# Python Heart Rate Analysis Toolkit Documentation

*Release 1.2.5*

Paul van Gent

Dec 17, 2019

.

**1 Example Notebooks are available! 3**

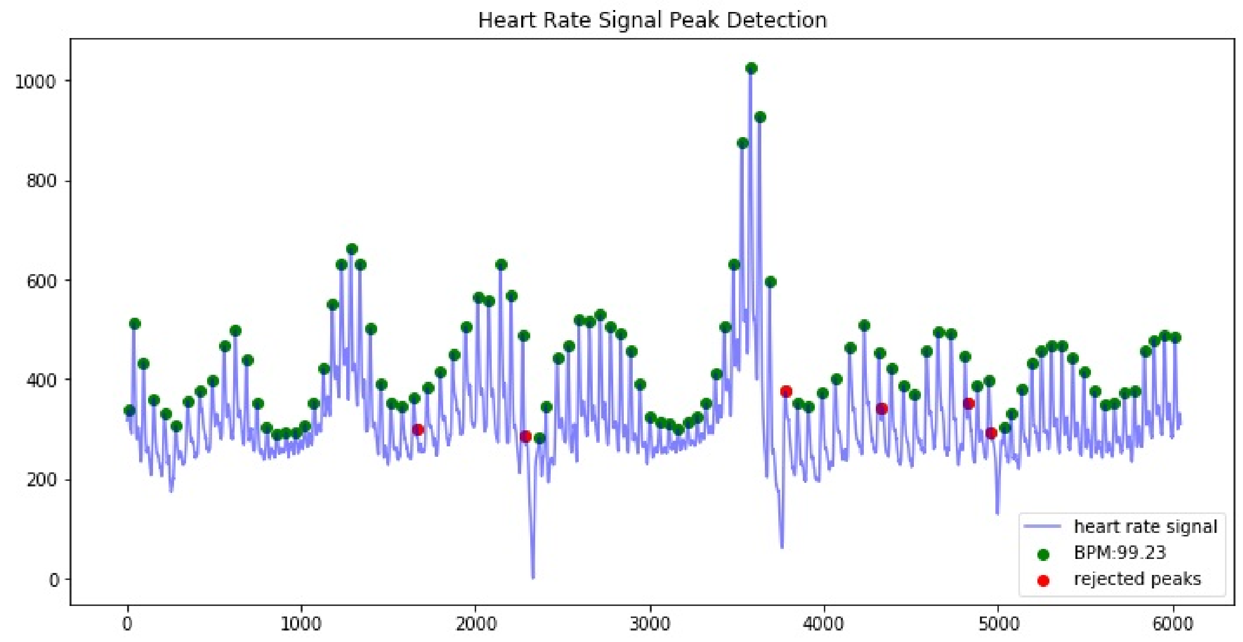
**2 Note on using it in scientific research 5**

**3 Index 7**

3.1 Quickstart Guide . . . 7
3.1.1 Installation . . . 7
3.1.2 Basic Example . . . 7
3.1.3 Getting Data From Files . . . 9
3.1.4 Estimating Sample Rate . . . 10
3.1.5 Plotting Results . . . 10
3.1.6 Getting heart rate over time . . . 13
3.1.7 Example Notebooks are available for further reading! . . . 13
3.2 API Reference . . . 14
3.2.1 heartpy (main) . . . 14
3.2.2 analysis . . . 28
3.2.3 datautils . . . 34
3.2.4 exceptions . . . 38
3.2.5 Filtering . . . 38
3.2.6 Peakdetection . . . 41
3.2.7 Preprocessing . . . 45
3.2.8 visualizeutils . . . 49
3.3 Heart Rate Analysis . . . 51
3.3.1 Background . . . 51
3.3.2 Measuring the heart rate signal . . . 52
3.3.3 On the Accuracy of Peak Position . . . 54
3.3.4 References . . . 58
3.4 Algorithm functioning . . . 58
3.4.1 Pre-processing . . . 58
3.4.2 Peak detection . . . 64
3.4.3 Peak rejection . . . 65
3.4.4 Calculation of measures . . . 66
3.4.5 References . . . 68
3.5 Development . . . 68
3.5.1 Release Notes . . . 68
3.5.2 Questions . . . 71

**Python Module Index** **73**

**Index** **75**



Welcome to the documentation of the HeartPy, Python Heart Rate Analysis Toolkit. The toolkit is designed to handle (noisy) PPG data collected with either PPG or camera sensors.

- The toolkit was presented at the Humanist 2018 conference in The Hague (see paper here ).
- A technical paper about the functionality is available here

**Please cite one or both of these papers when using the toolkit in your research!**

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 *Heart Rate Analysis* overview.

CHAPTER 1

# Example Notebooks are available!

If you're looking for a few hands-on examples on how to get started with HeartPy, have a look at the links below! These notebooks show how to handle various analysis tasks with HeartPy, from smartwatch data, smart ring data, regular PPG, and regular (and very noisy) ECG. The notebooks sometimes don't render through the github engine, so either open them locally, or use an online viewer like [nbviewer](https://nbviewer.jupyter.org/).

We recommend you follow the notebooks in order: - [1. Analysing a PPG signal](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/regular_PPG/Analysing_a_PPG_signal.ipynb), a notebook for starting out with HeartPy using built-in examples. - [2. Analysing an ECG signal](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/regular_ECG/Analysing_a_regular_ECG_signal.ipynb), a notebook for working with HeartPy and typical ECG data. - [3. Analysing smartwatch data](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/smartwatch_data/Analysing_Smartwatch_Data.ipynb), a notebook on analysing low resolution PPG data from a smartwatch. - [4. Analysing smart ring data](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/smartring_data/Analysing_Smart_Ring_Data.ipynb), a notebook on analysing smart ring PPG data. - [5. Analysing noisy ECG data](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/noisy_ECG/Analysing_Noisy_ECG.ipynb), an advanced notebook on working with very noisy ECG data, using data from the MIT-BIH noise stress test dataset.

# CHAPTER 2

# 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 3

# Index

## 3.1 Quickstart Guide

### 3.1.1 Installation

#### pip

```
python -m pip install heartpy
```

#### github

Download the latest release here

```
python setup.py install
```

### 3.1.2 Basic Example

Import the *HeartPy* module and load a file

```
import heartpy as hp

hrdata = hp.get_data('data.csv')
```

This returns a `numpy.ndarray`.

Analysis requires the sampling rate for your data. If you know this *a priori*, supply it when calling the *process()* function, which returns a *dict{}* object containing all measures:

```
import heartpy as hp

#load example data
```

(continues on next page)

(continued from previous page)

```
data, _ = hp.load_exampledata(0) #this example set is sampled at 100Hz

    working_data, measures = hp.process(data, 100.0)
```

**process(dataset, sample_rate, windowsize=0.75, report_time=False, calc_freq=False, freq_method='welch', interp_clipping=False, clipping_scale=False, interp_threshold=1020, hampel_correct=False, bpmmin=40, bpmmax=180, reject_segmentwise=False, high_precision=False, high_precision_fs=1000.0, measures = {}, working_data = {})**

requires two arguments:

- **dataset:** An 1-dimensional list, numpy array or array-like object containing the heart rate data;
- **sample_rate**: The samplerate of the signal in Hz;

Several optional arguments are available:

- **windowsize:** _optional_ *windowsize* is the window size used for the calculation of the moving average. The windowsize is defined as *windowsize * samplerate*. Default windowsize=0.75.
- **report_time:** _optional_ whether to report total processing time of process() loop.
- **calc_fft:** _optional_ whether to calculate frequency domain measures. Default = false Note: can cause slowdowns in some cases.
- **calc_freq:** _optional_ whether to calculate frequency domain measures. Default = false Note: can cause slowdowns in some cases.
- **freq_method:** _optional_ method used to extract the frequency spectrum. Available: 'fft' (Fourier Analysis), 'periodogram', and 'welch' (Welch's method), Default = 'welch'
- **interp_clipping:** if True, clipping parts of the signal are identified and the implied peak shape is interpolated. Default=False
- **clipping_scale:** whether to scale the data priod to clipping detection. Can correct errors if signal amplitude has been affected after digitization (for example through filtering). Default = False
- **interp_threshold**: the amplitude threshold beyond which will be checked for clipping. Recommended is to take this as the maximum value of the ADC with some margin for signal noise (default 1020, default ADC max 1024)
- **hampel_correct:** whether to reduce noisy segments using large median filter. Disabled by default due to computational complexity, and generally it is not necessary. Default = false.
- **bpmmin:** minimum value to see as likely for BPM when fitting peaks. Default = 40
- **bpmmax:** maximum value to see as likely for BPM when fitting peaks. Default = 180
- **reject_segmentwise:** whether to reject segments with more than 30% rejected beats. By default looks at segments of 10 beats at a time. Default = false.
- **high_precision:** _optional_ boolean, whether to estimate peak positions by upsampling hr signal to sample rate as specified in _high_precision_fs_. Default = False
- **high_precision_fs:** _optional_: the sample rate to which to upsample for ore accurate peak position estimation. Default = 1000 Hz, resulting in 1 ms peak position accuracy
- **measures:** measures dict in which results are stored. Custom dictionary can be passed, otherwise one is created and returned.
- **working_data:** working_data dict in which results are stored. Custom dictionary can be passed, otherwise one is created and returned.

Two `dict{}` objects are returned: one working data dict, and one containing all measures. Access as such:

```
import heartpy as hp

data = hp.load_exampledata(0)
fs = 100.0 #example file 0 is sampled at 100.0 Hz

working_data, measures = hp.process(data, fs, report_time=True)

print(measures['bpm']) #returns BPM value
print(measures['rmssd']) # returns RMSSD HRV measure

#You can also use Pandas if you so desire
import pandas as pd
df = pd.read_csv("data.csv", names=['hr'])
#note we need calc_freq if we want frequency-domain measures
working_data, measures = hp.process(df['hr'].values, fs, calc_freq=True)
print(measures['bpm'])
print(measures['lf/hf'])
```

### 3.1.3 Getting Data From Files

The toolkit has functionality to open and parse delimited .csv and .txt files, as well as matlab .mat files. [Find the data here](https://github.com/paulvangentcom/heartrate_analysis_python/tree/master/heartpy/data) Opening a file is done by the `get_data()` function:

```
import heartpy as hp

data = hp.get_data('data.csv')
```

This returns a 1-dimensional `numpy.ndarray` containing the heart rate data.

`get_data(filename, delim = ',', column_name = 'None')` requires one argument:

- **filename:** absolute or relative path to a valid (delimited .csv/.txt or matlab .mat) file;

Several optional arguments are available:

- **delim** _optional_: when loading a delimited .csv or .txt file, this specifies the delimiter used. Default delim = ‘,’;
- **column_name** _optional_: In delimited files with header: specifying column_name will return data from that column. Not specifying column_name for delimited files will assume the file contains only numerical data, returning np.nan values where data is not numerical. For matlab files: column_name specifies the table name in the matlab file.

