Python Heart Rate Analysis Toolkit Documentation

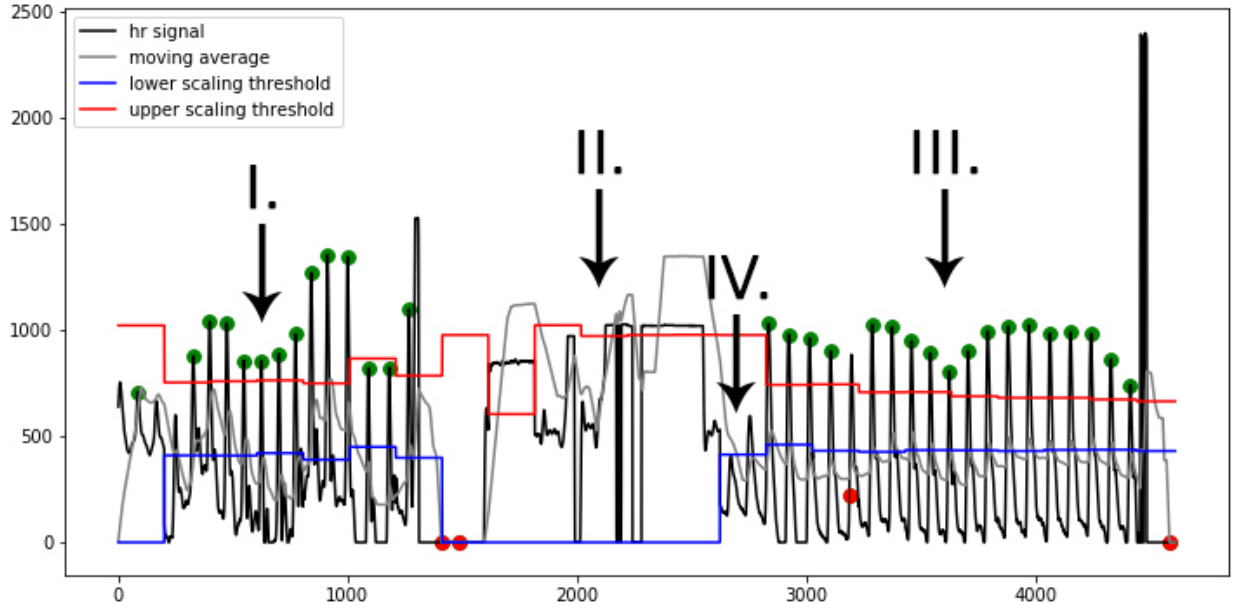
Release 0.8

Paul van Gent

Apr 02, 2019

.

Welcome to the Arduino Heart Rate Analysis Toolkit's documentation. This documentation describes the embedded implementations available for heart rate analysis.



The toolkit was presented at the Humanist 2018 conference in The Hague (see paper here). A technical paper about the functionality is currently under review and will be linked here as soon as it's published.

Please cite one or both of these papers when using the toolkit in your research. Citation format below

van Gent, P., Farah, H., van Nes, N., & van Arem, B. (2018). Heart Rate Analysis for Human Factors: Development and Validation of an Open Source Toolkit for Noisy Naturalistic Heart Rate Data. In Proceedings of the 6th HUMANIST Conference (pp. 173–178).

The documentation will help you get up to speed quickly. Follow the *Quickstart Guide* guide for a general overview of how to use the toolkit in only a few lines of code. For a more in-depth review of the module's functionality you can refer to the papers mentioned above, or the *Background - Heart Rate Analysis* overview.

CHAPTER 1

Note on using it in scientific research

Support is available at P.vanGent@tudelft.nl. When using the toolkit in your scientific work: please include me in the process. I can help you implement the toolkit, and the collaboration will also help improve the toolkit so that it can handle more types of data in the future.

CHAPTER 2

Index

2.1 Quickstart Guide

If you find yourself here, chances are you want to use the developed toolkit in your research or some other open-source application. Great! On this page we will describe the options available to you. If you already know what you want please go to the *Implementations* section.

Otherwise, look at the statements below and click whichever one is closest to your situation:

2.1.1 Where to begin?

- *I have recorded heart rate data and want to analyse it*
- *I have a way of recording heart rate data, and just want to analyse the recorded data*
- *I just want to record heart rate data unintrusively, I have my own analysis tools*
- *I want to record heart rate data unintrusively, it's ok if the analysis is done later (offline)*
- *I want to record heart rate data and analyse the results real-time*

I have recorded heart rate data and want to analyse it

You need the Python implementation of this toolkit. Take a look at the repository here: https://github.com/paulvangentcom/heartrate_analysis_python

And at the documentation here: https://python-heart-rate-analysis-toolkit.readthedocs.io/en/latest/

I have a way of recording heart rate data, and just want to analyse the recorded data

Our Python version of the toolkit handles pre-recorded data (and is most complete in functionality). However, often open source alternatives might be a better alternative to record your heart rate data. They are completely transparent, adjustable to your needs and free of charge.

For the Python version if you just want to analyse, take a look at the repository here: https://github.com/paulvangentcom/heartrate_analysis_python

And at the documentation here: https://python-heart-rate-analysis-toolkit.readthedocs.io/en/latest/

We recommend you take a look at the *Implementations* section to familiarise yourself with what's possible. It only takes around $20,- to get started!

I just want to record heart rate data unintrusively, I have my own analysis tools

The *Simple Logger* implementation is just what you need! It reliably logs data either over USB or to an SD memory card. Feel free to contact me if you need help implementing this: P.vanGent@tudelft.nl

I want to record heart rate data unintrusively, it's ok if the analysis is done later (offline)

The *Simple Logger* implementation is just what you need! It reliably logs data either over USB or to an SD memory card.

To analyse the results, we suggest you take a look at our Python analysis toolkit: https://github.com/paulvangentcom/heartrate_analysis_python

I want to record heart rate data and analyse the results real-time

Your options here depend on what exactly you want.

Do you have on-line analysis tools on your PC and just want to stream sensor data? Look at the *Simple Logger* USB implementation.

A second option is the peak finder, which detects and returns detected peaks and RR-intervals realtime: *Peak Finder*.

The time series analysis version is based on the peak finder, and outputs time-series measurements real-time: *Time Series Analysis*

Finally, the full implementation is almost identical to the Python implementation. It is the most noise-robust and reliable. It only runs on Teensy ARM board due to the amount of RAM required for buffering and analysis: *Full Implementation*.

2.1.2 What board do I have?

"8-bit AVR, 32-bit ARM, what? I just bought this board. Can it run the toolkits you made?" I understand, it can be confusing. If you're unsure what you have please find a few methods of getting your board type in this section.

Method 1: Is it mentioned in this table?

