Heartbeat PPG | ED18B027 Report

Created	@Mar 19, 2021 8:09 PM
Status	
Tag	

Task 1: A Survey of PPG Applications

The apps selected for this survey are listed below:

- **Heart Rate Monitor Pulse checker: BPM tracker** by Fitness Lab
- **Cardiograph - Heart Rate Meter** by MacroPinch
- **Instant Heart Rate: HR Monitor & Pulse Checker** by Azumio Inc.
- **Heart Rate Monitor** by Meet Your Need Production

Why do you find them interesting and is there anything unique?

Heart Rate Monitor Pulse checker:

- Easy to use and fingerprint indicator tells if your finger is steady.

Instant Heart Rate:

- Can be linked to gmail to save data.

Heart Rate Monitor:

- Straightforward to use and informative.

Is it an outcome of an academic research lab. Are there any research papers or articles linked to it. Is it a startup doing it?

- I couldn't find any apps that was an outcome of academic research or any which had a research paper attached to it.
- The Instant Heart Rate App though is supposedly trusted by Stanfor'd leading cardiologists for use in clinical trials.

State a couple of things the app does and are missing in others.

- Some apps like the Heart Rate Monitor Pulse checker and the cardiograph miss the reat time graph display feature witch helps the user stabilize their finger and know is they are not making unwanted movements.
- The Instant Heart Rate app turns off the flash if left idle for some time to save battery and reduce heating which the others dont.

State a couple of glitches that you feel the app has.

- The Heart Rate Monitor Pulse Checker vibrates each time it detects a peak which may cause noise to creep into the data.
- Almost all app perform poorly or don't even work when in a dark environment.
- The Cardigraph app gives the most inaccurate data when compared to all the other apps even in decent lighting conditions.
- Though it is not the most eye pleasing of all the apps I found the **Heart Rate Monitor** by Meet Your Need Production to be the most accurate of all the app when tested with a Pulseoximeter.

Task 2: Generating test video/datasets

I have used my smartphone's camera with the flash on to record sample datasets.

- The maximum FPS that my phone can support is **30 fps**.
- The maximum Resolution that my phone can support is **1080p.**

I have taken 3 types of videos at a given time.

- **Normal**: Decent lighting condition
- **Bright**: Held directly in front of light source
- **Dark**: Recorded in dark room

Naming format for the videos:

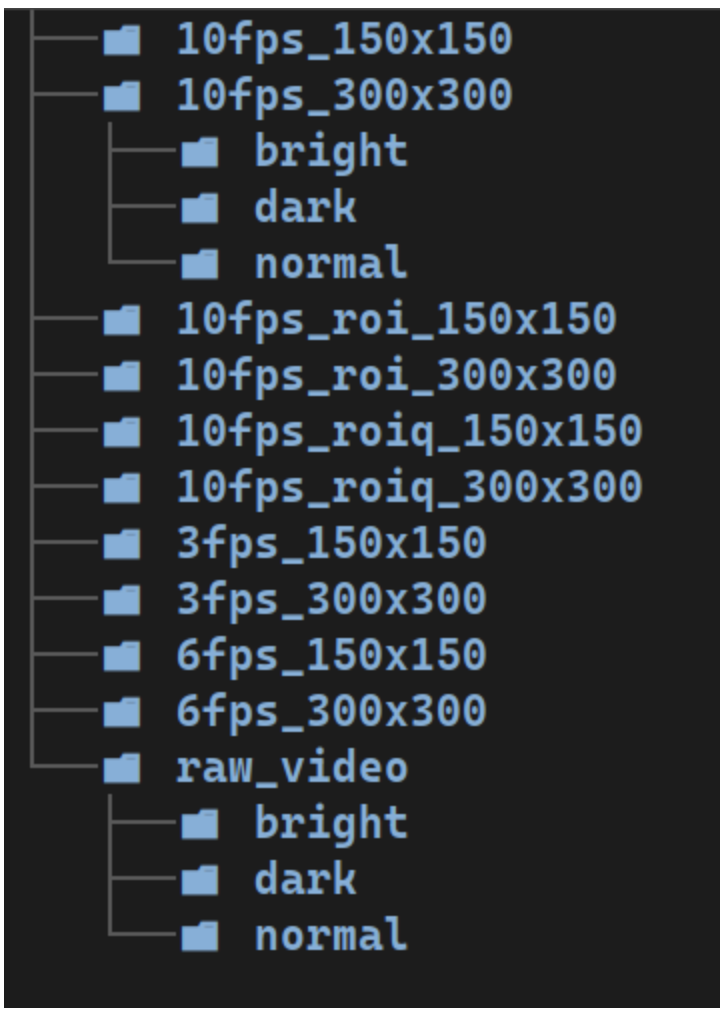
s.no_(activity/time)_(normal/bright/dark)_(pulse-oximeter-reading)