Examples:

```
import heartpy as hp

#load data from a delimited file without header info
headerless_data = hp.get_data('data.csv')

#load data from column labeles 'hr' in a delimited file with header info
headered_data = hp.get_data('data2.csv', column_name = 'hr')

#load matlab file
```

(continues on next page)

(continued from previous page)

```
matlabdata = hp.get_data('data2.mat', column_name = 'hr')
#note that the column_name here represents the table name in the matlab file
```

### 3.1.4 Estimating Sample Rate

The toolkit has a simple built-in sample-rate detection. It can handle ms-based timers and datetime-based timers.

```
import heartpy as hp

#if you have a ms-based timer:
    mstimer_data = hp.get_data('data2.csv', column_name='timer')
fs = hp.get_samplerate_mstimer(mstimer_data)
    print(fs)

#if you have a datetime-based timer:
    datetime_data = hp.get_data('data3.csv', column_name='datetime')
fs = hp.get_samplerate_datetime(datetime_data, timeformat='%Y-%m-%d %H:%M:%S.%f')
    print(fs)
```

`get_samplerate_mstimer(timerdata)` requires one argument:

- **timerdata:** a list, numpy array or array-like object containing ms-based timestamps (float or int).

`get_samplerate_datetime(datetimedata, timeformat = '%H:%M:%S.f')` requires one argument:

- **datetimedata:** a list, numpy array or array-like object containing datetime-based timestamps (string);

One optional argument is available:

- **timeformat** _optional_: the format of the datetime-strings in your dataset. Default timeformat='%H:%M:%S.f', 24-hour based time including ms: 21:43:12.569.

### 3.1.5 Plotting Results

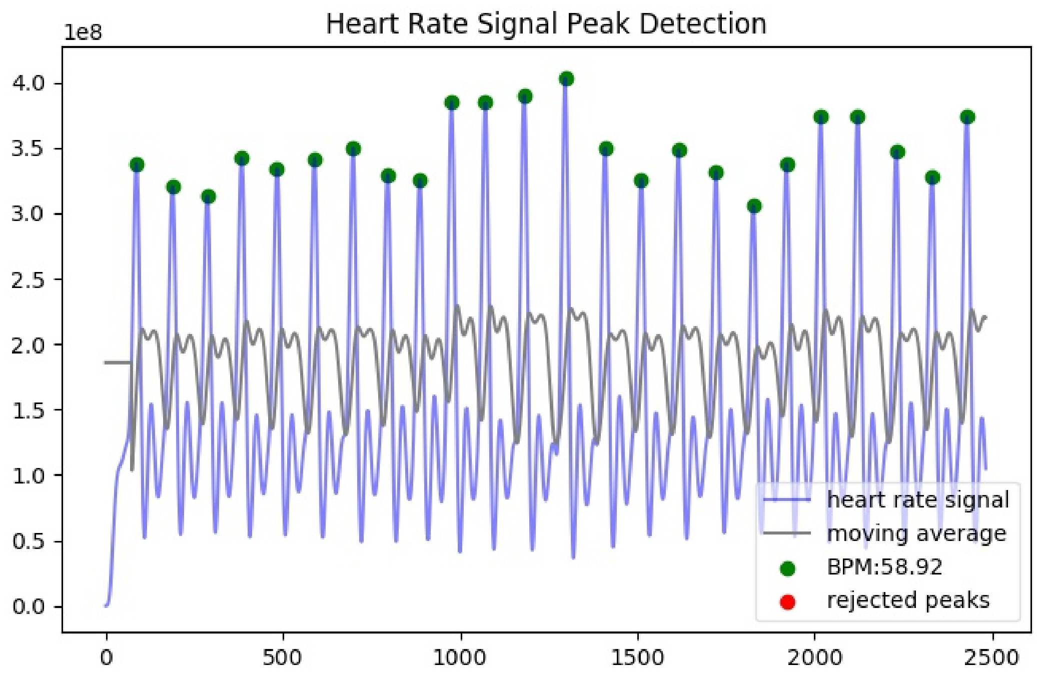
A plotting function is included. It plots the original signal and overlays the detected peaks and the rejected peaks (if any were rejected).

Example with the included *data.csv* example file (recorded at 100.0Hz):

```
import heartpy as hp

data = hp.get_data('data.csv')
working_data, measures = hp.process(data, 100.0)
hp.plotter(working_data, measures)
```

This returns:



`plotter(working_data, measures, show = True, title = 'Heart Rate Signal Peak Detection')` has two required arguments:

- **working_data** The working data `dict{}` container returned by the `process()` function.
- **measures** The measures `dict{}` container returned by the `process()` function.

Several optional arguments are available:

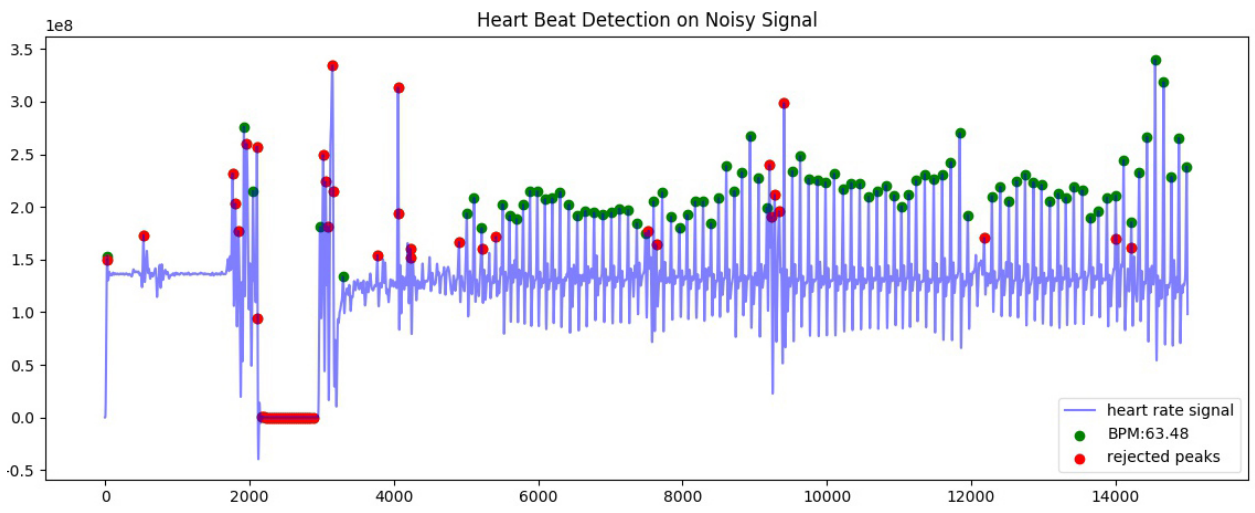
- **show** _optional_: if set to True a plot is visualised, if set to False a matplotlib.pyplot object is returned. Default show = True;
- **title** _optional_: Sets the title of the plot. If not specified, default title is used.

**Examples:**

```python
import heartpy as hp
hrdata = hp.get_data('data2.csv', column_name='hr')
timerdata = hp.get_data('data2.csv', column_name='timer')

working_data, measures = hp.process(hrdata, hp.get_samplerate_mstimer(timerdata))

#plot with different title
hp.plotter(working_data, measures, title='Heart Beat Detection on Noisy Signal')
```



Measures are only calculated for non-rejected peaks and intervals between two non-rejected peaks. Rejected detections do not influence the calculated measures.

By default a plot is visualised when plotter() is called. The function returns a matplotlib.pyplot object if the argument show=False is passed:

```
working_data, measures = hp.process(hrdata, hp.get_samplerate_mstimer(timerdata))
plot_object = hp.plotter(working_data, measures, show=False)
```

This returns:

```
<module 'matplotlib.pyplot' [...]>
```

Object can then be saved, appended to, or visualised:

```
working_data, measures = hp.process(hrdata, hp.get_samplerate_mstimer(timerdata))
plot_object = hp.plotter(working_data, measures, show=False)

plot_object.savefig('plot_1.jpg') #saves the plot as JPEG image.

plot_object.show() #displays plot
```

## Plotting results of segmentwise analysis

After calling *process_segmentwise()*, the returned working_data and measures contain analysis results on the segmented data. This can be visualised using the function *segment_plotter()*:

`segment_plotter(working_data, measures, title='Heart Rate Signal Peak Detection', path = '', start=0, end=None, step=1)`. The function has two required arguments:

- **working_data** The working data `dict{}` container returned by the `process_segmentwise()` function.
- **measures** The measures `dict{}` container returned by the `process_segmentwise()` function.

Several optional arguments are available:

- **title** _optional_: Sets the title of the plot. If not specified, default title is used.
- **path** _optional_: Where to save the plots. Folder will be created if it doesn't exist.
- **start** _optional_: segment index to start at, default = 0, beginning of segments.

- **end** _optional_: plotting stops when this segment index is reached. Default=None, which is interpreted as meaning plot until end of segment list.
- **step** _optional_: the stepsize of the plotting. Every step'th segment will be visualised. Default=1, meaning every segment.

### 3.1.6 Getting heart rate over time

There may be situations where you have a long heart rate signal, and want to compute how the heart rate measures change over time in the signal. HeartPy includes the *process_segmentwise* function that does just that!

Usage works like this:

```
working_data, measures = hp.process_segmentwise(data, sample_rate=100.0, segment_
↪width = 40, segment_overlap = 0.25)
```

What this will do is segment the data into sections of 40 seconds each. In this example each window will have an overlap with the previous window of 25%, meaning each iteration the 40 second window moves by 30 seconds.

*process_segmentwist()* expects two arguments: - data: 1-d numpy array or list containing heart rate data - sample_rate: the sample rate with which the data is collected, in Hz

Several optional arguments are possible:

- **segment_width**: the width of the window used, in seconds.
- **segment_overlap**: the fraction of overlap between adjacent windows: 0 <= segment_overlap < 1.0
- **replace_outliers**: bool, whether to replace outliers in the computed measures with the median
- **segment_min_size**: When segmenting, the tail end of the data if often shorter than the specified size in segment_width. The tail end is only included if it is longer than the *segment_min_size*. Default = 20. Setting this too low is not recommended as it may make peak fitting unstable, and it also doesn't make much sense from a biosignal analysis perspective to use very short data segments.
- **outlier_method**: which outlier detection method to use. The interquartile-range ('iqr') or modified z-score ('z-score') methods are available as of now. Default: 'iqr'
- **mode**: 'fast' or 'full'. The 'fast' method detects peaks over the entire signal, then segments and computes heart rate and heart rate variability measures. The 'full' method segments the data first, then runs the full analysis pipelin on each segment. For small numbers of segments (<10), there is not much difference and the fast method can actually be slower. The more segments there are, the larger the difference becomes. By default you should choose the 'fast' method. If there are problems with peak fitting, consider trying the 'full' method.
- **kwargs*: you can pass all the arguments normally passed to the *process()* function at the end of the arguments here as well. These will be passed on and used in the analysis. Example:

```
working_data, measures = hp.process_segmentwise(data, sample_rate=100.0, segment_
↪width = 40, segment_overlap = 0.25, calc_freq=True, reject_segmentwise=True, report_
↪time=True)
```

In this example the last three arguments will be passed on the the *process()* function and used in the analysis. For a full list of arguments that *process()* supports, see the *Basic Example*

### 3.1.7 Example Notebooks are available for further reading!

If you're looking for a few hands-on examples on how to get started with HeartPy, have a look at the links below! These notebooks show how to handle various analysis tasks with HeartPy, from smartwatch data, smart ring data, regular PPG, and regular (and very noisy) ECG. The notebooks sometimes don't render through the github engine, so either open them locally, or use an online viewer like [nbviewer](https://nbviewer.jupyter.org/).