Check the table below to see if you recognize the name of the board. You can also look up what CPU your board has and see if that is in the table.

Board type	CPU	Instruction set	RAM available	Available implementations
Arduino Gemma, Adafruit Trinket, Digispark USB	ATTiny45/85	8-bit	512 byte	*Simple Logger* USB version only!
Arduino Uno, Lilypad, Lilypad Simplesnap, Pro 16MHz, Pro mini, Ethernet, Mini, Nano, BT, Fio	Atmega 328p	8-bit	2 KiloByte	*Simple Logger* *Peak Finder* up to XX Hz
Arduino Lilypad USB, Micro, Espiora, Leonardo, Yún, Robot Teensy 2.0	Atmega 32U4	8-bit	2.5 KiloByte	*Simple Logger* *Peak Finder* up to XX Hz
Arduino Mega, Mega ADK	Atmega 2560	8-bit	8 KiloByte	*Simple Logger* *Peak Finder* up to XX Hz
Teensy LC, Arduino MKR Zero Adafruit Itsybitsy, Feather M0	ARM Cortex M0+	32-bit	32 kiloByte	*Simple Logger* *Peak Finder* *Full Implementation*
Teensy 3.1, 3.2, 3.5, 3.6 Adafruit Feather M4	ARM Cortex M4	32-bit	64 - 256 KiloByte	*Simple Logger* *Peak Finder* *Full Implementation* up to XX Hz

Method 2: Look at the processor

Look to see if you can find any information on there. Usually if it says "Atmel" it will be an 8-bit RISC and you will find the type in the table above. If it says "ARM" or "Cortex", you can also find the chip in the table above. If you find markings like this:

That likely means you have an ARM Cortex chip on the board.

If this doesn't help, you can google the number printed on the chip and see what comes up.

Method 4: Talk to me Contact me at P.vanGent@tudelft.nl and show me a picture of the board and/or its name.

2.2 Implementations

Several implementations are available, depending on the goal you want to achieve. This page describes them in detail. If you're unsure which to pick, take a look at the *Quickstart Guide*.

Each implementation may be available for different chipsets. The requirements are always mentioned with each implementation. If you don't know what you have, take a look at the section "What board do I have?" in the quickstart to get help with that.

have different options and characteristics. This section describes them as best as possible. The implementations are split into AVR (Arduino, and other 8-bit Atmel chipsets), and ARM (Teensy and other boards using ARM (Cortex) or other 32-bit chipsets).

General note: When using the SD versions, install the SDFat library from Greiman first: https://github.com/greiman/SdFat

2.2.1 Simple Logger

The simple logger implementation functions as a basic data logging device with highly precise timing. It utilizes hardware interrupt timers so that the chosen sampling rate is reliably maintained throughout the logging process. This differs from for example logging solutions using linux-boards (such as Raspberry Pi), pc-based logging systems, or indeed many Arduino versions out there, where timing is usually done through software timers and therefore often not precise *at all* . See here for more information on hardware timers.

Board type	Available?	Notes
Arduino	Yes	All except ATTiny based
Teensy	Yes	All versions
Other	Yes	Requires >700 bytes RAM SD version requires 512 bytes extra for buffering Sampling rate dependent on chip speed Tested up to 2KHz on 16MHz 328p

Settable options available in the code:

```
// -------------------- User Settable Variables --------------------
int8_t hrpin = 0; //Whatever analog pin the sensor is hooked up to
int8_t scale_data = 1; /*Uses dynamic scaling of data when set to 1, not if set to 0 \
                        sampling speed over 1500Hz not recommended when scaling data
                        on 8-bit AVR (e.g. Arduino)*/
int8_t mode = 0; /*Speed mode. \
               0 means the "sample_rate" speed will be used \
               1 means custom. Custom sampling rate is set through Serial after␣
↪connect.\
               See documentation for details.  */
int16_t sample_rate = 1500; /*should be 4Hz or more, over 2KHz on AVR not recommended.
                            When using adaptive scaling: over 1.5KHz not recommended␣
↪on AVR.
                            Higher speeds attainable on 32-bit chipsets.
                            Please see documentation for suggested limits and␣
↪theoretical limits*/
```

- **hrpin**: the pin you connected the sensor to. By default it is set to 0, meaning Analog-0 (often called A0 on the board pinout).
- **scale_data**: Whether to use adaptive scaling, see *Adaptive Input Amplitude Scaling*. Set to '1' to enable adaptive scaling, set to '0' to disable.
- **mode**: the logging mode, indicating if you want a predefined sampling rate, or want to set it at boot. If set to '0', whatever value is set in "sample_rate" will be adhered to.
- **sample_rate**: the sample rate to adhere to when mode is set to 0.

USB version

The **USB logger** starts when a serial connection is made to the device. It is meant to be used in connection with a computer to log heart rate. There is an example Python file supplied that shows how to do so using `PySerial`. When

set to mode 6, once a serial connection is established the logger will request a logging speed and wait for a reply, otherwise it will just start logging once a serial connection is made.

SD Version

Note that this SD version has been moved to the folder experimental on the repository, while I investigate reports of slowdowns and missed datapoints with some SD card types.

The **SD logger** starts as soon as power is applied to it. If no SD card is present or there is an error writing to the card, the default board light (pin 13) turns on and stays on. It flashes while writing data. "mode" is not available on SD version.

2.2.2 Peak Finder

The Peak Finder implementation logs heart rate data, analyses it real-time to identify peaks, and returns the peak positions + RR-intervals. It can also be set to output the raw signal as well. On 8-bit AVR implementations the sampling rate is limited by the available RAM used for buffering. It uses adaptive scaling and error correction described in *Background - Algorithm functioning*.

Board type	Available?	Notes
Arduino	Yes	All except ATTiny based
Teensy	Yes	All versions, implementation with settable sampling rate coming soon
Other	Yes	Requires >900 bytes RAM* SD version requires 512 bytes extra for buffering Sampling rate dependent on chip speed

* RAM is dependent on sample rate, as a higher sample rate will expand the size of the used buffers.

```
// -------------------- User Settable Variables --------------------
int8_t hrpin = 0; //Whatever analog pin the sensor is hooked up to
const int16_t sample_rate = 250; //up to 250Hz tested on the 328p. Not enough RAM for␣
↪more than ~320.
int8_t report_hr = 1; //if 1, reports raw heart rate and peak threshold data as well,␣
↪else set to 0 (default 0)
float max_bpm = 180; //The max BPM to be expected, used in error detection (default␣
↪180)
float min_bpm = 45; //The min BPM to be expected, used in error detection (default 45)
```

- **hrpin**: the pin you connected the sensor to. By default it is set to 0, meaning Analog-0 (often called A0 on the board pinout).
- **sample_rate**: sample rate to use for raw signal collection and peak detection. On the 8-bit AVR (Arduino) it is tested up to 300Hz. I would recommend caution when going over 250Hz, as stability over 250Hz is not

explicitly tested. There is a theoretical maximum of 325Hz based on available RAM. *On the ARM chip it is safe to go up to 1KHz.* A future update will expand the ARM abilities.

