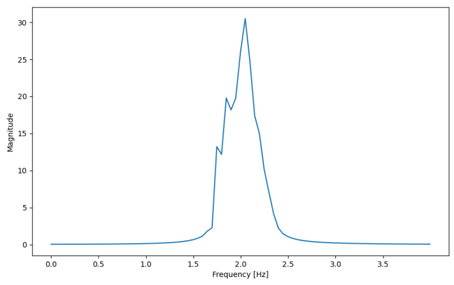

Figure 1.9: Second analysis FTT

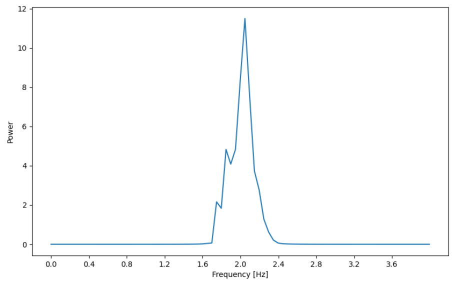

Figure 1.10: Second analysis PSD

A visual inspection of the filtered graph (fig. 1.8) reveals the presence of clear, continuous peaks. The data of interest has been successfully extracted, and the subsequent calculation of the average BPM over the entire video can now be undertaken. This will be achieved using the following code, following the transmission of the frequency with the maximum power in PSD.

```
    @staticmethod
    def bpm(frequency, length_vid, windows):
        lst = []
        figs = []
        start = True

        if windows != 1:
            for freq in frequency:
                if start is True:
                    lst.append(round(freq*60))
                    start = False

                lst.append(round(freq*60))

            xvals = np.arange(0, length_vid+length_vid/windows,
length_vid/windows)

            fig = Plot.plot(xvals = xvals, signal = lst,
                            xlabel = "Time [s]", ylabel = "BPM",
                            filename = "BPM", save = True)

        else:
            fig = round(frequency[0]*60)

        return fig
```

The aforementioned process yields the following result for the post-exercise video: "The average BPM over the entire video is 123 BPM." This entire signal processing workflow can be applied to all types of videos. The only adjustment that needs to be made is the lowcut and highcut. The process can be represented by the following schematic.

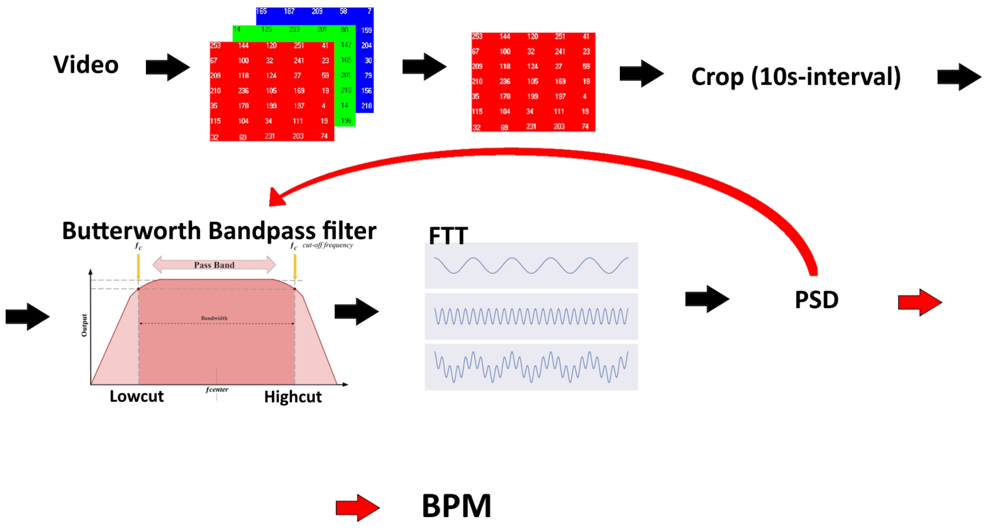

Figure 1.11: Schematic

1.6 Percentile ranking

One of the objectives is to determine the percentile ranking of a resting heart rate and a post-exercise heart rate. Given that the post-exercise heart rate is already known, the same processing steps can be applied to the video of the heart rate in rest. This approach yields the following results:

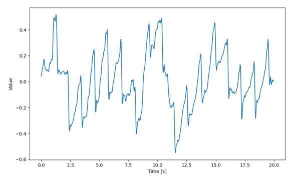
Figure 1.12: First analysis filtered

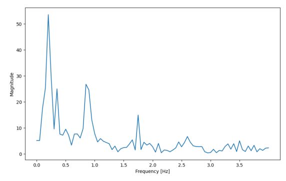
Figure 1.13: First analysis FTT

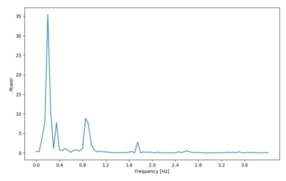
Figure 1.14: First analysis PSD

Upon examination of the PSD graph (fig. 1.10), it becomes evident that peaks are observed at sub-0.5 Hz levels. However, it is already known that these peaks are caused by slower physiological phenomena, such as breathing. Therefore, it can be assumed that these peaks are not significant and can be disregarded. Following this, a single prominent peak and a smaller peak are observed. Given that the video is in a resting state and that the second peak is smaller than the first. It can be concluded that the second peak is also insignificant and can be neglected. The next low-cut and high-cut frequencies were chosen to be 0.8 and 1.1 Hz, respectively, resulting in the following graphs.

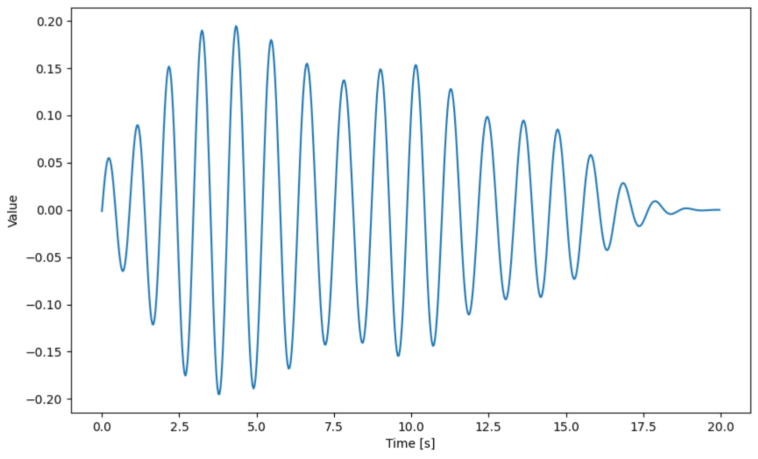

Figure 1.15: Second analysis filtered

The filtered graph (fig. 1.15) demonstrates that continuous peaks remain, so we succesfully extracted the data we are interessted in. Calculating the BPM with this data gives us the following result: 51 BPM. Which is an extremely good resting hearth rate for a fit young person.

We can now calculate the percentile ranking based on this dataset. In the following code the library Pandas is used to read the dataset. Pandas is a common library used to read larger datasets in an efficient way, allowing for easy operations involving that particular dataset. The code is as follows:

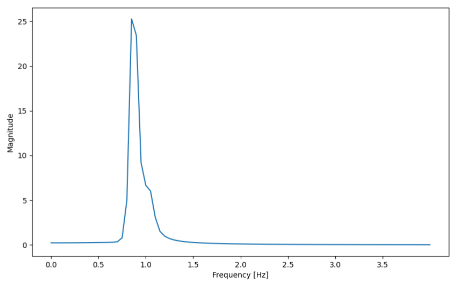

Figure 1.16: Second analysis FTT

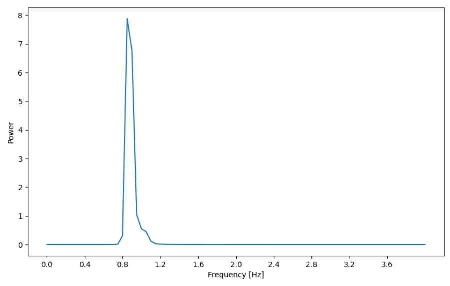

Figure 1.17: Second analysis PSD

```python
class PercentileRanker():
    @staticmethod
        dataset_url = "https://pmagunia.com/assets/data/csv/dataset-72971.csv"
        pulse_data = pd.read_csv(dataset_url)

        pulse_RestValues = pulse_data['Rest'] # rest column in dataset
        pulse_ActValues = pulse_data['Active'] # active column in dataset

        resting_percentile = (pulse_RestValues < restBPM).mean() * 100
        post_exercise_percentile = (pulse_ActValues < activeBPM).mean() * 100

        print("Your resting heart rate percentile ranking:", resting_percentile)
        print("Your post-exercise heart rate percentile ranking:", post_exercise_percentile)
```

Which gives us the following result:

Table 1.1: Data percentile ranking

activity	BPM	percentile ranking
post-exercise	123	93,5%
rest	51	2.2%

So compared to the dataset the post-exercise (active) heart rate is lower than approximately 93.5% of the individuals in the dataset. In other words, only 6.5% of the individuals in the dataset have a lower post-exercise heart rate. A high percentile ranking in post-exercise heart rate indicates that the heart rate after exercise is lower compared to the majority of the population in the dataset. This could suggest efficient cardiovascular recovery after exercise, which is often associated with good fitness levels.

For the hearth rate in rest the percentile ranking is 2.2%. In other words, 97.8% of the individuals in the dataset have a higher resting heart rate. A low percentile ranking in resting heart rate typically indicates a lower resting heart rate compared to the majority of the population in the dataset, which can be a sign of good cardiovascular fitness in some contexts.

1.7 BPM over time

Lastly for this task we need to capture our rest hearth rate followed by heart rates after 30 seconds, 1 minute, 2 minutes, and 5 minutes of exertion. The goal in this is to see the drop in BPM. If we would do this just using the average BPM of the full video, a drop will be less visible. That is why we will be using windows. Our video will be split in 5 different windows, allowing us to do the same processing techniques but now on five separate smaller videos. The code given previously already has this feature implemented. To ensure that we have enough data the video that we will use must be of a minimum 30 second length, this way the windows are of a +6 seconds size.

1.7.1 Rest

The first video we will analyze is from a video taken in rest.

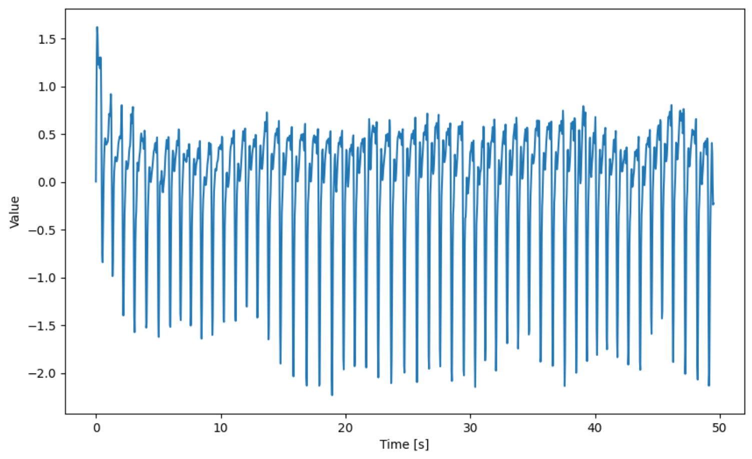

Figure 1.18: First analysis filtered

The figure (fig. 1.18) represents the full video filtered. Now since we want to see the change in BPM we will process the video in the different windows we mentioned earlier. Doing this gives us the following results