We recommend you follow the notebooks in order: - [1. Analysing a PPG signal](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/1_regular_PPG/Analysing_a_PPG_signal.ipynb), a notebook for starting out with HeartPy using built-in examples. - [2. Analysing an ECG signal](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/2_regular_ECG/Analysing_a_regular_ECG_signal.ipynb), a notebook for working with HeartPy and typical ECG data. - [3. Analysing smartwatch data](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/3_smartwatch_data/Analysing_Smartwatch_Data.ipynb), a notebook on analysing low resolution PPG data from a smartwatch. - [4. Analysing smart ring data](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/4_smartring_data/Analysing_Smart_Ring_Data.ipynb), a notebook on analysing smart ring PPG data. - [5. Analysing noisy ECG data](https://github.com/paulvangentcom/heartrate_analysis_python/blob/master/examples/5_noisy_ECG/Analysing_Noisy_ECG.ipynb), an advanced notebook on working with very noisy ECG data, using data from the MIT-BIH noise stress test dataset.

## 3.2 API Reference

### 3.2.1 heartpy (main)

#### Main functions

`heartpy.`**`process`**(*hrdata, sample_rate, windowsize=0.75, report_time=False, calc_freq=False, freq_method='welch', freq_square=True, interp_clipping=False, clipping_scale=False, interp_threshold=1020, hampel_correct=False, bpmmin=40, bpmmax=180, reject_segmentwise=False, high_precision=False, high_precision_fs=1000.0, breathing_method='welch', clean_rr=False, clean_rr_method='quotient-filter', measures={}, working_data={}*)

processes passed heart rate data.

Processes the passed heart rate data. Returns measures{} dict containing results.

**Parameters**

- **`hrdata`** (`1d array or list`) – array or list containing heart rate data to be analysed
- **`sample_rate`** (`int or float`) – the sample rate with which the heart rate data is sampled
- **`windowsize`** (`int or float`) – the window size in seconds to use in the calculation of the moving average. Calculated as windowsize * sample_rate default : 0.75
- **`report_time`** (`bool`) – whether to report total processing time of algorithm default : True
- **`calc_freq`** (`bool`) – whether to compute time-series measurements default : False
- **`freq_method`** (`str`) – method used to extract the frequency spectrum. Available: 'fft' (Fourier Analysis), 'periodogram', and 'welch' (Welch's method). default : 'welch'
- **`freq_square`** (`bool`) – whether to square the power spectrum returned when computing frequency measures default : true
- **`interp_clipping`** (`bool`) – whether to detect and interpolate clipping segments of the signal default : True

- **clipping_scale** (*bool*) – whether to scale the data prior to clipping detection. Can correct errors if signal amplitude has been affected after digitization (for example through filtering). Not recommended by default. default : False
- **interp_threshold** (*int or float*) – threshold to use to detect clipping segments. Recommended to be a few datapoints below the sensor or ADC's maximum value (to account for slight data line noise). default : 1020, 4 below max of 1024 for 10-bit ADC
- **hampel_correct** (*bool*) – whether to reduce noisy segments using large median filter. Disabled by default due to computational complexity and (small) distortions induced into output measures. Generally it is not necessary. default : False
- **bpmmin** (*int or float*) – minimum value to see as likely for BPM when fitting peaks default : 40
- **bpmmax** (*int or float*) – maximum value to see as likely for BPM when fitting peaks default : 180
- **reject_segmentwise** (*bool*) – whether to reject segments with more than 30% rejected beats. By default looks at segments of 10 beats at a time. default : False
- **high_precision** (*bool*) – whether to estimate peak positions by upsampling signal to sample rate as specified in high_precision_fs default : false
- **high_precision_fs** (*int or float*) – the sample rate to which to upsample for more accurate peak position estimation default : 1000 Hz
- **breathing_method** (*str*) – method to use for estimating breathing rate, should be 'welch' or 'fft' default : fft
- **clean_rr** (*bool*) – if true, the RR_list is further cleaned with an outlier rejection pass default : false
- **clean_rr_method** (*str*) – how to find and reject outliers. Available methods are ' quotient-filter', 'iqr' (interquartile range), and 'z-score'. default : 'quotient-filter'
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function.
- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Returns**

- **working_data** (*dict*) – dictionary object used to store temporary values.
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures.

**Examples**

There's example data included in HeartPy to help you get up to speed. Here are provided two examples of how to approach heart rate analysis.

The first example contains noisy sections and comes with a timer column that counts miliseconds since start of recording.

```
>>> import heartpy as hp
>>> data, timer = hp.load_exampledata(1)
>>> sample_rate = hp.get_samplerate_mstimer(timer)
>>> '%.3f' %sample_rate
'116.996'
```

The sample rate is one of the most important characteristics during the heart rate analysis, as all measures are relative to this.

With all data loaded and the sample rate determined, analysis is now easy:

```
>>> wd, m = hp.process(data, sample_rate = sample_rate)
```

The measures ('m') dictionary returned contains all determined measures

```
>>> '%.3f' %m['bpm']
'62.376'
>>> '%.3f' %m['rmssd']
'57.070'
```

Using a slightly longer example:

```
>>> data, timer = hp.load_exampledata(2)
>>> print(timer[0])
2016-11-24 13:58:58.081000
```

As you can see something is going on here: we have a datetime-based timer. HeartPy can accomodate this and determine sample rate nontheless:

```
>>> sample_rate = hp.get_samplerate_datetime(timer, timeformat = '%Y-%m-%d %H:%M:
↪%S.%f')
>>> '%.3f' %sample_rate
'100.420'
```

Now analysis can proceed. Let's also compute frequency domain data and interpolate clipping. In this segment the clipping is visible around amplitude 980 so let's set that as well:

```
>>> data, timer = hp.load_exampledata(1)
>>> sample_rate = hp.get_samplerate_mstimer(timer)
>>> wd, m = hp.process(data, sample_rate = sample_rate, calc_freq = True,
... interp_clipping = True, interp_threshold = 975)
>>> '%.3f' %m['bpm']
'62.376'
>>> '%.3f' %m['rmssd']
'57.070'
>>> '%.3f' %m['lf/hf']
'0.893'
```

High precision mode will upsample 200ms of data surrounding detected peak and attempt to estimate the peak's real position with higher accuracy. Use high_precision_fs to set the virtual sample rate to which the peak will be upsampled (e.g. 1000Hz gives an estimated 1ms accuracy)

```
>>> wd, m = hp.process(data, sample_rate = sample_rate, calc_freq = True,
... high_precision = True, high_precision_fs = 1000.0)
```

Finally setting reject_segmentwise will reject segments with more than 30% rejected beats See check_binary_quality in the peakdetection.py module.

```
>>> wd, m = hp.process(data, sample_rate = sample_rate, calc_freq = True,
... reject_segmentwise = True)
```

Final test for code coverage, let's turn all bells and whistles on that haven't been tested yet

```
>>> wd, m = hp.process(data, sample_rate = 100.0, calc_freq = True,
... interp_clipping = True, clipping_scale = True, reject_segmentwise = True,␣
↪clean_rr = True)
```

heartpy.**process_segmentwise**(*hrdata*, *sample_rate*, *segment_width=120*, *segment_overlap=0*, *segment_min_size=20*, *replace_outliers=False*, *outlier_method='iqr'*, *mode='full'*, ***kwargs*)

processes passed heart rate data with a windowed function

Analyses a long heart rate data array by running a moving window over the data, computing measures in each iteration. Both the window width and the overlap with the previous window location are settable.

**Parameters**

- **hrdata** (*1d array or list*) – array or list containing heart rate data to be analysed
- **sample_rate** (*int or float*) – the sample rate with which the heart rate data is sampled
- **segment_width** (*int or float*) – width of segments in seconds default : 120
- **segment_overlap** (*float*) – overlap fraction of adjacent segments. Needs to be 0 <= segment_overlap < 1. default : 0 (no overlap)
- **segment_min_size** (*int*) – often a tail end of the data remains after segmenting into segments. segment_min_size indicates the minimum length (in seconds) the tail end needs to be in order to be included in analysis. It is discarded if it's shorter. default : 20
- **replace_outliers** (*bool*) – whether to detct and replace outliers in the segments. Will iterate over all computed measures and evaluate each.
- **outlier_method** (*str*) – what method to use to detect outlers. Available are 'iqr', which uses the inter-quartile range, and 'z-score', which uses the modified z-score approach.
- **mode** (*str*) – 'full' or 'fast'
- **arguments** (*Keyword*) –
- **------------------** –
- **-- 1-dimensional numpy array or list containing heart rate data** (*hrdata*) –
- **-- the sample rate of the heart rate data** (*sample_rate*) –
- **-- the width of the segment, in seconds, within which all measures** (*segment_width*) – will be computed.
- **-- the fraction of overlap of adjacent segments,** (*segment_overlap*) – needs to be 0 <= segment_overlap < 1
- **-- After segmenting the data, a tail end will likely remain that is shorter than the specified** (*segment_min_size*) – segment_size. segment_min_size sets the minimum size for the last segment of the generated series of segments to still be included. Default = 20.
- **-- bool, whether to replace outliers (likely caused by peak fitting** (*replace_outliers*) – errors on one or more segments) with the median.
- **-- which method to use to detect outliers. Available are the** (*outlier_method*) – 'interquartile-range' ('iqr') and the 'modified z-score' ('z-score') methods.

**Returns**

- **working_data** (*dict*) – dictionary object used to store temporary values.
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures.