I have tried taking recordings at different times of the day and also linked to it the pulseoximeter reading taken.

> *Note that in some cases the pulseoximeter reading may not truely represent the actual value due to errors in the pulseoximeter as well as the difference in time between readings.*

*Eg: In the **9th** video it says **83** bpm but on manual calculation using video tells me it is actuall **76**.*

The processed and raw video file structure looks as below:



I have used OpenCV to process the video. The code for which can be found in **post_processing.py**

The following post processing techniques are applied:

- **Down Sampling**: Picking 1 in every n frames
- **Grayscaling**: Converting to grayscale from RGB
- **Resizing/Quantization**: Reduces resolution using INTER-AREA interpolation technique which reduces losses asmuch as possible.
- **Dilate**: This dilates or increases the intensity of the brighter pixels.
- **Gaussian Blur**: To smoothen out the signal.

The name of the files tell you what the fps and resolution of the processed videos.

I have started of with **10fps** and **300×300** resolution.

Task 3: Sensing Algorithm

Time Domain Analysis

This is implemented in the file **time_domain.py** using **numpy, scipy, matplotlib** and **openCV.**

The time series data is formed taking the average of pixel values in each frame as the average intensity. How this quantity varies with time is exploited to find the BPM.

A 5 second window sliding by 5 seconds each time is used to find the peaks and calculate the bpm.

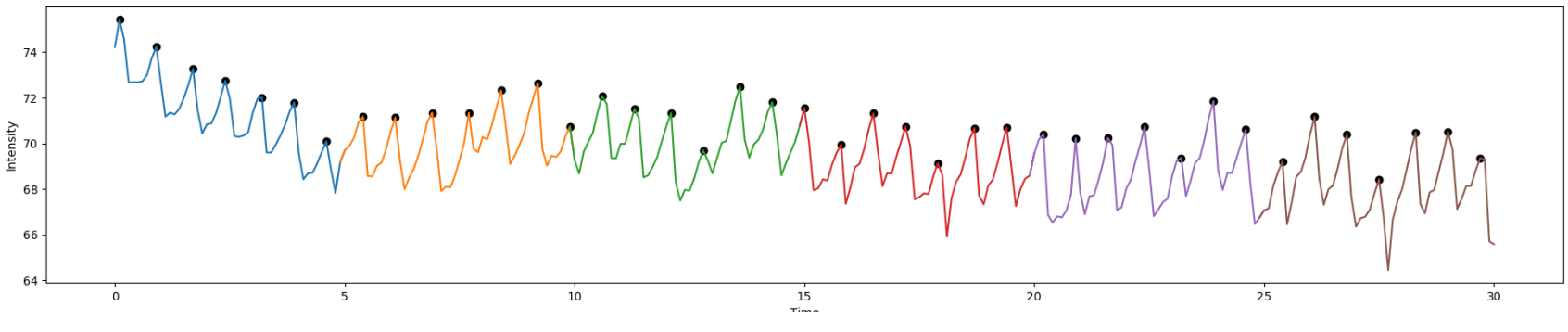
The peaks are found using the aid of the **scipy.signal** library which contains a **find_peaks** function which finds the peak by checking slope change from positive to negative.

I have set the distance between peaks to follow the formula **d = fps*60/max_bpm** assuming the range of sensing would be from **60 - 120 bpm**. So there should be atleast d frames separating two peaks.