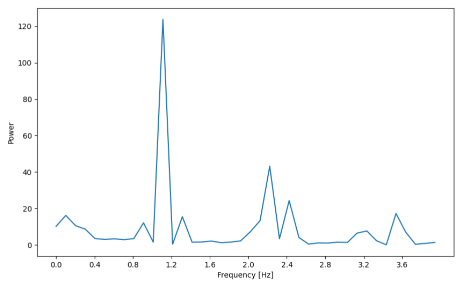

Figure 1.19: First analysis PSD window 1

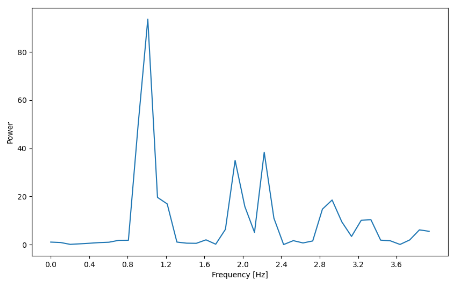

Figure 1.20: First analysis PSD window 2

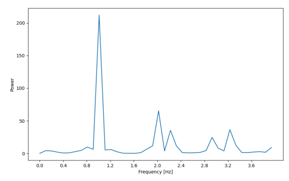

Figure 1.21: First analysis PSD window 3

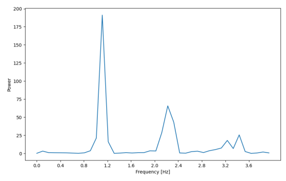

Figure 1.22: First analysis PSD window 4

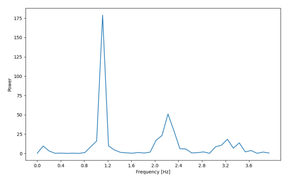

Figure 1.23: First analysis PSD window 5

The procedure remains identical; we will now determine the lower and upper limits of each individual window.

Table 1.2: High and lowcut per window

window	lowcut	highcut
1	0.9	1.3
2	0.8	1.2
3	0.8	1.2
4	0.8	1.2
5	0.8	1.2

This results in the following analysis.

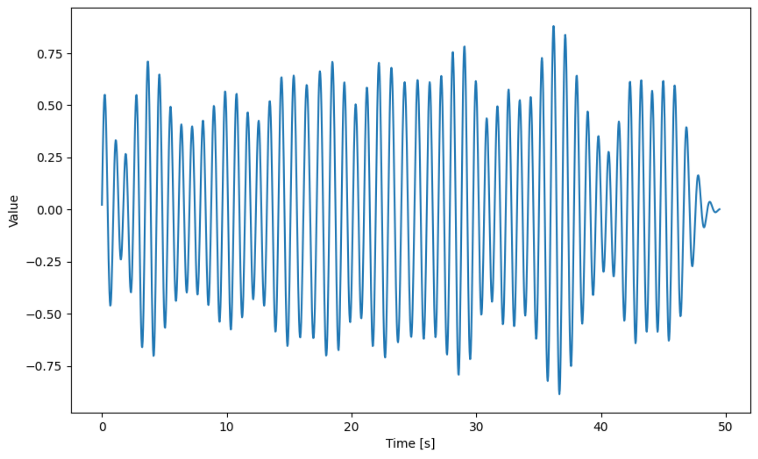

Figure 1.24: First analysis filtered

As illustrated in the figure above (fig. 1.24), the data has been successfully extracted and a clean function with peaks has been obtained. The FFT and PSD have been applied to the filtered data, resulting in the following outcomes:

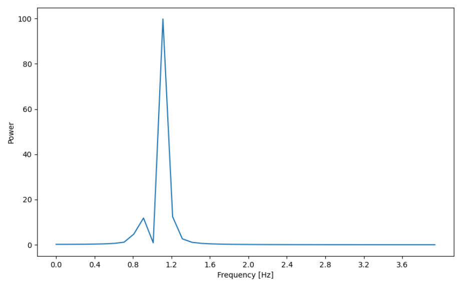
Figure 1.25: Second analysis PSD window 1

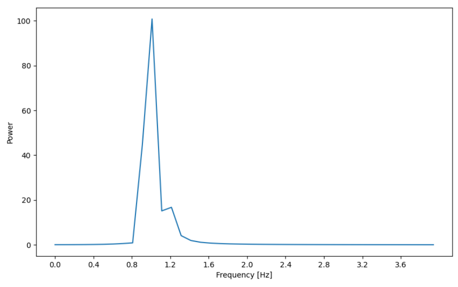
Figure 1.26: First analysis PSD window 2