**Examples**

Given one of the included example datasets we can demonstrate this function:

```
>>> import heartpy as hp
>>> data, timer = hp.load_exampledata(2)
>>> sample_rate = hp.get_samplerate_datetime(timer, timeformat = '%Y-%m-%d %H:%M:
↪%S.%f')
>>> wd, m = hp.process_segmentwise(data, sample_rate, segment_width=120, segment_
↪overlap=0.5)
>>> len(m['bpm'])
11
```

The function has split the data into 11 segments and analysed each one. Every key in the measures (m) dict now contains a list of that measure for each segment.

```
>>> [round(x, 1) for x in m['bpm']]
[100.0, 96.8, 97.2, 97.9, 96.7, 96.8, 96.8, 95.0, 92.9, 96.7, 99.2]
```

Specifying mode = 'fast' will run peak detection once and use detections to compute measures over each segment. Useful for speed ups, but typically the full mode has better results.

```
>>> wd, m = hp.process_segmentwise(data, sample_rate, segment_width=120, segment_
↪overlap=0.5,
... mode = 'fast', replace_outliers = True)
```

You can specify the outlier detection method ('iqr' - interquartile range, or 'z-score' for modified z-score approach).

```
>>> wd, m = hp.process_segmentwise(data, sample_rate, segment_width=120, segment_
↪overlap=0.5,
... mode = 'fast', replace_outliers = True, outlier_method = 'z-score')
```

## Visualisation

`heartpy.`**`plotter`**(*working_data*, *measures*, *show=True*, *title='Heart Rate Signal Peak Detection'*, *moving_average=False*)

plots the analysis results.

Function that uses calculated measures and data stored in the working_data{} and measures{} dict objects to visualise the fitted peak detection solution.

**Parameters**

- **`working_data`** (`dict`) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function
- **`measures`** (`dict`) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function
- **`show`** (`bool`) – when False, function will return a plot object rather than display the results. default : True

- **title** (*string*) – title for the plot. default : "Heart Rate Signal Peak Detection"
- **moving_average** (*bool*) – whether to display the moving average on the plot. The moving average is used for peak fitting. default: False

**Returns** **out** – only returned if show == False.

**Return type** matplotlib plot object

**Examples**

First let's load and analyse some data to visualise

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(0)
>>> wd, m = hp.process(data, 100.0)
```

Then we can visualise

```
>>> plot_object = plotter(wd, m, show=False, title='some awesome title')
```

This returns a plot object which can be visualized or saved or appended. See matplotlib API for more information on how to do this.

A matplotlib plotting object is returned. This can be further processed and saved to a file.

heartpy.**segment_plotter**(*working_data*, *measures*, *title='Heart Rate Signal Peak Detection'*, *figsize=(6, 6)*, *path=''*, *start=0*, *end=None*, *step=1*)

plots analysis results

Function that plots the results of segmentwise processing of heart rate signal and writes all results to separate files at the path provided.

**Parameters**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function
- **title** (*str*) – the title used in the plot
- **figsize** (*tuple*) – figsize tuple to be passed to matplotlib
- **path** (*str*) – the path where the files will be stored, folder must exist.
- **start** (*int*) – what segment to start plotting with default : 0
- **end** (*int*) – last segment to plot. Must be smaller than total number of segments default : None, will plot until end
- **step** (*int*) – stepsize used when iterating over plots every step'th segment will be plotted default : 1

**Returns**

**Return type** None

**Examples**

This function has no examples. See documentation of heartpy for more info.

## Preprocessing functions

heartpy.**enhance_peaks**(*hrdata*, *iterations=2*)

enhances peak amplitude relative to rest of signal

Function thta attempts to enhance the signal-noise ratio by accentuating the highest peaks. Note: denoise first

**Parameters**

- **hrdata** (*1-d numpy array or list*) – sequence containing heart rate data
- **iterations** (*int*) – the number of scaling steps to perform default : 2

**Returns** **out** – array containing enhanced peaks

**Return type** 1-d numpy array

**Examples**

Given an array of data, the peaks can be enhanced using the function

```
>>> x = [200, 300, 500, 900, 500, 300, 200]
>>> enhance_peaks(x)
array([   0.        ,    4.31776016,   76.16528926, 1024.        ,
         76.16528926,    4.31776016,    0.        ])
```

heartpy.**enhance_ecg_peaks**(*hrdata*, *sample_rate*, *iterations=4*, *aggregation='mean'*, *notch_filter=True*)

enhances ECG peaks

Function that convolves synthetic QRS templates with the signal, leading to a strong increase signal-to-noise ratio. Function ends with an optional Notch filterstep (default : true) to reduce noise from the iterating convolution steps.

**Parameters**

- **hrdata** (*1-d numpy array or list*) – sequence containing heart rate data
- **sample_rate** (*int or float*) – sample rate with which the data is sampled
- **iterations** (*int*) – how many convolutional iterations should be run. More will result in stronger peak enhancement, but over a certain point (usually between 12-16) overtones start appearing in the signal. Only increase this if the peaks aren't amplified enough. default : 4
- **aggregation** (*str*) – how the data from the different convolutions should be aggregated. Can be either 'mean' or 'median'. default : mean
- **notch_filter** (*bool*) – whether to apply a notch filter after the last convolution to get rid of remaining low frequency noise. default : true

**Returns** **output** – The array containing the filtered data with enhanced peaks

**Return type** 1d array

**Examples**

First let's import the module and load the data

```
>>> import heartpy as hp
>>> data, timer = hp.load_exampledata(1)
>>> sample_rate = hp.get_samplerate_mstimer(timer)
```

After loading the data we call the function like so:

```
>>> filtered_data = enhance_ecg_peaks(data, sample_rate, iterations = 3)
```

By default the module uses the mean to aggregate convolutional outputs. It is also possible to use the median.

```
>>> filtered_data = enhance_ecg_peaks(data, sample_rate, iterations = 3,
... aggregation = 'median', notch_filter = False)
```

In the last example we also disabled the notch filter.

---

`heartpy.`**`flip_signal`**(*data*, *enhancepeaks=False*, *keep_range=True*)
invert signal waveforms.

Function that flips raw signal with negative mV peaks to normal ECG. Required for proper peak finding in case peaks are expressed as negative dips.

**Parameters**

- **`data`** (*`1d list or numpy array`*) – data section to be evaluated
- **`enhance_peaks`** (*`bool`*) – whether to apply peak accentuation default : False
- **`keep_range`** (*`bool`*) – whether to scale the inverted data so that the original range is maintained

**Returns** out

**Return type** 1d array

---

**Examples**

Given an array of data

```
>>> x = [200, 300, 500, 900, 500, 300, 200]
```

We can call the function. If keep_range is False, the signal will be inverted relative to its mean.

```
>>> flip_signal(x, keep_range=False)
array([628.57142857, 528.57142857, 328.57142857, -71.42857143,
       328.57142857, 528.57142857, 628.57142857])
```

However, by specifying keep_range, the inverted signal will be put 'back in place' in its original range.

```
>>> flip_signal(x, keep_range=True)
array([900., 800., 600., 200., 600., 800., 900.])
```

It's also possible to use the enhance_peaks function:

```
>>> flip_signal(x, enhancepeaks=True)
array([1024.        ,  621.75746332,  176.85545623,    0.        ,
        176.85545623,  621.75746332, 1024.        ])
```

`heartpy.`**`remove_baseline_wander`**(*data*, *sample_rate*, *cutoff=0.05*)
removes baseline wander

Function that uses a Notch filter to remove baseline wander from (especially) ECG signals

**Parameters**

- **`data`** (*`1-dimensional numpy array or list`*) – Sequence containing the to be filtered data
- **`sample_rate`** (*`int or float`*) – the sample rate with which the passed data sequence was sampled
- **`cutoff`** (*`int, float`*) – the cutoff frequency of the Notch filter. We recommend 0.05Hz. default : 0.05

**Returns** **out** – 1d array containing the filtered data

**Return type** 1d array

**Examples**

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(0)
```

baseline wander is removed by calling the function and specifying the data and sample rate.

```
>>> filtered = remove_baseline_wander(data, 100.0)
```

`heartpy.`**`scale_data`**(*data*, *lower=0*, *upper=1024*)
scales passed sequence between thresholds

Function that scales passed data so that it has specified lower and upper bounds.

**Parameters**

- **`data`** (*`1-d array or list`*) – Sequence to be scaled
- **`lower`** (*`int or float`*) – lower threshold for scaling default : 0
- **`upper`** (*`int or float`*) – upper threshold for scaling default : 1024

**Returns** **out** – contains scaled data

**Return type** 1-d array

**Examples**

When passing data without further arguments to the function means it scales 0-1024

```
>>> x = [2, 3, 4, 5]
>>> scale_data(x)
array([   0.        ,  341.33333333,  682.66666667, 1024.        ])
```

Or you can specify a range:

```
>>> scale_data(x, lower = 50, upper = 124)
array([ 50.        ,  74.66666667,  99.33333333, 124.        ])
```

heartpy.**scale_sections**(*data*, *sample_rate*, *windowsize=2.5*, *lower=0*, *upper=1024*)
scales data using sliding window approach

Function that scales the data within the defined sliding window between the defined lower and upper bounds.

**Parameters**

- **data** (*1-d array or list*) – Sequence to be scaled
- **sample_rate** (*int or float*) – Sample rate of the passed signal
- **windowsize** (*int or float*) – size of the window within which signal is scaled, in seconds default : 2.5
- **lower** (*int or float*) – lower threshold for scaling. Passed to scale_data. default : 0
- **upper** (*int or float*) – upper threshold for scaling. Passed to scale_data. default : 1024

**Returns** **out** – contains scaled data

**Return type** 1-d array

**Examples**

```
>>> x = [20, 30, 20, 30, 70, 80, 20, 30, 20, 30]
>>> scale_sections(x, sample_rate=1, windowsize=2, lower=20, upper=30)
array([20., 30., 20., 30., 20., 30., 20., 30., 20., 30.])
```

## Utilities

heartpy.**get_data**(*filename*, *delim='*, *'*, *column_name='None'*, *encoding=None*, *ignore_extension=False*)
load data from file

Function to load data from a .CSV or .MAT file into numpy array. File can be accessed from local disk or url.