- **report_hr**: Set this to '1' to have the logger also output the raw heart rate signal and moving average.
- **max_bpm**: The maximum BPM to expect, used as a first estimation of peak position accuracy.
- **min_bpm**: The minimum BPM to expect, used as a first estimation of peak position accuracy.

USB version

The **USB logger** AVR starts when a serial connection is made to the device (The ARM version starts when power is applied regardless of serial status). It is meant to be used in connection with a computer to log peak positions and RR-intervals (and raw heart rate if set to output). There is an example Python file supplied that shows how to do so using `PySerial`. The peak finder runs at a settable sampling rate. Over 250Hz is not tested. On the 328p-chip the theoretical limit is 320Hz based on available RAM buffers, but stability above 250Hz is not guaranteed.

SD Version

Note that this SD version has been moved to the folder experimental on the repository, while I investigate reports of slowdowns and missed datapoints with some SD card types.

The **SD logger** starts as soon as power is applied to it. If no SD card is present or there is an error writing to the card, the default board light (pin 13) turns on and stays on. It flashes while writing data.

2.2.3 Time Series Analysis

This implementation is a basic heart rate analysis toolkit for both AVR and ARM chipsets. It functions like the peak detector, but will also output the described under *Time-series* every beat. For now the logger is locked at 100Hz, which makes it a lot less accurate than the ARM full implementation!

By default it will output only RR-interval of the last two peaks, and the absolute position in samples-since-start of the last detected peak.

Sample rate will be made settable in the next update.

```
// --------------------- User Settable Variables ---------------------
int8_t hrpin = 0; //Whatever analog pin the sensor is hooked up to
int8_t Verbose = 0; //Whether to report measures + description (1) or just measures␣
↪(0); See docs.
int8_t report_hr = 0; //if 1, reports raw heart rate and peak threshold data as well,␣
↪else set to 0 (default 0)
int8_t thresholding = 0; //Whether to use thresholding, can cause incorrect␣
↪rejections in conditions of high variability
float max_bpm = 180; //The max BPM to be expected, used in error detection (default␣
↪180)
float min_bpm = 45; //The min BPM to be expected, used in error detection (default 45)
```

- **hrpin**: the pin you connected the sensor to. By default it is set to 0, meaning Analog-0 (often called A0 on the board pinout).
- **Verbose**: If set to 0, variables are output in CSV format, a descriptive output is given including the variable names.
  - CSV format = "bpm,ibi,sdnn,sdsd,rmssd,pnn20,pnn50"

– Verbose looks like this:

```
1090,2679 //first is RR-value, second is peak position in samples-since-start
bpm: 66.91
ibi: 896.67
sdnn: 87.69
sdsd: 55.75
rmssd: 96.69
pnn20: 0.85
pnn50: 0.65
```

Note that the SD logger does not have the `Verbose` option.

- **report_hr**: Set this to '1' to have the logger also output the raw heart rate signal and moving average.
- **max_bpm**: The maximum BPM to expect, used as a first estimation of peak position accuracy.
- **min_bpm**: The minimum BPM to expect, used as a first estimation of peak position accuracy.

Board type	Available?	Notes
Arduino	Yes	All Except ATTiny based
Teensy	Yes	All versions
Other	Yes	Requires >1050 bytes of RAM SD version requires 512 bytes extra for buffering Sampling rate fixed @100Hz for now

USB version

The **USB logger** AVR starts when a serial connection is made to the device (The ARM version starts when power is applied regardless of serial status). It is meant to be used in connection with a computer. There is an example Python file supplied that shows how to do so using `PySerial`. The peak finder runs at a fixed 100Hz rate. The next update will introduce settable sampling rate

SD Version

Note that this SD version has been moved to the folder experimental on the repository, while I investigate reports of slowdowns and missed datapoints with some SD card types.

The **SD logger** starts as soon as power is applied to it. If no SD card is present or there is an error writing to the card, the default board light (pin 13) turns on and stays on. It flashes while writing data.

2.2.4 Full Implementation

This implementation mirrors the full Python implementation on a Teensy (ARM Cortex-based) board and makes it real-time. The logger collects 20 seconds of heart rate data, and at the end of each measurement period outputs both the time-serie and frequency-series heart rate measures.

For now the sampling rate is fixed at 100Hz. An update is being worked on that will make it settable. The Frequency Measures that are output rely on a squared FFT to estimate the periodogram, which is not a good estimator. It gives an indication, but I would **not recommend** using the frequency measures for scientific use yet. In a future version Welch's method will be implemented.

Board type	Available?	Notes
Arduino	No	Amount of RAM too limited for required buffers
Teensy	Yes	All ARM-based versions except Teensy LC, meaning 3.1, 3.2, 3.5, 3.6
Other	Yes	Requires >30 Kilobytes of RAM SD version requires 512 bytes extra for buffering Sampling rate fixed @100Hz for now

USB version

The **USB logger** starts when power is applied regardless of serial status. It is meant to be used in connection with a computer. There is an example Python file supplied that shows how to do so using `PySerial`. The analysis suite runs at a fixed 100Hz rate. A future update will introduce settable sampling rate

SD Version

The **SD logger** starts as soon as power is applied to it. If no SD card is present or there is an error writing to the card, the default board light (pin 13) turns on and stays on. It flashes while writing data.

2.3 Background - Heart Rate Analysis

A complete description of the algorithm can be found in: <ref embedded paper>.

2.3.1 Background

The Python Heart Rate Analysis Toolkit has been designed mainly with PPG signals in mind. The Raspberry Pi and the Arduino platforms have enabled more diverse data collection methods by providing affordable open hardware platforms. This is great for researchers, especially because traditional ECG may be considered to invasive or too disruptive for experiments.

2.3.2 Measuring the heart rate signal

Two often used ways of measuring the heart rate are the electrocardiogram (ECG) and the Photoplethysmogram (PPG). Many of the online available algorithms are designed for ECG measurements. Applying an ECG algorithm (like the

famous Pan-Tompkins one[1]) to PPG data does not necessarily make sense. Although both the ECG and PPG are measures for cardiac activity, they measure very different constructs to estimate it.

The ECG measures the electrical activations that lead to the contraction of the heart muscle, using electrodes attached to the body, usually at the chest. The PPG uses a small optical sensor in conjunction with a light source to measure the discoloration of the skin as blood perfuses through it after each heartbeat. This measuring of electrical activation and pressure waves respectively, leads to very different signal and noise properties, that require specialised tools to process. This toolkit specialises in PPG data.