No_of_peaks*12 gives the bpm of current window since it is a 5 second duration. This is averaged over 5 windows (30 seconds) to give the average bpm.

To also include the case where a peak is exactly at the edge of the window(peak wouldn't be recoggnised), one frame on the **left** and **right** were included in the window. Except the left of the first window which doesn't exist.

Initially I thought of having a 6 second window sliding by 3 seconds each time but this gave less correct averages as less weightage is given to first and last 3 seconds. Sliding 1 frame at a time would solve this but this increases computation significantly and the no overlap sliding would give the same answer, hence went with that.



Frequency Domain Analysis

This is implemented in the file **freq_domain.py** using **numpy, matplotlib** and **openCV.**

The same way as above the time series data is obtained by averaging the pixels of each frame.

This time series data is used to perform a **fft**(Fast Fourier Transform) using numpy's implimentation of the same. Since fft gives both negative and positive frequencies **[-f/2, f/2]** and complex values, its magnitude is found and frequency sliced appropriately.

Initially, I thought of plotting the fft for every 5 seconds and taking the average of the max component. But I noticed that the resolution is very low for a fps of 10 which gives 50 samples to work with.

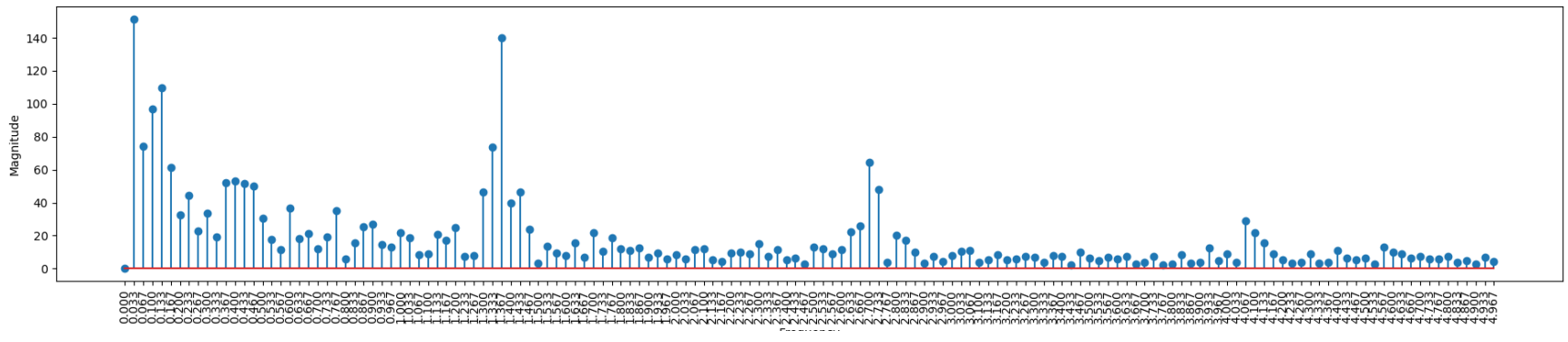
Hence the fft of the whole series upto the multiple of 5 seconds was taken which increases the **resolution**. Maximum resolution(minimum separation) is obtained when time = 30seconds (300 frames)

The **mean** of the series is removed to remove the high magnitude at f=0 and frequencies only from 1Hz to 2Hz were considered as I assumed BPM between 60 to 120. It may not be feasible to take frequencies outside this range as this could have also been caused by sensor noise, etc. Eg dark lighting conditions hence better medical sensor should be considered.

Another observation was that frequencies right next to the max frequency had significant amplitude sometimes. This maybe due to resolution not being enough and hence got separated into 2 discrete frequencies. To tackle

this a **weighted average** of the previous and next of the max frequency was taken.