**Parameters**

- **filename** (*string*) – absolute or relative path to the file object to read
- **delim** (*string*) – the delimiter used if CSV file passed default : ','
- **column_name** (*string*) – for CSV files with header: specify column that contains the data for matlab files it specifies the table name that contains the data default : 'None'
- **ignore_extension** (*bool*) – if True, extension is not tested, use for example for files where the extention is not .csv or .txt but the data is formatted as if it is. default : False

**Returns** **out** – array containing the data from the requested column of the specified file

**Return type** 1-d numpy array

**Examples**

As an example, let's load two example data files included in the package For this we use pkg_resources for automated testing purposes, you don't need this when using the function.

```
>>> from pkg_resources import resource_filename
>>> filepath = resource_filename(__name__, 'data/data.csv')
```

So, assuming your file lives at 'filepath', you open it as such:

```
>>> get_data(filepath)
array([530., 518., 506., ..., 492., 493., 494.])
```

Files with multiple columns can be opened by specifying the 'column_name' where the data resides:

```
>>> filepath = resource_filename(__name__, 'data/data2.csv')
```

Again you don't need the above. It is there for automated testing.

```
>>> get_data(filepath, column_name='timer')
array([0.00000000e+00, 8.54790319e+00, 1.70958064e+01, ...,
       1.28192904e+05, 1.28201452e+05, 1.28210000e+05])
```

You can open matlab files in much the same way by specifying the column where the data lives:

```
>>> filepath = resource_filename(__name__, 'data/data2.mat')
```

Again you don't need the above. It is there for automated testing. Open matlab file by specifying the column name as well:

```
>>> get_data(filepath, column_name='hr')
array([515., 514., 514., ..., 492., 494., 496.])
```

You can any csv formatted text file no matter the extension if you set ignore_extension to True:

```
>>> filepath = resource_filename(__name__, 'data/data.log')
>>> get_data(filepath, ignore_extension = True)
array([530., 518., 506., ..., 492., 493., 494.])
```

You can specify column names in the same way when using ignore_extension

```
>>> filepath = resource_filename(__name__, 'data/data2.log')
>>> data = get_data(filepath, column_name = 'hr', ignore_extension = True)
```

---

heartpy.**load_exampledata**(*example=0*)

loads example data

Function to load one of the example datasets included in HeartPy and used in the documentation.

**Parameters** **example** (*int (0, 1, 2)*) – selects example data used in docs of three datafiles. Available (see github repo for source of files): 0 : data.csv 1 : data2.csv 2 : data3.csv default : 0

**Returns** **out** – Contains the data and timer column. If no timer data is available, such as in example 0, an empty second array is returned.

**Return type** tuple of two arrays

---

### Examples

This function can load one of the three example data files provided with HeartPy. It returns both the data and a timer if that is present

For example:

```
>>> data, _ = load_exampledata(0)
>>> data[0:5]
array([530., 518., 506., 494., 483.])
```

And another example:

```
>>> data, timer = load_exampledata(1)
>>> [round(x, 2) for x in timer[0:5]]
[0.0, 8.55, 17.1, 25.64, 34.19]
```

---

heartpy.**get_samplerate_mstimer**(*timerdata*)

detemine sample rate based on ms timer

Function to determine sample rate of data from ms-based timer list or array.

**Parameters** **timerdata** (*1d numpy array or list*) – sequence containing values of a timer, in ms

**Returns** **out** – the sample rate as determined from the timer sequence provided

**Return type** float

---

**Examples**

first we load a provided example dataset

```
>>> data, timer = load_exampledata(example = 1)
```

since it's a timer that counts miliseconds, we use this function. Let's also round to three decimals

```
>>> round(get_samplerate_mstimer(timer), 3)
116.996
```

of course if another time unit is used, converting it to ms-based should be trivial.

---

heartpy.**get_samplerate_datetime**(*datetimedata*, *timeformat='%H:%M:%S.%f'*)

determine sample rate based on datetime

Function to determine sample rate of data from datetime-based timer list or array.

**Parameters**

- **timerdata** (*1-d numpy array or list*) – sequence containing datetime strings
- **timeformat** (*string*) – the format of the datetime-strings in datetimedata default : '%H:%M:%S.f' (24-hour based time including ms: e.g. 21:43:12.569)

**Returns** **out** – the sample rate as determined from the timer sequence provided

**Return type** float

---

**Examples**

We load the data like before

```
>>> data, timer = load_exampledata(example = 2)
>>> timer[0]
'2016-11-24 13:58:58.081000'
```

Note that we need to specify the timeformat used so that datetime understands what it's working with:

```
>>> round(get_samplerate_datetime(timer, timeformat = '%Y-%m-%d %H:%M:%S.%f'), 3)
100.42
```

---

## Filtering

`heartpy.`**`filter_signal`**(*data*, *cutoff*, *sample_rate*, *order=2*, *filtertype='lowpass'*, *return_top=False*)

Apply the specified filter

Function that applies the specified lowpass, highpass or bandpass filter to the provided dataset.

**Parameters**

- **data** (*1-dimensional numpy array or list*) – Sequence containing the to be filtered data
- **cutoff** (*int, float or tuple*) – the cutoff frequency of the filter. Expects float for low and high types and for bandpass filter expects list or array of format [lower_bound, higher_bound]
- **sample_rate** (*int or float*) – the sample rate with which the passed data sequence was sampled
- **order** (*int*) – the filter order default : 2
- **filtertype** (*str*) – The type of filter to use. Available: - lowpass : a lowpass butterworth filter - highpass : a highpass butterworth filter - bandpass : a bandpass butterworth filter - notch : a notch filter around specified frequency range both the highpass and notch filter are useful for removing baseline wander. The notch filter is especially useful for removing baseling wander in ECG signals.

**Returns** **out** – 1d array containing the filtered data

**Return type** 1d array

---

**Examples**

```
>>> import numpy as np
>>> import heartpy as hp
```

Using standard data provided

```
>>> data, _ = hp.load_exampledata(0)
```

We can filter the signal, for example with a lowpass cutting out all frequencies of 5Hz and greater (with a sloping frequency cutoff)

```
>>> filtered = filter_signal(data, cutoff = 5, sample_rate = 100.0, order = 3,␣
↪filtertype='lowpass')
>>> print(np.around(filtered[0:6], 3))
[530.175 517.893 505.768 494.002 482.789 472.315]
```

Or we can cut out all frequencies below 0.75Hz with a highpass filter:

```
>>> filtered = filter_signal(data, cutoff = 0.75, sample_rate = 100.0, order = 3,␣
↪filtertype='highpass')
>>> print(np.around(filtered[0:6], 3))
[-17.975 -28.271 -38.609 -48.992 -58.422 -67.902]
```

Or specify a range (here: 0.75 - 3.5Hz), outside of which all frequencies are cut out.

```
>>> filtered = filter_signal(data, cutoff = [0.75, 3.5], sample_rate = 100.0,
... order = 3, filtertype='bandpass')
>>> print(np.around(filtered[0:6], 3))
[-12.012 -23.159 -34.261 -45.12  -55.541 -65.336]
```

A 'Notch' filtertype is also available (see remove_baseline_wander).

```
>>> filtered = filter_signal(data, cutoff = 0.05, sample_rate = 100.0, filtertype=
↪'notch')
```

Finally we can use the return_top flag to only return the filter response that has amplitute above zero. We're only interested in the peaks, and sometimes this can improve peak prediction:

```
>>> filtered = filter_signal(data, cutoff = [0.75, 3.5], sample_rate = 100.0,
... order = 3, filtertype='bandpass', return_top = True)
>>> print(np.around(filtered[48:53], 3))
[ 0.      0.      0.409 17.088 35.673]
```

heartpy.**hampel_filter**(*data, filtsize=6*)

Detect outliers based on hampel filter

Funcion that detects outliers based on a hampel filter. The filter takes datapoint and six surrounding samples. Detect outliers based on being more than 3std from window mean. See: https://www.mathworks.com/help/signal/ref/hampel.html

**Parameters**

- **data** (*1d list or array*) – list or array containing the data to be filtered
- **filtsize** (*int*) – the filter size expressed the number of datapoints taken surrounding the analysed datapoint. a filtsize of 6 means three datapoints on each side are taken. total filtersize is thus filtsize + 1 (datapoint evaluated)

**Returns** out

**Return type** array containing filtered data

**Examples**

```
>>> from .datautils import get_data, load_exampledata
>>> data, _ = load_exampledata(0)
>>> filtered = hampel_filter(data, filtsize = 6)
>>> print('%i, %i' %(data[1232], filtered[1232]))
497, 496
```

heartpy.**hampel_correcter**(*data, sample_rate*)

apply altered version of hampel filter to suppress noise.

Function that returns te difference between data and 1-second windowed hampel median filter. Results in strong noise suppression characteristics, but relatively expensive to compute.

Result on output measures is present but generally not large. However, use sparingly, and only when other means have been exhausted.

**Parameters**

- **data** (*1d numpy array*) – array containing the data to be filtered
- **sample_rate** (*int or float*) – sample rate with which data was recorded

**Returns** **out** – array containing filtered data

**Return type** 1d numpy array

---

**Examples**

```
>>> from .datautils import get_data, load_exampledata
>>> data, _ = load_exampledata(1)
>>> filtered = hampel_correcter(data, sample_rate = 116.995)
```

---

## 3.2.2 analysis

Functions that handle computation of heart rate (HR) and heart rate variability (HRV) measures.

heartpy.analysis.**calc_rr**(*peaklist*, *sample_rate*, *working_data={}*)
calculates peak-peak intervals

Function that calculates the peak-peak data required for further analysis. Stores results in the working_data{} dict.

**Parameters**

- **peaklist** (*1d list or array*) – list or array containing detected peak positions
- **sample_rate** (*int or float*) – the sample rate with which the heart rate signal is collected
- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Returns** **working_data** – working_data dictionary object containing all of heartpy's temp objects

**Return type** dict

---

**Examples**

Let's assume we detected peaks at these positions in the signal:

```
>>> peaklist = [200, 280, 405, 501, 615]
```

It is then easy to call calc_rr to compute what we need:

```
>>> wd = calc_rr(peaklist, sample_rate = 100.0)
>>> wd['RR_list']
array([ 800., 1250.,  960., 1140.])
>>> wd['RR_diff']
array([450., 290., 180.])
>>> wd['RR_sqdiff']
array([202500.,  84100.,  32400.])
```

Note that the list of peak-peak intervals is of length len(peaks) - 1 the length of the differences is of length len(peaks) - 2

---

heartpy.analysis.**update_rr**(*working_data={}*)

updates differences between adjacent peak-peak distances

Function that updates RR differences and RR squared differences based on corrected RR list

**Parameters** **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects will be created if not passed to function

**Returns** **out** – working_data dictionary object containing all of heartpy's temp objects

**Return type** dict

---

**Examples**

Let's assume we detected peaks at these positions in the signal:

```
>>> peaklist = [200, 280, 405, 410, 501, 615]
```

And we subsequently ran further analysis:

```
>>> wd = calc_rr(peaklist, sample_rate = 100.0)
```

The peak at position 410 is likely an incorrect detection and will be marked as such by other heartpy functions. This is indicated by an array 'binary_peaklist' in working_data. Binary peaklist is of the same length as peaklist, and is formatted as a mask:

For now let's set it manually, normally this is done by the check_peaks() function from HeartPy's peakdetection module.

```
>>> wd['binary_peaklist'] = [1, 1, 1, 0, 1, 1]
```

Rejected peaks are marked with a zero and accepted with a 1.