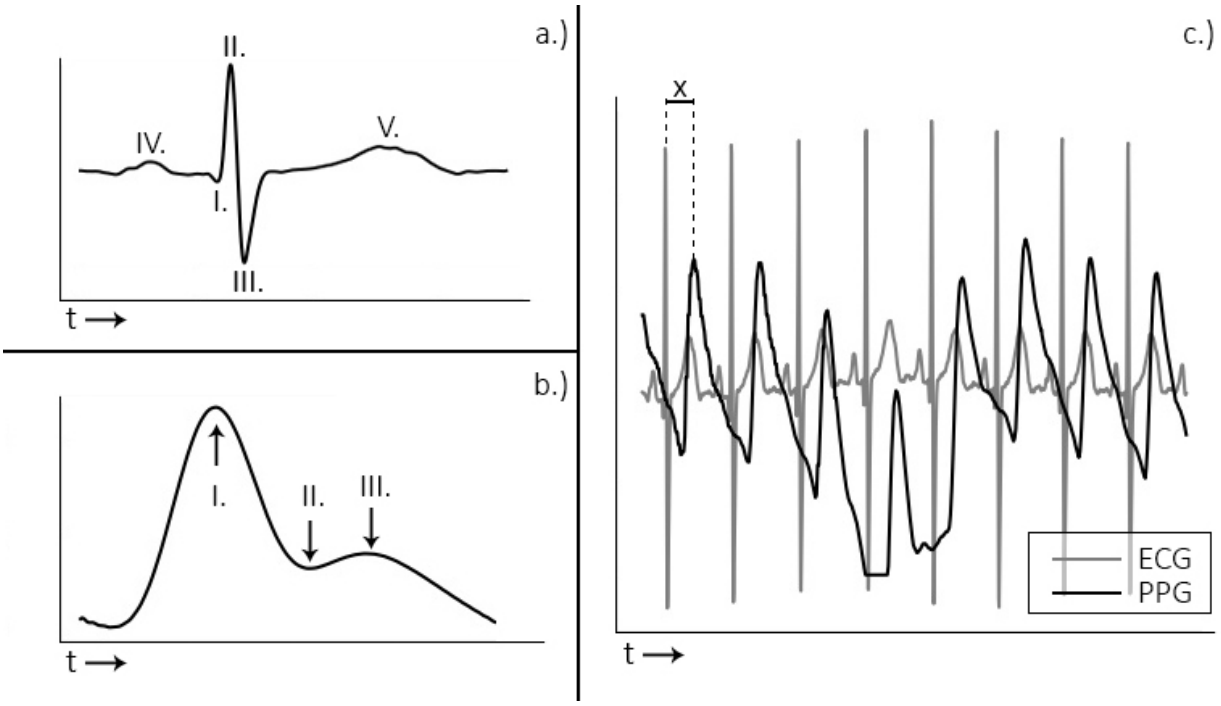


Figure 1: a. and b. display the ECG and PPG waveform morphology, respectively. The ECG is divided into distinct waves (a, I-V), of which the R-wave (a, II) is used for heart beat extraction. With the PPG wave, the systolic peak (b, I) is used. The plot in c. shows the relationship between ECG and PPG signals.

Most notably in the ECG is the QRS-complex (Fig 1a, I-III), which represents the electrical activation that leads to the ventricles contracting and expelling blood from the heart muscle. The R-peak is the point of largest amplitude in the signal. When extracting heart beats, these peaks are marked in the ECG. Advantages of the ECG are that it provides a good signal/noise ratio, and the R-peak that is of interest generally has a large amplitude compared to the surrounding data points (Fig 1c). The main disadvantage is that the measurement of the ECG is invasive. It requires the attachment of wired electrodes to the chest of the participant, which can interfere with experimental tasks such as driving.

The PPG measures the discoloration of the skin as blood perfuses through the capillaries and arteries after each heartbeat. The signal consists of the systolic peak (Fig 1-b, I), dicrotic notch (II), and the diastolic peak (III). When extracting heart beats, the systolic peaks (I) are used. PPG sensors offer a less invasive way of measuring heart rate data, which is one of their main advantages. Usually the sensors are placed at the fingertip, earlobe, or on the wrist

[1] Pan, J., & Tompkins, W. J. A simple real-time QRS detection algorithm. IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING, BME-32(3), 230–236, 1985. https://doi.org/10.1109/IEMBS.1996.647473

using a bracelet. Contactless camera-based systems have recently been demonstrated[2],[3],[4]. These offer non-intrusive ways of acquiring the PPG signal. PPG signals have the disadvantages of showing more noise, large amplitude variations, and the morphology of the peaks displays broader variation (Figure 2b, c). This complicates analysis of the signal, especially when using software designed for ECG, which the available open source tools generally are.

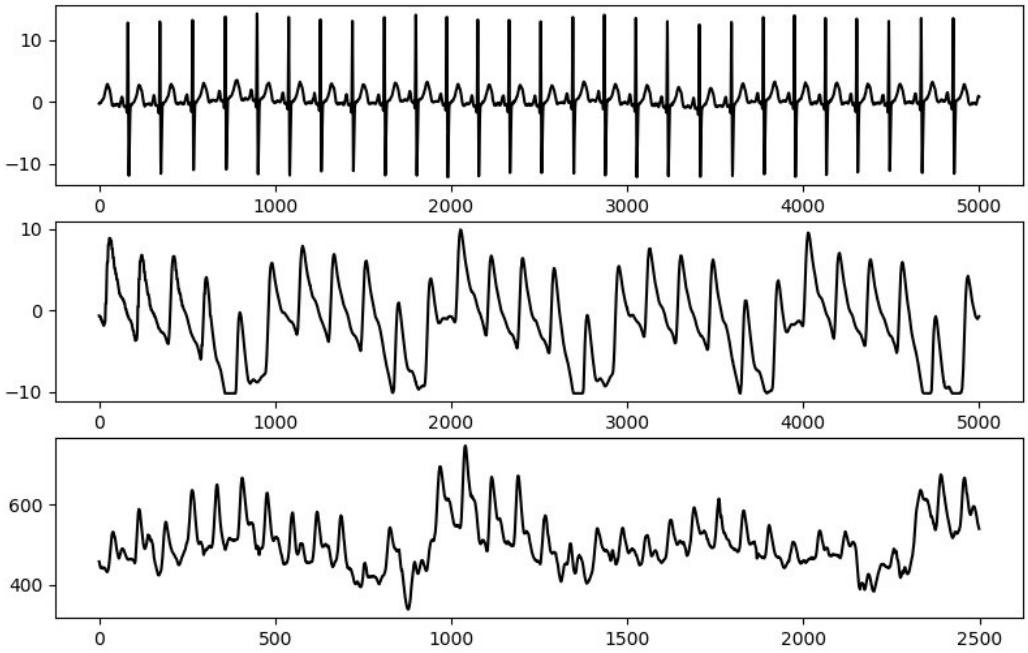


Figure 2 – The ECG signal (a.) shows a strong QRS complex together with little amplitude variation. The PPG signal measured simultaneously while the patient is at rest in a hospital bed (b.) shows some amplitude variation but relatively stable morphology. When measuring PPG in a driving simulator using low-cost sensors (c.), strong amplitude and waveform morphology variation is visible.