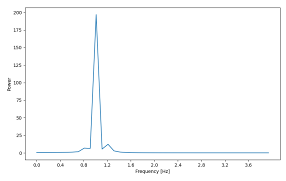
Figure 1.27: First analysis PSD window 3

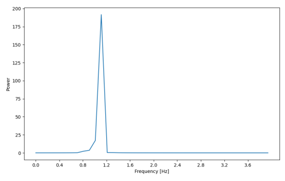
Figure 1.28: First analysis PSD window 4

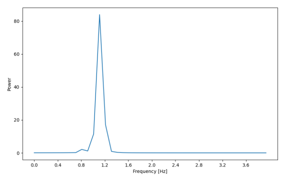
Figure 1.29: First analysis PSD window 5

After the second analysis we get the following BPM plot.

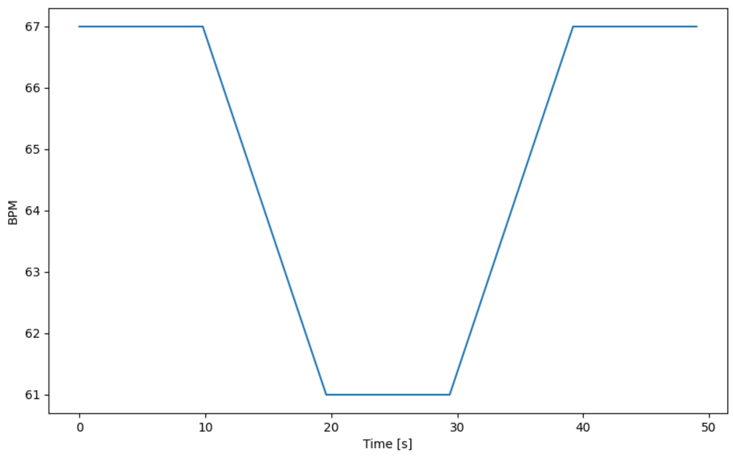
Figure 1.30: BPM over time

We see in the graph (fig. 1.30) that we hover around 65 BPM which is indeed a BPM in rest. Their is a slight drop in BPM after 20 seconds and after 10 seconds it jumps back up to 67 BPM.

1.7.2 Active

We will do all the previous processes for captured hearth rates 30 seconds, 1 minute, 2 minutes and 5 minutes after exertion. To keep the length of the report to a minimal, only the second analysis will be shown.

1.7.2.1 30 seconds

30 seconds after the exercise we get the following second analyzed data.

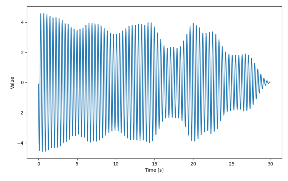
Figure 1.31: Second analysis filtered

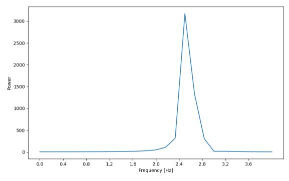
Figure 1.32: Second analysis PSD window 1

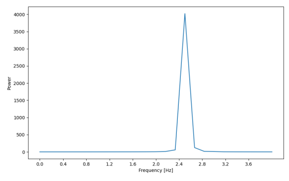
Figure 1.33: Second analysis PSD window 2

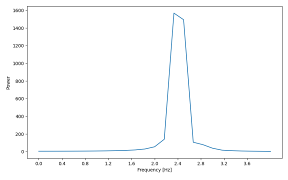
Figure 1.34: Second analysis PSD window 3

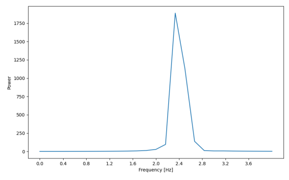
Figure 1.35: Second analysis PSD window 4

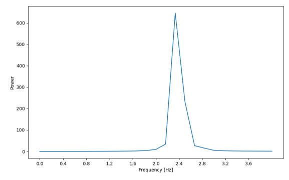
Figure 1.36: Second analysis PSD window 5

Resulting in the following BPM graph:

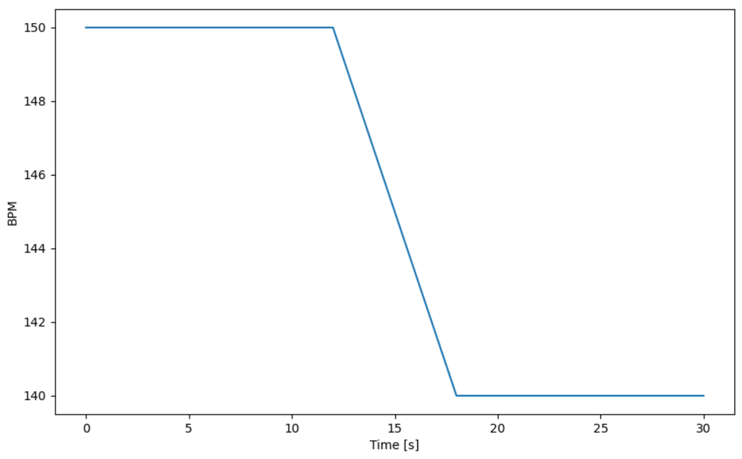

Figure 1.37: BPM over time

We start at 150 BPM, after 15 seconds our hearth rate drops to 140 BPM.

1.7.2.2 1 minute

1 minute after the exercise we get the following second analyzed data.

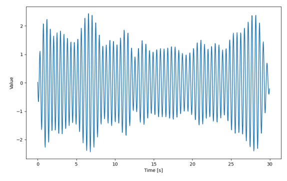

Figure 1.38: Second analysis filtered

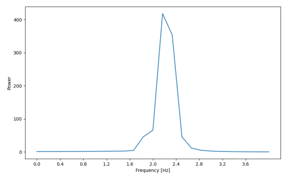

Figure 1.39: Second analysis PSD window 1

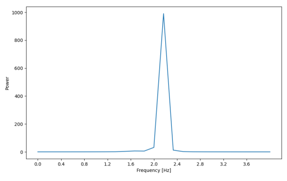

Figure 1.40: Second analysis PSD window 2

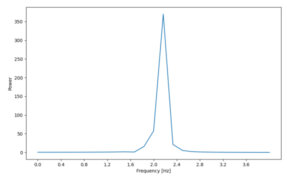

Figure 1.41: Second analysis PSD window 3

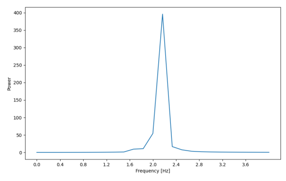

Figure 1.42: Second analysis PSD window 4

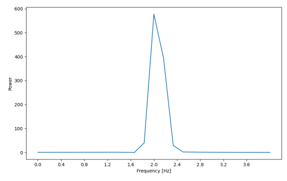

Figure 1.43: Second analysis PSD window 5

Resulting in the following BPM graph:

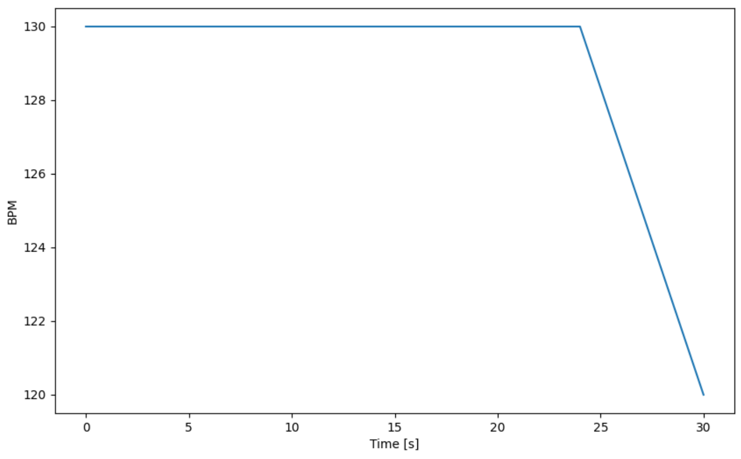

Figure 1.44: BPM over time

We see that the BPM has dropped to 130 BPM after a minute. In this second window in the last 5 seconds we see a drop to 120 BPM.