By now running update_rr(), heartpy will update all associated measures and will only compute peak-peak intervals between two accepted peaks.

```
>>> wd = update_rr(wd)
```

This will have generated a corrected RR_list object in the dictionary:

```
>>> wd['RR_list_cor']
[800.0, 1250.0, 1140.0]
```

As well as updated the lists RR_diff (differences between adjacent peak-peak intervals) and RR_sqdiff (squared differences between adjacent peak-peak intervals).

---

heartpy.analysis.**calc_rr_segment**(*rr_source*, *b_peaklist*)

calculates peak-peak differences for segmentwise processing

Function that calculates rr-measures when analysing segmentwise in the 'fast' mode.

**Parameters**

- **rr_source** (*1d list or array*) – list or array containing peak-peak intervals.
- **b_peaklist** (*1d list or array*) – list or array containing mask for peaklist.

**Returns**

- **rr_list** (*array*) – array containing peak-peak intervals.
- **rr_diff** (*array*) – array containing differences between adjacent peak-peak intervals
- **rr_sqdiff** (*array*) – array containing squared differences between adjacent peak-peak intervals

**Examples**

The function works in the same way as update_rr, except it returns three separate objects. It's used by process_segmentwise. Revert to doc on update_rr for more information.

```
>>> rr, rrd, rrsd = calc_rr_segment(rr_source = [ 800., 1250.,   50.,   910., 1140.
↪, 1002., 1142.],
... b_peaklist = [1, 1, 1, 0, 1, 1, 1, 1])
>>> print(rr)
[800.0, 1250.0, 1140.0, 1002.0, 1142.0]
>>> print(rrd)
[450.0 138.0 140.0]
>>> print(rrsd)
[202500.  19044.  19600.]
```

heartpy.analysis.**clean_rr_intervals**(*working_data*, *method='quotient-filter'*)

detects and rejects outliers in peak-peak intervals

Function that detects and rejects outliers in the peak-peak intervals. It updates the RR_list_cor in the working data dict

**Parameters**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. Needs to contain RR_list_cor, meaning one analysis cycle has already completed.
- **method** (*str*) – which method to use for outlier rejection, included are: - 'quotient-filter', based on the work in "Piskorki, J., Guzik, P. (2005), Filtering Poincare plots", - 'iqr', which uses the inter-quartile range, - 'z-score', which uses the modified z-score method. default : quotient-filter

**Returns** **working_data** – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Return type** dict

**Examples**

Let's load some data

```
>>> import heartpy as hp
>>> data, timer = hp.load_exampledata(1)
>>> sample_rate = hp.get_samplerate_mstimer(timer)
```

Run at least one analysis cycle first so that the dicts are populated

```
>>> wd, m = hp.process(data, sample_rate)
>>> wd = clean_rr_intervals(working_data = wd)
>>> ['%.3f' %x for x in wd['RR_list_cor'][0:5]]
['897.470', '811.997', '829.091', '777.807', '803.449']
```

You can also specify the outlier rejection method to be used, for example using the z-score method:

```
>>> wd = clean_rr_intervals(working_data = wd, method = 'z-score')
>>> ['%.3f' %x for x in wd['RR_list_cor'][0:5]]
['897.470', '811.997', '829.091', '777.807', '803.449']
```

Or the inter-quartile range (iqr) based method:

```
>>> wd = clean_rr_intervals(working_data = wd, method = 'iqr')
>>> ['%.3f' %x for x in wd['RR_list_cor'][0:5]]
['897.470', '811.997', '829.091', '965.849', '803.449']
```

---

`heartpy.analysis.`**`calc_ts_measures`**(*rr_list, rr_diff, rr_sqdiff, measures={}, working_data={}*)
calculates standard time-series measurements.

Function that calculates the time-series measurements for HeartPy.

**Parameters**

- **`rr_list`** (*`1d list or array`*) – list or array containing peak-peak intervals
- **`rr_diff`** (*`1d list or array`*) – list or array containing differences between adjacent peak-peak intervals
- **`rr_sqdiff`** (*`1d list or array`*) – squared rr_diff
- **`measures`** (*`dict`*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function.
- **`working_data`** (*`dict`*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Returns**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects.
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures.

---

**Examples**

Normally this function is called during the process pipeline of HeartPy. It can of course also be used separately.

Assuming we have the following peak-peak distances:

```
>>> import numpy as np
>>> rr_list = [1020.0, 990.0, 960.0, 1000.0, 1050.0, 1090.0, 990.0, 900.0, 900.0,␣
↪950.0, 1080.0]
```

we can then compute the other two required lists by hand for now:

```
>>> rr_diff = np.diff(rr_list)
>>> rr_sqdiff = np.power(rr_diff, 2)
>>> wd, m = calc_ts_measures(rr_list, rr_diff, rr_sqdiff)
```

All output measures are then accessible from the measures object through their respective keys:

```
>>> print('%.3f' %m['bpm'])
60.384
```

(continues on next page)

(continued from previous page)

```
>>> print('%.3f' %m['rmssd'])
67.082
```

heartpy.analysis.**calc_fd_measures**(*method='welch'*, *square_spectrum=True*, *measures={}*, *working_data={}*)
calculates the frequency-domain measurements.

Function that calculates the frequency-domain measurements for HeartPy.

**method** [str] method used to compute the spectrogram of the heart rate. available methods: fft, periodogram, and welch default : welch

**square_spectrum** [bool] whether to square the power spectrum returned. default : true

**measures** [dict] dictionary object used by heartpy to store computed measures. Will be created if not passed to function.

**working_data** [dict] dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**working_data** [dict] dictionary object that contains all heartpy's working data (temp) objects.

**measures** [dict] dictionary object used by heartpy to store computed measures.

Normally this function is called during the process pipeline of HeartPy. It can of course also be used separately.

Let's load an example and get a list of peak-peak intervals

```
>>> import heartpy as hp
>>> data, timer = hp.load_exampledata(2)
>>> sample_rate = hp.get_samplerate_datetime(timer, timeformat='%Y-%m-%d
↪%H:%M:%S.%f')
>>> wd, m = hp.process(data, sample_rate)
```

wd now contains a list of peak-peak intervals that has been cleaned of outliers ('RR_list_cor'). Calling the function then is easy

```
>>> wd, m = calc_fd_measures(method = 'periodogram', measures = m,␣
↪working_data = wd)
>>> print('%.3f' %m['lf/hf'])
4.964
```

Available methods are 'fft', 'welch' and 'periodogram'. To set another method, do:

```
>>> wd, m = calc_fd_measures(method = 'fft', measures = m, working_data =␣
↪wd)
>>> print('%.3f' %m['lf/hf'])
4.964
```

If there are no valid peak-peak intervals specified, returned measures are NaN: >>> wd['RR_list_cor'] = [] >>> wd, m = calc_fd_measures(working_data = wd) >>> np.isnan(m['lf/hf']) True

If there are rr-intervals but not enough to reliably compute frequency measures, a warning is raised:

RuntimeWarning: Short signal. ———Warning:——— too few peak-peak intervals for (reliable) frequency domain measure computation, frequency output measures are still computed but treat them with caution!

HF is usually computed over a minimum of 1 minute of good signal. LF is usually computed over a minimum of 2 minutes of good signal. The LF/HF ratio is usually computed over minimum 24 hours, although an absolute minimum of 5 min has also been suggested.

For more info see:

**Shaffer, F., Ginsberg, J.P. (2017).** An Overview of Heart Rate Variability Metrics and Norms.

Task Force of Pacing and Electrophysiology (1996), Heart Rate Variability in: European Heart Journal, vol.17, issue 3, pp354-381

`heartpy.analysis.`**`calc_breathing`**(*rrlist, method='welch', filter_breathing=True, bw_cutoff=[0.1, 0.4], measures={}, working_data={}*)

estimates breathing rate

Function that estimates breathing rate from heart rate signal. Upsamples the list of detected rr_intervals by interpolation then tries to extract breathing peaks in the signal.

**Parameters**

- **`rr_list`** (*`1d list or array`*) – list or array containing peak-peak intervals
- **`method`** (*`str`*) – method to use to get the spectrogram, must be 'fft' or 'welch' default : fft
- **`filter_breathing`** (*`bool`*) – whether to filter the breathing signal derived from the peak-peak intervals default : True
- **`bw_cutoff`** (*`list or tuple`*) – breathing frequency range expected default : [0.1, 0.4], meaning between 6 and 24 breaths per minute
- **`measures`** (*`dict`*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function.
- **`working_data`** (*`dict`*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Returns** **measures** – dictionary object used by heartpy to store computed measures.

**Return type** dict

**Examples**

Normally this function is called during the process pipeline of HeartPy. It can of course also be used separately.

Let's load an example and get a list of peak-peak intervals

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(0)
>>> wd, m = hp.process(data, 100.0)
```

Breathing is then computed with the function

```
>>> m, wd = calc_breathing(wd['RR_list_cor'], measures = m, working_data = wd)
>>> round(m['breathingrate'], 3)
0.171
```

There we have it, .17Hz, or about one breathing cycle in 6.25 seconds.

### 3.2.3 datautils

Functions for loading and slicing data

`heartpy.datautils.get_data`(*filename*, *delim=',  '*, *column_name='None'*, *encoding=None*, *ignore_extension=False*)

load data from file

Function to load data from a .CSV or .MAT file into numpy array. File can be accessed from local disk or url.

**Parameters**

- **filename** (*string*) – absolute or relative path to the file object to read
- **delim** (*string*) – the delimiter used if CSV file passed default : ','
- **column_name** (*string*) – for CSV files with header: specify column that contains the data for matlab files it specifies the table name that contains the data default : 'None'
- **ignore_extension** (*bool*) – if True, extension is not tested, use for example for files where the extention is not .csv or .txt but the data is formatted as if it is. default : False

**Returns** **out** – array containing the data from the requested column of the specified file

**Return type** 1-d numpy array

**Examples**

As an example, let's load two example data files included in the package For this we use pkg_resources for automated testing purposes, you don't need this when using the function.

```
>>> from pkg_resources import resource_filename
>>> filepath = resource_filename(__name__, 'data/data.csv')
```

So, assuming your file lives at 'filepath', you open it as such:

```
>>> get_data(filepath)
array([530., 518., 506., ..., 492., 493., 494.])
```

Files with multiple columns can be opened by specifying the 'column_name' where the data resides:

```
>>> filepath = resource_filename(__name__, 'data/data2.csv')
```

Again you don't need the above. It is there for automated testing.