2

Y. Sun, S. Hu, V. Azorin-Peris, R. Kalawsky, and S. Greenwald, "Noncontact imaging photoplethysmography to effectively access pulse rate variability," J. Biomed. Opt., vol. 18, no. 6, p. 61205, 2012.

3

M. Lewandowska, J. Ruminsky, T. Kocejko, and J. Nowak, "Measuring Pulse Rate with a Webcam - a Non-contact Method for Evaluating Cardiac Activity," in Proceedings of the Federated Conference on Computer Science and Information Systems, 2011, no. January, pp. 405–410.

4

F. Bousefsaf, C. Maaoui, and a. Pruski, "Remote detection of mental workload changes using cardiac parameters assessed with a low-cost webcam," Comput. Biol. Med., vol. 53, pp. 1–10, 2014.

2.3.3 On the Accuracy of Peak Position

When analysing heart rate, the main crux lies in the accuracy of the peak position labeling being used. When extracting instantaneous heart rate (BPM), accurate peak placement is not crucial. The BPM is an aggregate measure, which is calculated as the average beat-beat interval across the entire analysed signal (segment). This makes it quite robust to outliers.

However, when extracting heart rate variability (HRV) measures, the peak positions are crucial. Take as an example two often used variability measures, the RMSSD (root mean square of successive differences) and the SDSD (standard deviation of successive differences). Given a segment of heart rate data as displayed in the figure below, the RMSSD is calculated as shown. The SDSD is the standard deviation between successive differences.

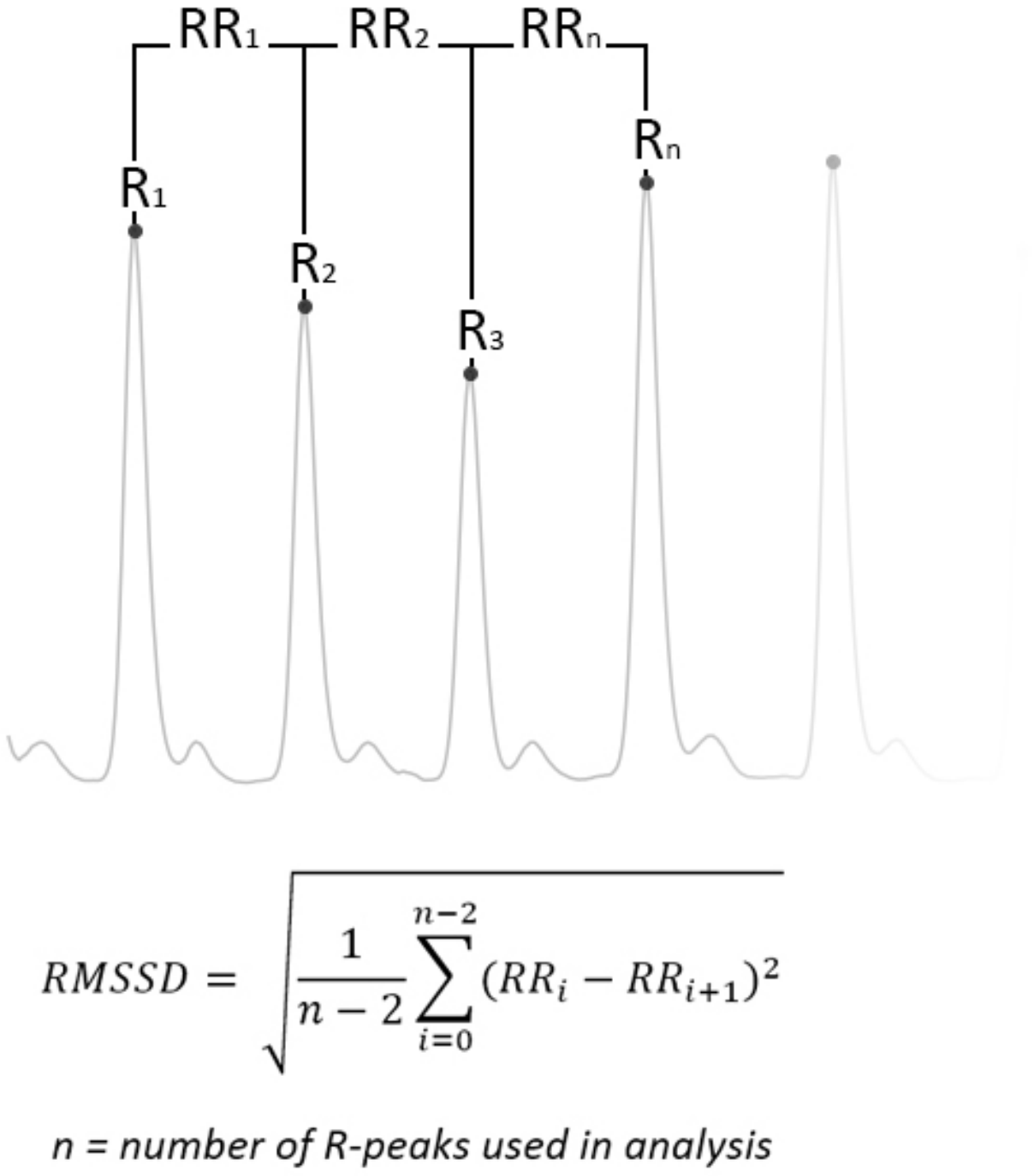


Figure 3 - Image displaying the desired peak detection result, as well as the calculation of the RMSSD measure. The SDSD measure is the standard deviation between successive differences