1.7.2.3 2 minutes

2 minute after the exercise we get the following second analyzed data.

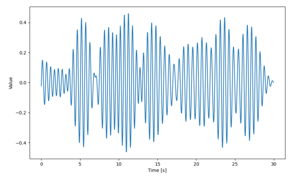
Figure 1.45: Second analysis filtered

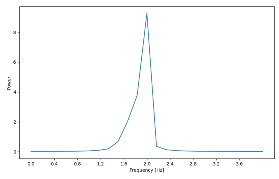
Figure 1.46: Second analysis PSD window 1

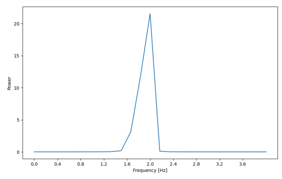
Figure 1.47: Second analysis PSD window 2

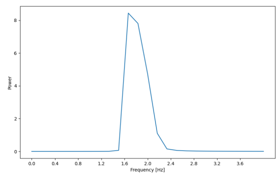
Figure 1.48: Second analysis PSD window 3

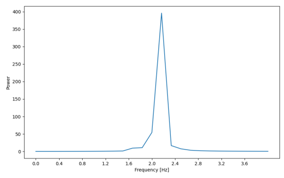
Figure 1.49: Second analysis PSD window 4

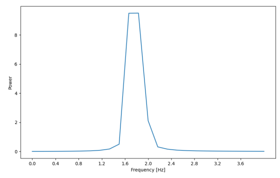
Figure 1.50: Second analysis PSD window 5

Resulting in the following BPM graph:

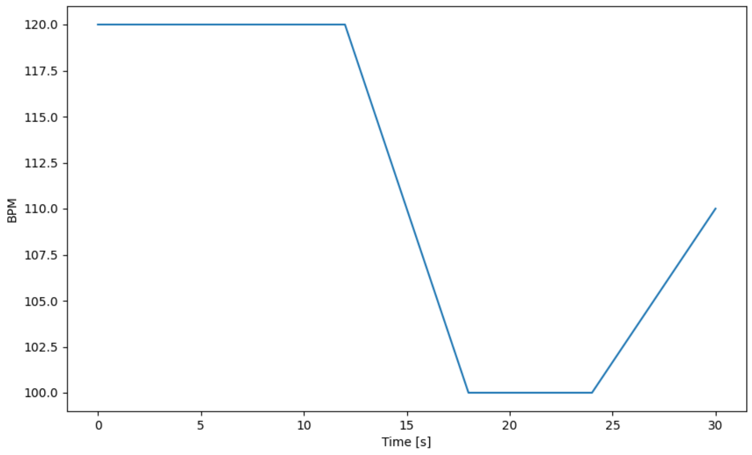

Figure 1.51: BPM over time

We can see that the bpm continuous to be 120 BPM until 15 seconds in the window where it drops to 100 BPM and then increases back to 110 BPM.

1.7.2.4 5 minutes

Lastly 5 minute after the exercise we get the following second analyzed data.

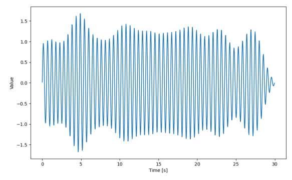

Figure 1.52: Second analysis filtered

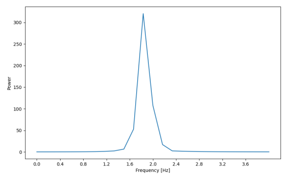

Figure 1.53: Second analysis PSD window 1

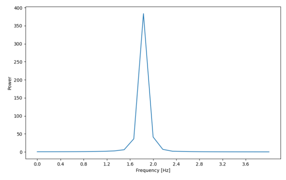

Figure 1.54: Second analysis PSD window 2

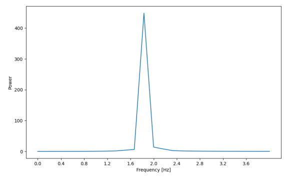

Figure 1.55: Second analysis PSD window 3

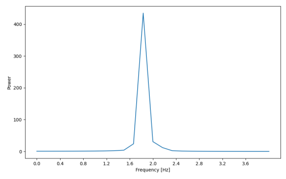

Figure 1.56: Second analysis PSD window 4

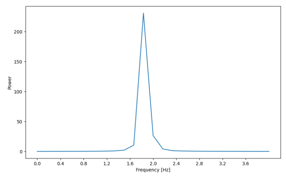

Figure 1.57: Second analysis PSD window 5

Resulting in the following BPM graph:

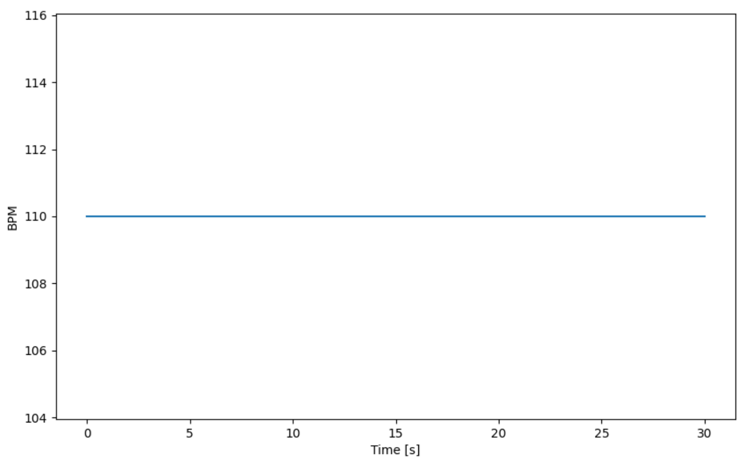

Figure 1.58: BPM over time

As for the last window we can see that for the entire length of the video we have a stable 110 BPM hearth rate.

Analyzing all the graphs together we can see that the BPM drops quite rapidly after the exercise. After a minute or so it evens out and will drop more gradually.

1.8 Program

1.8.1 Introduction

For this task, I have developed a graphical user interface (GUI) program that facilitates the analysis of photoplethysmography (PPG) signals. This program allows users to easily visualize and analyze the graphs, as well as to specify lower and upper frequency cutoffs for signal processing within the same interface.

The program is open-source, ensuring that it can be freely downloaded, used, and modified by anyone interested in PPG signal analysis. The source code and further details are available on GitHub at the following link: https://github.com/samuelg808/photoplethysmography_root.

It should be noted that the graphical user interface (GUI) is not yet compatible with different resolution screens.

The Tkinter library was employed to create the graphical user interface (GUI), while the sys library was utilized to ensure that both the GUI and the processing script cease operation upon the user's exit from the GUI.

1.8.2 Features

- **Calculate the average BPM of the whole video:** A single window can be used to calculate the average BPM of the entire video (30fps or 60fps).
- **Calculate BPM over 5 windows:** Using five windows, the BPM can be calculated and plotted over time. It is recommended to use a video of +30 seconds to ensure enough data is collected per window.
- **Determine percentile ranking:** Percentile ranking can be determined from a data set (not in GUI).

1.8.3 Installation

To run this project, you will need to have the following Python libraries installed:

- matplotlib
- numpy
- pandas
- tkinter
- sys

You can install these libraries using pip, Python's package manager. Open your terminal or command prompt and issue the following commands:

```
pip install matplotlib numpy pandas
```

1.8.4 Usage

1.8.4.1 Calculate the average BPM over the entire video

1. Enter the full path of the .mp4 file in the GUI.

2. Select the number of windows, in this case 1.
3. Look at the Power Spectral Density graph and determine its Lowcut and Highcut, enter them in the input box and click Next.
4. The graphs now plotted are those of the second iteration, click OK.
5. The average BPM over the whole video is given.

1.8.4.2 Calculate the BPM over 5 windows

1. Enter the full .mp4 file path in the GUI.
2. Select the number of windows, in this case 5.
3. Look at all the Power Spectral Density graphs and determine their lowcuts and highcuts, fill in all the input boxes and click Next.
4. The graphs now plotted are the second iteration graphs, click on OK.
5. The BPM over 5 windows is now plotted over the duration of the entire video.