BPM is given by **BPS*60**



Task 4: Evaluating the Performance of the Algorithm

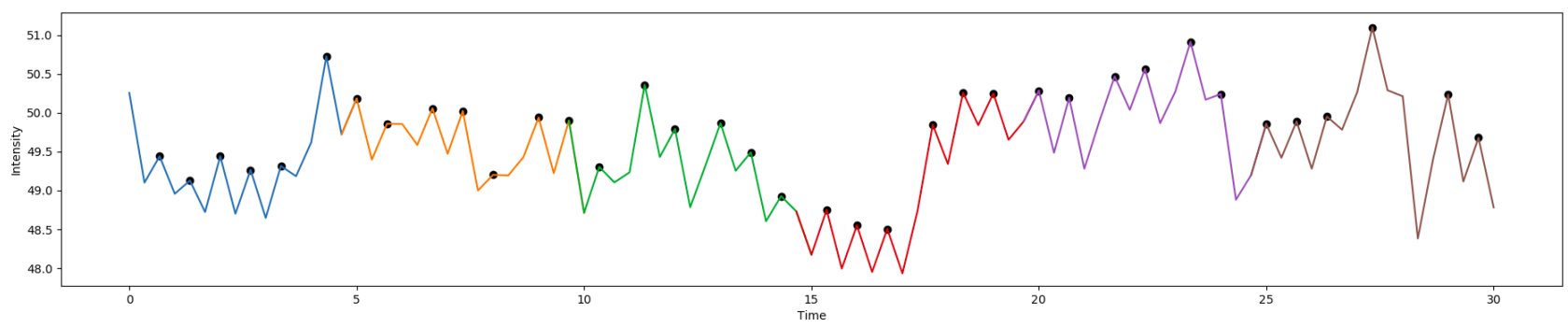
To stress test the algorithm many fps and resolution conditions were considered. The output raw txt output is given in **results.txt.**

Different cases considered:

- 10fps - 300×300
- 6fps - 300×300
- 3fps - 300×300
- 10fps - 150×150
- 6fps - 150×150
- 3fps - 150×150
- 10fps - 300×300 - RoI (Region of Interest - 600×600 center)
- 10fps - 150×150 - RoI (Region of Interest - 600×600 center)
- 10fps - 300×300 - RoIq (Region of Interest - 540×960 quater)
- 10fps - 150×150 - RoIq (Region of Interest - 540×960 quater)

Effect of FPS

Decreasing fps increase errors. It works pretty good for 10fps and 6fps in the time domian but performs badly for 3fps. Some aliasing effects are also visible in the timeseries plot. Like in the below plot you see a different frequency on the regular frequency.