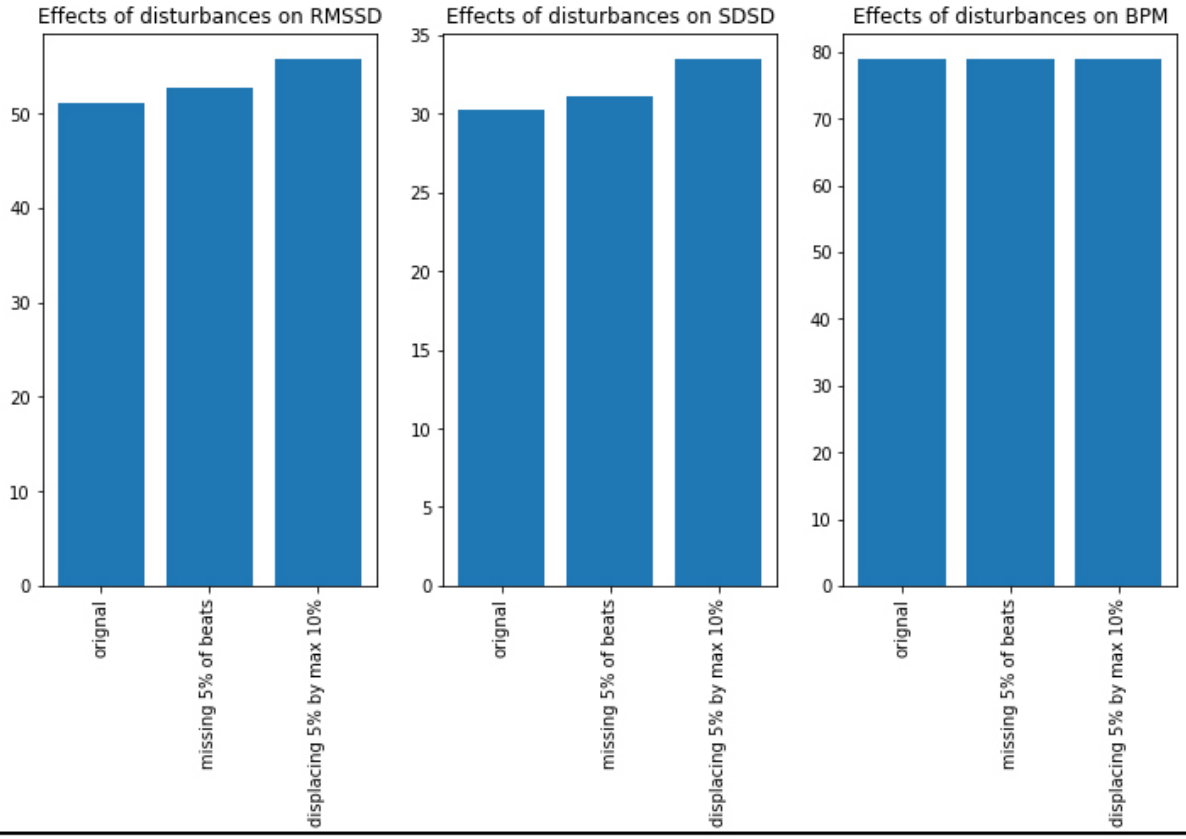
Now consider that two mistakes are possible: either a beat is not detected at all (missed), or a beat is placed at an incorrect time position (incorrectly placed). These will have an effect on the calculated HRV output measures, which are highly sensitive to outliers as they are designed to capture the slight natural variation between peak-peak intervals in the heart rate signal!

To illustrate the problem we have run a few simulations. We took a sample of a heart rate signal which was annotated manually, and introduced two types of errors:

- We randomly dropped n% of peaks from the signal, than re-ran the analysis considering only intervals between two peaks where no missing value occurred in between.
- We introduced a random position error (0.1% - 10% of peak position, meaning between about 1ms and 100ms deviation) in n% of peaks.
- The simulation ran bootstrapped for 10,000 iterations, with values n=[5, 10, 20].

Results show that the effect of incorrect beat placements **far outweigh** those of missing values. As described earlier, the instantaneous heart rate (BPM) is not sensitive to outliers, as is shown in the plots as well, where almost no discernible deviation is visible.

n=5%


Effects of disturbances on RMSSD
Effects of disturbances on SDSD
Effects of disturbances on BPM
orignal
missing 5% of beats
displacing 5% by max 10%

n=10%

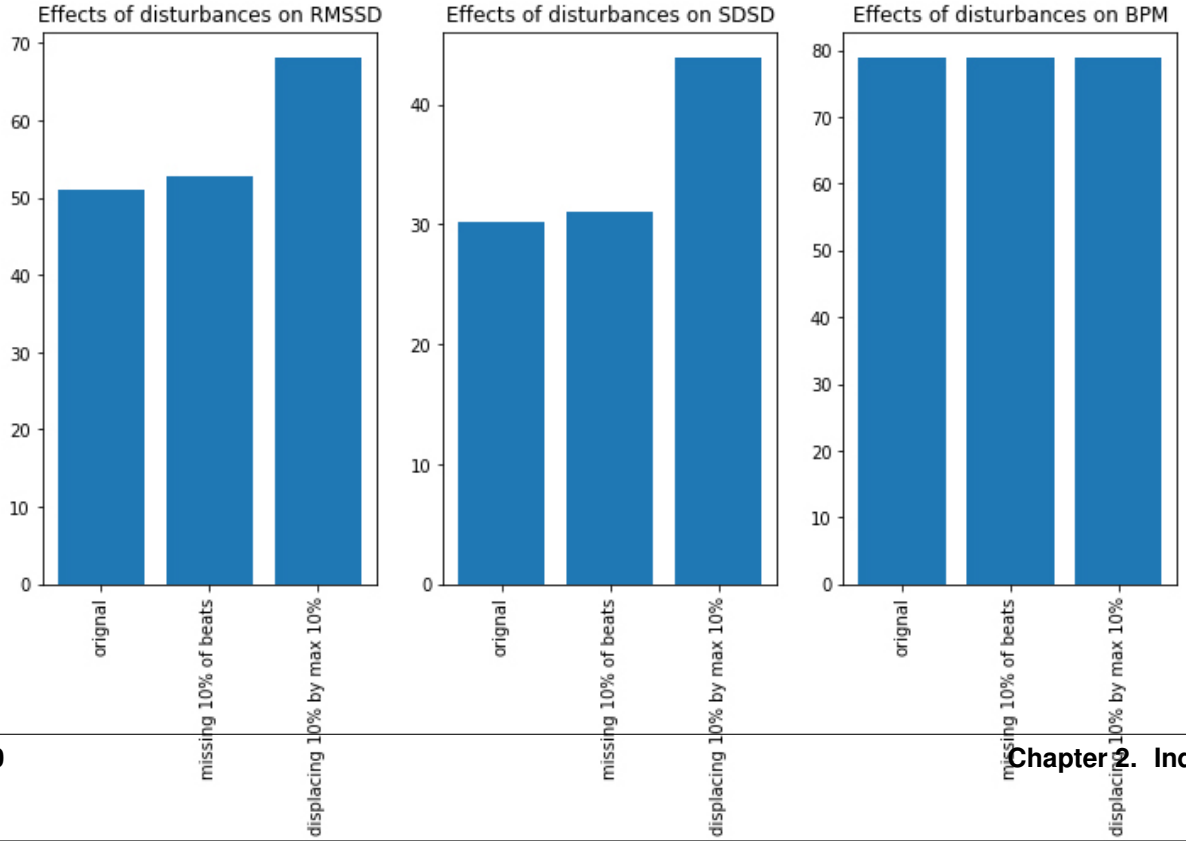

Effects of disturbances on RMSSD
Effects of disturbances on SDSD
Effects of disturbances on BPM
orignal
missing 10% of beats
displacing 10% by max 10%

Figure 4 - Results for manually anotated measures (ground truth), and error induction of n% missed beats, as well as error induction on the detected position of n% beats (random error 0.1% - 10%, or 1-100ms).