```
>>> get_data(filepath, column_name='timer')
array([0.00000000e+00, 8.54790319e+00, 1.70958064e+01, ...,
       1.28192904e+05, 1.28201452e+05, 1.28210000e+05])
```

You can open matlab files in much the same way by specifying the column where the data lives:

```
>>> filepath = resource_filename(__name__, 'data/data2.mat')
```

Again you don't need the above. It is there for automated testing. Open matlab file by specifying the column name as well:

```
>>> get_data(filepath, column_name='hr')
array([515., 514., 514., ..., 492., 494., 496.])
```

You can any csv formatted text file no matter the extension if you set ignore_extension to True:

```
>>> filepath = resource_filename(__name__, 'data/data.log')
>>> get_data(filepath, ignore_extension = True)
array([530., 518., 506., ..., 492., 493., 494.])
```

You can specify column names in the same way when using ignore_extension

```
>>> filepath = resource_filename(__name__, 'data/data2.log')
>>> data = get_data(filepath, column_name = 'hr', ignore_extension = True)
```

---

`heartpy.datautils.`**`get_samplerate_mstimer`**(*timerdata*)

detemine sample rate based on ms timer

Function to determine sample rate of data from ms-based timer list or array.

**Parameters** **`timerdata`** (*`1d numpy array or list`*) – sequence containing values of a timer, in ms

**Returns** **out** – the sample rate as determined from the timer sequence provided

**Return type** float

---

**Examples**

first we load a provided example dataset

```
>>> data, timer = load_exampledata(example = 1)
```

since it's a timer that counts miliseconds, we use this function. Let's also round to three decimals

```
>>> round(get_samplerate_mstimer(timer), 3)
116.996
```

of course if another time unit is used, converting it to ms-based should be trivial.

---

`heartpy.datautils.`**`get_samplerate_datetime`**(*datetimedata*, *timeformat='%H:%M:%S.%f'*)

determine sample rate based on datetime

Function to determine sample rate of data from datetime-based timer list or array.

**Parameters**

- **`timerdata`** (*`1-d numpy array or list`*) – sequence containing datetime strings
- **`timeformat`** (*`string`*) – the format of the datetime-strings in datetimedata default : '%H:%M:%S.f' (24-hour based time including ms: e.g. 21:43:12.569)

**Returns** **out** – the sample rate as determined from the timer sequence provided

**Return type** float

---

**Examples**

We load the data like before

```
>>> data, timer = load_exampledata(example = 2)
>>> timer[0]
'2016-11-24 13:58:58.081000'
```

Note that we need to specify the timeformat used so that datetime understands what it's working with:

```
>>> round(get_samplerate_datetime(timer, timeformat = '%Y-%m-%d %H:%M:%S.%f'), 3)
100.42
```

heartpy.datautils.**rolling_mean**(*data*, *windowsize*, *sample_rate*)
calculates rolling mean

Function to calculate the rolling mean (also: moving average) over the passed data.

**Parameters**

- **data** (*1-dimensional numpy array or list*) – sequence containing data over which rolling mean is to be computed
- **windowsize** (*int or float*) – the window size to use, in seconds calculated as windowsize * sample_rate
- **sample_rate** (*int or float*) – the sample rate of the data set

**Returns** **out** – sequence containing computed rolling mean

**Return type** 1-d numpy array

**Examples**

```
>>> data, _ = load_exampledata(example = 1)
>>> rmean = rolling_mean(data, windowsize=0.75, sample_rate=100)
>>> rmean[100:110]
array([514.49333333, 514.49333333, 514.49333333, 514.46666667,
       514.45333333, 514.45333333, 514.45333333, 514.45333333,
       514.48      , 514.52      ])
```

heartpy.datautils.**outliers_iqr_method**(*hrvalues*)
removes outliers

Function that removes outliers based on the interquartile range method and substitutes them for the median see: https://en.wikipedia.org/wiki/Interquartile_range

**Parameters** **hrvalues** (*1-d numpy array or list*) – sequence of values, from which outliers need to be identified

**Returns** **out** – [0] cleaned sequence with identified outliers substituted for the median [1] list of indices that have been replaced in the original array or list

**Return type** tuple

**Examples**

```
>>> x = [2, 4, 3, 4, 6, 7, 35, 2, 3, 4]
>>> outliers_iqr_method(x)
([2, 4, 3, 4, 6, 7, 4.0, 2, 3, 4], [6])
```

heartpy.datautils.**outliers_modified_z**(*hrvalues*)
removes outliers

Function that removes outliers based on the modified Z-score metric and substitutes them for the median

**Parameters** **hrvalues** (*1-d numpy array or list*) – sequence of values, from which outliers need to be identified

**Returns** **out** – [0] cleaned sequence with identified outliers substituted for the median [1] list of indices that have been replaced in the original array or list

**Return type** tuple

**Examples**

```
>>> x = [2, 4, 3, 4, 6, 7, 35, 2, 3, 4]
>>> outliers_modified_z(x)
([2, 4, 3, 4, 6, 7, 4.0, 2, 3, 4], [6])
```

heartpy.datautils.**MAD**(*data*)

computes median absolute deviation

Function that compute median absolute deviation of data slice See: https://en.wikipedia.org/wiki/Median_absolute_deviation

**Parameters** **data** (*1-dimensional numpy array or list*) – sequence containing data over which to compute the MAD

**Returns** **out** – the Median Absolute Deviation as computed

**Return type** float

**Examples**

```
>>> x = [2, 4, 3, 4, 6, 7, 35, 2, 3, 4]
>>> MAD(x)
1.5
```

heartpy.datautils.**load_exampledata**(*example=0*)

loads example data

Function to load one of the example datasets included in HeartPy and used in the documentation.

**Parameters** **example** (*int (0, 1, 2)*) – selects example data used in docs of three datafiles. Available (see github repo for source of files): 0 : data.csv 1 : data2.csv 2 : data3.csv default : 0

**Returns** **out** – Contains the data and timer column. If no timer data is available, such as in example 0, an empty second array is returned.

**Return type** tuple of two arrays

**Examples**

This function can load one of the three example data files provided with HeartPy. It returns both the data and a timer if that is present

For example:

```
>>> data, _ = load_exampledata(0)
>>> data[0:5]
array([530., 518., 506., 494., 483.])
```

And another example:

```
>>> data, timer = load_exampledata(1)
>>> [round(x, 2) for x in timer[0:5]]
[0.0, 8.55, 17.1, 25.64, 34.19]
```

### 3.2.4 exceptions

Custom exceptions and warnings for HeartPy

**exception** `heartpy.exceptions.`**`BadSignalWarning`**

Bases: `exceptions.UserWarning`

warning class to raise when no heart rate is detectable in supplied signal.

This warning notifies the user that the supplied signal is of insufficient quality and/or does not contain enough information to properly process.

### 3.2.5 Filtering

Functions for data filtering tasks.

`heartpy.filtering.`**`filter_signal`**(*data*, *cutoff*, *sample_rate*, *order=2*, *filtertype='lowpass'*, *return_top=False*)

Apply the specified filter

Function that applies the specified lowpass, highpass or bandpass filter to the provided dataset.

**Parameters**

- **data** (*1-dimensional numpy array or list*) – Sequence containing the to be filtered data
- **cutoff** (*int, float or tuple*) – the cutoff frequency of the filter. Expects float for low and high types and for bandpass filter expects list or array of format [lower_bound, higher_bound]
- **sample_rate** (*int or float*) – the sample rate with which the passed data sequence was sampled
- **order** (*int*) – the filter order default : 2
- **filtertype** (*str*) – The type of filter to use. Available: - lowpass : a lowpass butterworth filter - highpass : a highpass butterworth filter - bandpass : a bandpass butterworth filter - notch : a notch filter around specified frequency range both the highpass and notch filter are useful for removing baseline wander. The notch filter is especially useful for removing baseling wander in ECG signals.

**Returns** **out** – 1d array containing the filtered data

**Return type** 1d array

**Examples**

```
>>> import numpy as np
>>> import heartpy as hp
```

Using standard data provided

```
>>> data, _ = hp.load_exampledata(0)
```

We can filter the signal, for example with a lowpass cutting out all frequencies of 5Hz and greater (with a sloping frequency cutoff)

```
>>> filtered = filter_signal(data, cutoff = 5, sample_rate = 100.0, order = 3,␣
↪filtertype='lowpass')
>>> print(np.around(filtered[0:6], 3))
[530.175 517.893 505.768 494.002 482.789 472.315]
```

Or we can cut out all frequencies below 0.75Hz with a highpass filter:

```
>>> filtered = filter_signal(data, cutoff = 0.75, sample_rate = 100.0, order = 3,␣
↪filtertype='highpass')
>>> print(np.around(filtered[0:6], 3))
[-17.975 -28.271 -38.609 -48.992 -58.422 -67.902]
```

Or specify a range (here: 0.75 - 3.5Hz), outside of which all frequencies are cut out.

```
>>> filtered = filter_signal(data, cutoff = [0.75, 3.5], sample_rate = 100.0,
... order = 3, filtertype='bandpass')
>>> print(np.around(filtered[0:6], 3))
[-12.012 -23.159 -34.261 -45.12  -55.541 -65.336]
```

A 'Notch' filtertype is also available (see remove_baseline_wander).

```
>>> filtered = filter_signal(data, cutoff = 0.05, sample_rate = 100.0, filtertype=
↪'notch')
```

Finally we can use the return_top flag to only return the filter response that has amplitute above zero. We're only interested in the peaks, and sometimes this can improve peak prediction:

```
>>> filtered = filter_signal(data, cutoff = [0.75, 3.5], sample_rate = 100.0,
... order = 3, filtertype='bandpass', return_top = True)
>>> print(np.around(filtered[48:53], 3))
[ 0.      0.      0.409 17.088 35.673]
```

---

`heartpy.filtering.`**`hampel_filter`**(*data*, *filtsize=6*)

Detect outliers based on hampel filter

Funcion that detects outliers based on a hampel filter. The filter takes datapoint and six surrounding samples. Detect outliers based on being more than 3std from window mean. See: https://www.mathworks.com/help/signal/ref/hampel.html

**Parameters**

- **data** (*1d list or array*) – list or array containing the data to be filtered
- **filtsize** (*int*) – the filter size expressed the number of datapoints taken surrounding the analysed datapoint. a filtsize of 6 means three datapoints on each side are taken. total filtersize is thus filtsize + 1 (datapoint evaluated)

**Returns** out

**Return type** array containing filtered data

---

**Examples**

```
>>> from .datautils import get_data, load_exampledata
>>> data, _ = load_exampledata(0)
>>> filtered = hampel_filter(data, filtsize = 6)
>>> print('%i, %i' %(data[1232], filtered[1232]))
497, 496
```

---

heartpy.filtering.**hampel_correcter**(*data*, *sample_rate*)

apply altered version of hampel filter to suppress noise.

Function that returns te difference between data and 1-second windowed hampel median filter. Results in strong noise suppression characteristics, but relatively expensive to compute.

Result on output measures is present but generally not large. However, use sparingly, and only when other means have been exhausted.

**Parameters**

- **data** (*1d numpy array*) – array containing the data to be filtered
- **sample_rate** (*int or float*) – sample rate with which data was recorded

**Returns** **out** – array containing filtered data

**Return type** 1d numpy array

---

**Examples**