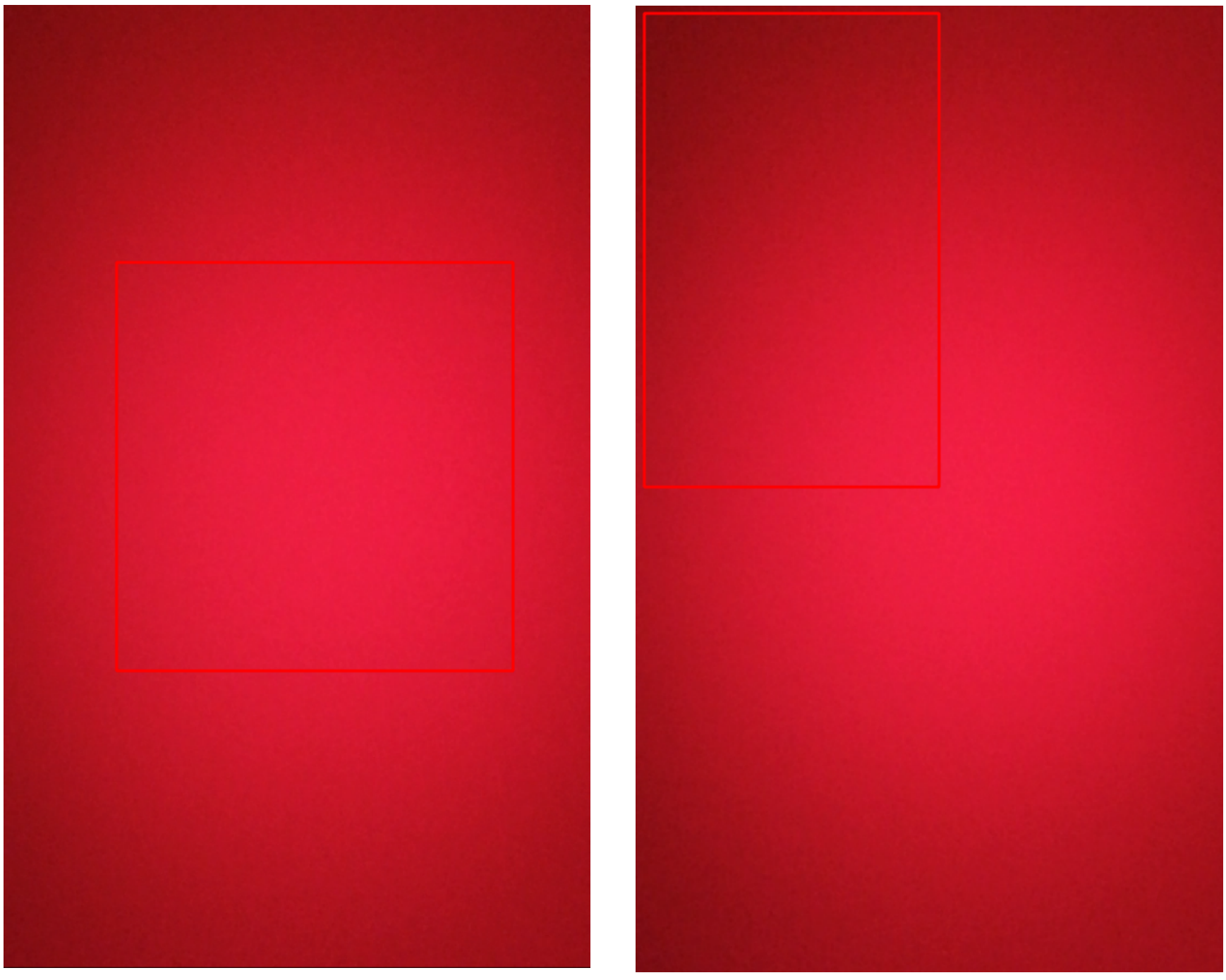
The fft method though **works surprisingly well** at 3fps but only works for bpm's till 1.5*60 = **90bpm.**

We can only go down till **4fps** for proper functioning according to the Nyquist Rule, which is twice what we want to capture (2Hz)

Effect of Resolution

Decreasing resolution has more effect on errors. This is because reducing a side by a factor has the effect of reducing the area by the square of the factor. Reducing to 150×150 increased the error from +-2 to +-3 on average.

To increase the SNR I tried cropping off everything apart from a 600×600 frame(**RoI**) which I then downsampled but this gave irregular values which might have shown up because intensity may not increase after a certain value and there is no dark pixel to increase its value. Hence averge may be same over many frames. Both time and frequency domain failed here.



Then I went with using a quater RoI, since each frame is basically a rough mirror image about the x and y axis passing through the center. This reduces the computation by **4** while giving the same output and it also has darker pixels which can be covered by the lighter ones when it expands.

FPS vs Resolution

Low FPS lower frames to compute, less effect on errors and no effect on frequency resolution of fft.

Reducing FPS has no effect on resolution of frequency since **df = fs/N** where **fs = fps** and **N = fps*time**. It only depends on the number of samples taken and this increases the longer you wait.

Low Resolution has significant increase in quantization errors but much less computations.

Reducing FPS works better than reducing Resolution in terms of less error but the reduction in computation power is much more when Resolution is reduced.

If fps is 3 and resolution 150×150 my laptop could compute 30 sec worth of data in about 5 sec. Hence there is a lot of scope to sacrifice more computational power for reduction in quantization errors.

Resources

- https://stackoverflow.com/questions/4098131/how-to-update-a-plot-in-matplotlib
- https://dsp.stackexchange.com/questions/16181/matlab-remove-the-frequency-at-zero-in-fft
- https://www.gaussianwaves.com/2015/11/interpreting-fft-results-obtaining-magnitude-and-phase-information/
- https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html#numpy.fft.fft
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks.html