Take into consideration that the scale for RMSSD doesn't typically exceed +/- 130, SDSD doesn't differ by much. This means that even a few incorrectly detected peaks are already introducing large measurement errors into the output variables. The algorithm described here is specifically designed to handle noisy PPG data from cheap sensors. The main design criteria was to minimise the number of incorrectly placed peaks as to minimise the error introduced into the output measures.

More information on the functioning can be found in the rest of the documentation, as well as the **embedded paper**. Information on the valiation can be found in[5].

2.3.4 References

2.4 Background - Algorithm functioning

This section describes the details of the algorithm functionality.

2.4.1 Pre-processing

Depending on which implementation you select, several pre-processing steps can be applied.

Adaptive Input Amplitude Scaling

This feature is available on all implementations. It can be turned on by setting the flag "adaptivescale" in the "user settable variables" section to 0. The 8-bit AVR boards (Arduino) will be restricted in sampling rate when using adaptive scaling, since it relies on 16- and 32-bit computations, which run slower on 8-bit systems. The ARM (Teensy, 32-bit) boards can handle high sampling rates without issue.

The adaptive scaling funcions in periods of 2 seconds. At the end of each 2-second period the signal minimum and maximum are determined over the preceding period. The signal in the next period is scaled using these values. This accomodates changes in signal amplitude.

In the PPG signal, amplitude is influenced mostly by the position where the signal is measured: areas with less capillaries show less perfusion-discoloration and thus produce a weaker signal. Area's like the fingertip and earlobe show strong signals, especially in younger to middle-aged individuals with no history of heavy smoking. Furthermore, low-frequency amplitude changes occur over time due to changes in for example blood pressure, or vasoconstriction in response to (internal or external) stressors. Adaptive scaling also accomodates instances where reduced blood perfusion leads to a lower signal amplitude baseline, such as in the elderly or long-term smokers.

While not as powerful as the adaptive moving average used in the other implementations, the adaptive signal scaling is still quite powerful as shown on the figure below:

[5] van Gent, P., Farah, H., van Nes, N., & van Arem, B. (2018). "Heart Rate Analysis for Human Factors: Development and Validation of an Open Source Toolkit for Noisy Naturalistic Heart Rate Data." In proceedings of the Humanist 2018 conference, 2018, pp.173-17

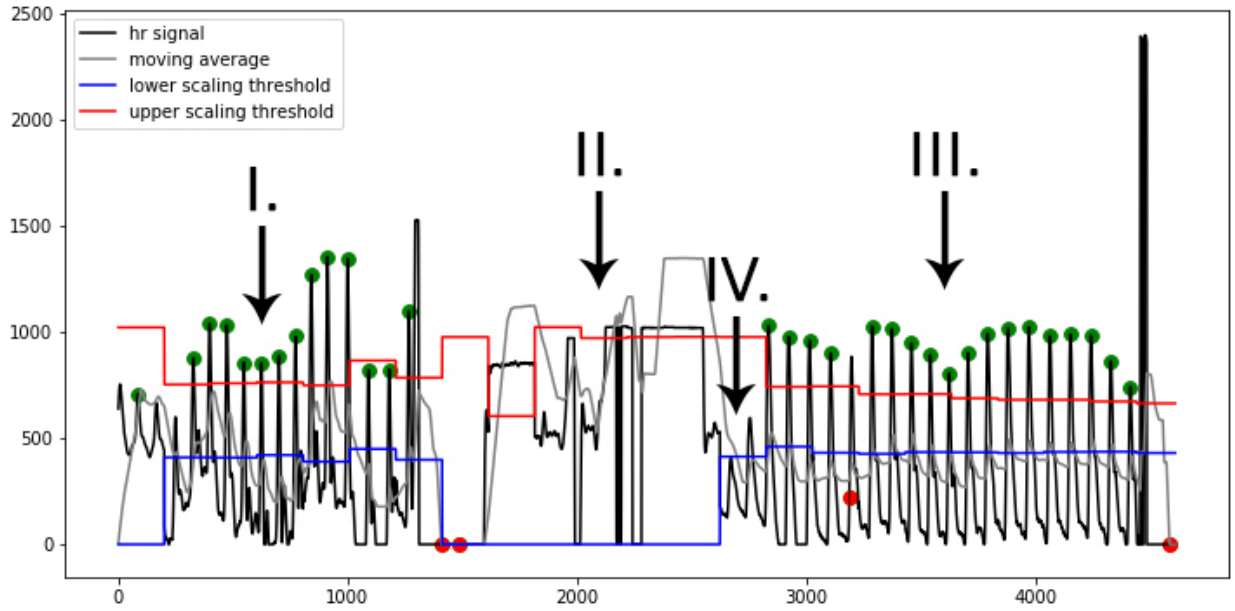
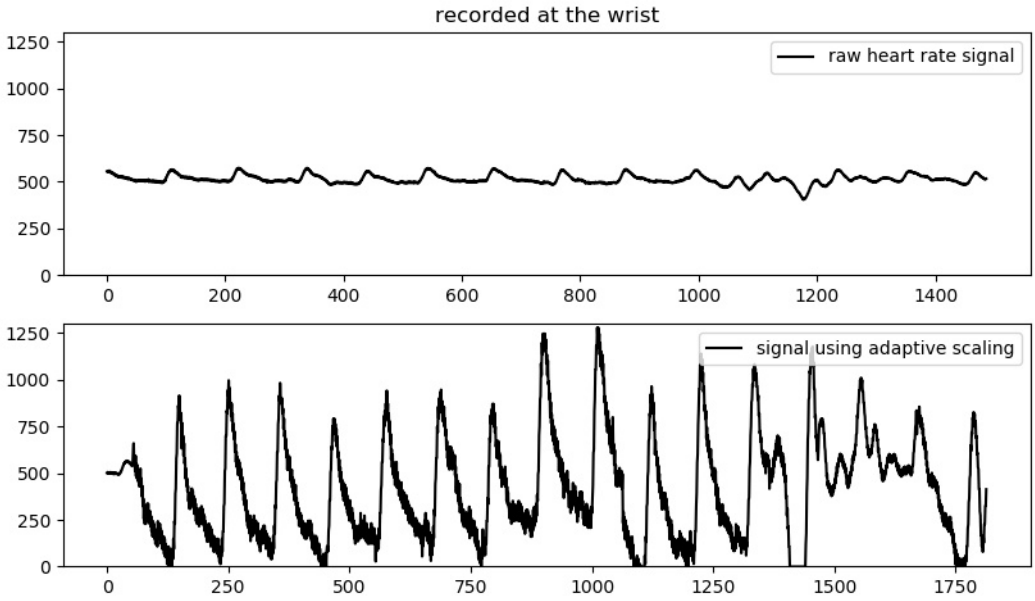


Image showing result of the real-time AVR implementation in different conditions. In the first part (I.), the signal is measured at the fingertip. A period of sensor disconnect occurs (II.) while moving the sensor to a new measurement location. In the final segment (III.) the signal is measured on the cheek, where signal amplitude is much weaker. The adaptive scaling kicks in after the first two low-amplitude beats (IV.) to stabilize amplitude and allow further analysis.

The following image shows a signal recorded at the top of the wrist, a location where typically PPG sensors record a very low amplitude signal. Two recordings are displayed below, one made without adaptive scaling (top) and one with (bottom).



As you can see it makes quite a difference. Keep in mind that it is not a magic bullet: noise will scale as well, meaning that the lower the amplitude of the original signal, generally the more noise you'll find in the scaled recording.

2.4.2 Peak detection & Error Detection

Two types of peak detection are available, one real-time using adaptive scaling, and one using an adaptive threshold similar to the Python implementation. The latter one is only available on ARM (Teensy) boards due to RAM demands made by the required buffers.

Adaptive Scaling

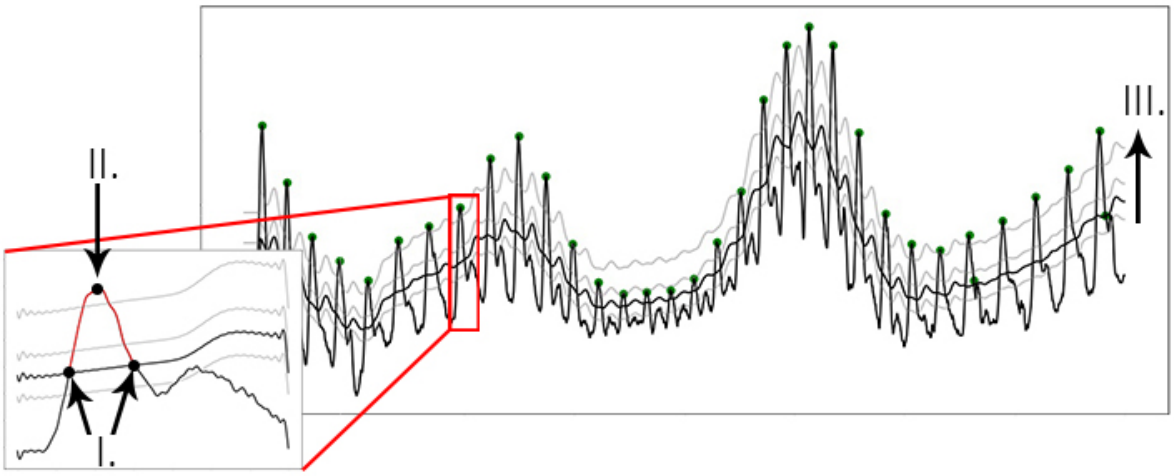
When comparing the AVR implementations to the full ARM or Python implementations, the first thing to note is that there is no adaptive moving average in the AVR version. The main reason is that AVR chips have very limited RAM, so the use of buffers required for the adaptive moving average is not feasible. In stead, the AVR implmentation implements adaptive signal scaling as discussed above.

In the case of adaptive scaling, peak detection functions with a fixed moving average that is computed on-the-fly for each datapoint read from the sensor. Whenever the signal exceeds the moving average, it is stored in a "region of interest" buffer until it dips below the moving average again. The peak is then identified in the region of interest and marked. This process is similar to the adaptive threshold used in the Python version, but in stead of moving the threshold (which requires too much RAM to buffer the signal first), it attempts to standardise the amplitude over time.

Adaptive Threshold

The adaptive threshold is similar to what is discussed in the Python version documentation. Limitations in the available RAM mean that sampling rate and size of the total segment of the analysed signal are more limited however. The adaptive thresholding implementation is only available on the Teensy 3.1, 3.2, 3.5 and 3.6. The 3.5 and 3.6 have enough RAM (128k, 256k) to allow for faster sampling speeds than the 3.1 and 3.2 (64k).

It functions as described in the Python documentation. Click here to go to the page.



Error Detection

Error-detection functions in three stages:

1. The RR-interval with the previously marked peak is computed and evaluated whether it falls into to the expected BPM range (settable under "user settable variables").

2. The RR-interval is compared to the previous RR-interval. It is not allowed to deviate more than 350ms, otherwise it is ignored.
3. Thresholds are computed based on the mean of the last 20 RR-intervals. Thresholds are determined as **RR_mean +/- (30% of RR_mean, with minimum value of 300)** (+ or - for upper and lower threshold, respectively). If the RR-interval exceeds one of the thresholds, it is ignored. This is the same as the thresholding used in the Python version

2.4.3 Calculation of Measures

All measures are computed on the detected and accepted peaks in the segment. When RR-intervals are used in computation, only the intervals created by two adjacent, accepted, peaks are used. Whenever differences in RR-intervals are required (for example in the RMSSD), only intervals between two adjacens RR-intervals, which in turn are created by three adjacent, accepted, peaks are used. This ensures that any rejected peaks do not inject measurement error in the subsequent measure calculations.

Time-series

Time series measurements are computed from detected peaks. The output measures are:

- beats per minute (BPM)
- interbeat interval (IBI)
- standard deviation of RR intervals (SDNN)
- standard deviation of successive differences (SDSD)
- root mean square of successive differences (RMSSD)
- proportion of successive differences above 20ms (pNN20)
- proportion of successive differences above 50ms (pNN50)
- median absolute deviation of RR intervals (MAD)

Frequency Domain

Frequency domain measures computed are:

- low-frequency, frequency spectrum between 0.05-0.15Hz (LF)
- high-frequency, frequency spectrum between 0.15-0.5Hz (HF)
- the ration high frequency / low frequency (HF/LF)

The measures are computed from the PSD (Power Spectral Density), which itself is estimated using either FFT-based, Periodogram-based, or Welch-based methods. The default is Welch's method.

2.5 Development

2.5.1 Release Notes

V0.3

- Included AVR Logger Version

- Included AVR Logger with Hampel filtering
- Included AVR Peak detector with error rejection and automatic scaling

2.5.2 Questions

contact me at P.vanGent@tudelft.nl