```
>>> from .datautils import get_data, load_exampledata
>>> data, _ = load_exampledata(1)
>>> filtered = hampel_correcter(data, sample_rate = 116.995)
```

---

heartpy.filtering.**smooth_signal**(*data*, *sample_rate*, *window_length=None*, *polyorder=3*)

smooths given signal using savitzky-golay filter

Function that smooths data using savitzky-golay filter using default settings.

Functionality requested by Eirik Svendsen. Added since 1.2.4

**Parameters**

- **data** (*1d array or list*) – array or list containing the data to be filtered
- **sample_rate** (*int or float*) – the sample rate with which data is sampled
- **window_length** (*int or None*) – window length parameter for savitzky-golay filter, see Scipy.signal.savgol_filter docs. Must be odd, if an even int is given, one will be added to make it uneven. default : 0.1 * sample_rate
- **polyorder** (*int*) – the order of the polynomial fitted to the signal. See scipy.signal.savgol_filter docs. default : 3

**Returns** **smoothed** – array containing the smoothed data

**Return type** 1d array

---

**Examples**

Given a fictional signal, a smoothed signal can be obtained by smooth_signal():

```
>>> x = [1, 3, 4, 5, 6, 7, 5, 3, 1, 1]
>>> smoothed = smooth_signal(x, sample_rate = 2, window_length=4, polyorder=2)
>>> np.around(smoothed[0:4], 3)
array([1.114, 2.743, 4.086, 5.   ])
```

If you don't specify the window_length, it is computed to be 10% of the sample rate (+1 if needed to make odd) >>> import heartpy as hp >>> data, timer = hp.load_exampledata(0) >>> smoothed = smooth_signal(data, sample_rate = 100)

### 3.2.6 Peakdetection

functions for peak detection and related tasks

`heartpy.peakdetection.`**`make_windows`**(*data*, *sample_rate*, *windowsize=120*, *overlap=0*, *min_size=20*)

slices data into windows

Funcion that slices data into windows for concurrent analysis. Used by process_segmentwise wrapper function.

**Parameters**

- **`data`** (*`1-d array`*) – array containing heart rate sensor data
- **`sample_rate`** (*`int or float`*) – sample rate of the data stream in 'data'
- **`windowsize`** (*`int`*) – size of the window that is sliced in seconds
- **`overlap`** (*`float`*) – fraction of overlap between two adjacent windows: 0 <= float < 1.0
- **`min_size`** (*`int`*) – the minimum size for the last (partial) window to be included. Very short windows might not stable for peak fitting, especially when significant noise is present. Slightly longer windows are likely stable but don't make much sense from a signal analysis perspective.

**Returns** **out** – tuples of window indices

**Return type** array

**Examples**

Assuming a given example data file:

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(1)
```

We can split the data into windows:

```
>>> indices = make_windows(data, 100.0, windowsize = 30, overlap = 0.5, min_size␣
↪= 20)
>>> indices.shape
(9, 2)
```

Specifying min_size = -1 will include the last window no matter what:

```
>>> indices = make_windows(data, 100.0, windowsize = 30, overlap = 0.5, min_size␣
↪= -1)
```

`heartpy.peakdetection.`**`append_dict`**(*dict_obj*, *measure_key*, *measure_value*)

appends data to keyed dict.

Function that appends key to continuous dict, creates if doesn't exist. EAFP

**Parameters**

- **`dict_obj`** (*`dict`*) – dictionary object that contains continuous output measures
- **`measure_key`** (*`str`*) – key for the measure to be stored in continuous_dict
- **`measure_value`** (*`any data container`*) – value to be appended to dictionary

**Returns** **dict_obj** – dictionary object passed to function, with specified data container appended

**Return type** dict

---

**Examples**

Given a dict object 'example' with some data in it:

```
>>> example = {}
>>> example['call'] = ['hello']
```

We can use the function to append it:

```
>>> example = append_dict(example, 'call', 'world')
>>> example['call']
['hello', 'world']
```

A new key will be created if it doesn't exist:

```
>>> example = append_dict(example, 'different_key', 'hello there!')
>>> sorted(example.keys())
['call', 'different_key']
```

---

`heartpy.peakdetection.`**`detect_peaks`**(*hrdata*, *rol_mean*, *ma_perc*, *sample_rate*, *update_dict=True*, *working_data={}*)

detect peaks in signal

Function that detects heartrate peaks in the given dataset.

**Parameters**

- **`data`** (*`hr`*) – array or list containing the heart rate data
- **`rol_mean`** (*`1-d numpy array`*) – array containing the rolling mean of the heart rate signal
- **`ma_perc`** (*`int or float`*) – the percentage with which to raise the rolling mean, used for fitting detection solutions to data
- **`sample_rate`** (*`int or float`*) – the sample rate of the provided data set
- **`update_dict`** (*`bool`*) – whether to update the peak information in the module's data structure Settable to False to allow this function to be re-used for example by the breath analysis module. default : True

---

**Examples**

Normally part of the peak detection pipeline. Given the first example data it would work like this:

```
>>> import heartpy as hp
>>> from heartpy.datautils import rolling_mean, _sliding_window
>>> data, _ = hp.load_exampledata(0)
>>> rol_mean = rolling_mean(data, windowsize = 0.75, sample_rate = 100.0)
>>> wd = detect_peaks(data, rol_mean, ma_perc = 20, sample_rate = 100.0)
```

Now the peaklist has been appended to the working data dict. Let's look at the first five peak positions:

```
>>> wd['peaklist'][0:5]
[63, 165, 264, 360, 460]
```

---

heartpy.peakdetection.**fit_peaks**(*hrdata*, *rol_mean*, *sample_rate*, *bpmmin=40*, *bpmmax=180*, *working_data={}*)

optimize for best peak detection

Function that runs fitting with varying peak detection thresholds given a heart rate signal.

**Parameters**

- **hrdata** (*1d array or list*) – array or list containing the heart rate data
- **rol_mean** (*1-d array*) – array containing the rolling mean of the heart rate signal
- **sample_rate** (*int or float*) – the sample rate of the data set
- **bpmmin** (*int*) – minimum value of bpm to see as likely default : 40
- **bpmmax** (*int*) – maximum value of bpm to see as likely default : 180

**Returns** **working_data** – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Return type** dict

---

**Examples**

Part of peak detection pipeline. Uses moving average as a peak detection threshold and rises it stepwise. Determines best fit by minimising standard deviation of peak-peak distances as well as getting a bpm that lies within the expected range.

Given included example data let's show how this works

```
>>> import heartpy as hp
>>> from heartpy.datautils import rolling_mean, _sliding_window
>>> data, _ = hp.load_exampledata(0)
>>> rol_mean = rolling_mean(data, windowsize = 0.75, sample_rate = 100.0)
```

We can then call this function and let the optimizer do its work:

```
>>> wd = fit_peaks(data, rol_mean, sample_rate = 100.0)
```

Now the wd dict contains the best fit paramater(s):

```
>>> wd['best']
20
```

This indicates the best fit can be obtained by raising the moving average with 20%.

The results of the peak detection using these parameters are included too. To illustrate, these are the first five detected peaks:

```
>>> wd['peaklist'][0:5]
[63, 165, 264, 360, 460]
```

and the corresponding peak-peak intervals:

```
>>> wd['RR_list'][0:4]
array([1020.,  990.,  960., 1000.])
```

---

`heartpy.peakdetection.`**`check_peaks`**(*rr_arr*, *peaklist*, *ybeat*, *quotient_filter=False*, *reject_segmentwise=False*, *working_data={}*)

find anomalous peaks.

Funcion that checks peaks for outliers based on anomalous peak-peak distances and corrects by excluding them from further analysis.

**Parameters**

- **`rr_arr`** (*`1d array or list`*) – list or array containing peak-peak intervals
- **`peaklist`** (*`1d array or list`*) – list or array containing detected peak positions
- **`ybeat`** (*`1d array or list`*) – list or array containing corresponding signal values at detected peak positions. Used for plotting functionality later on.
- **`reject_segmentwise`** (*`bool`*) – if set, checks segments per 10 detected peaks. Marks segment as rejected if 30% of peaks are rejected. default : False
- **`working_data`** (*`dict`*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Returns** **working_data** – working_data dictionary object containing all of heartpy's temp objects

**Return type** dict

---

**Examples**

Part of peak detection pipeline. No standalone examples exist. See docstring for hp.process() function for more info

---

`heartpy.peakdetection.`**`check_binary_quality`**(*peaklist*, *binary_peaklist*, *maxrejects=3*, *working_data={}*)

checks signal in chunks of 10 beats.

Function that checks signal in chunks of 10 beats. It zeros out chunk if number of rejected peaks > maxrejects. Also marks rejected segment coordinates in tuples (x[0], x[1] in working_data['rejected_segments']

**Parameters**

- **`peaklist`** (*`1d array or list`*) – list or array containing detected peak positions
- **`binary_peaklist`** (*`1d array or list`*) – list or array containing mask for peaklist, coding which peaks are rejected
- **`maxjerects`** (*`int`*) – maximum number of rejected peaks per 10-beat window default : 3
- **`working_data`** (*`dict`*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

**Returns** **working_data** – working_data dictionary object containing all of heartpy's temp objects

**Return type** dict

**Examples**

Part of peak detection pipeline. No standalone examples exist. See docstring for hp.process() function for more info

Given some peaklist and binary mask: >>> peaklist = [30, 60, 90, 110, 130, 140, 160, 170, 200, 220] >>> binary_peaklist = [0, 1, 1, 0, 0, 1, 0, 1, 0, 0] >>> wd = check_binary_quality(peaklist, binary_peaklist) >>> wd['rejected_segments'] [(30, 220)]

The whole segment is rejected as it contains more than the specified 3 rejections per 10 beats.

`heartpy.peakdetection.`**`interpolate_peaks`**(*data*, *peaks*, *sample_rate*, *desired_sample_rate=1000.0*, *working_data={}*)

interpolate detected peak positions and surrounding data points

Function that enables high-precision mode by taking the estimated peak position, then upsampling the peak position +/- 100ms to the specified sampling rate, subsequently estimating the peak position with higher accuracy.

**Parameters**

- **data** (*1d list or array*) – list or array containing heart rate data
- **peaks** (*1d list or array*) – list or array containing x-positions of peaks in signal
- **sample_rate** (*int or float*) – the sample rate of the signal (in Hz)
- **desired_sampled-rate** (*int or float*) – the sample rate to which to upsample. Must be sample_rate < desired_sample_rate

**Returns** **working_data** – working_data dictionary object containing all of heartpy's temp objects

**Return type** dict

**Examples**

Given the output of a normal analysis and the first five peak-peak intervals:

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(0)
>>> wd, m = hp.process(data, 100.0)
>>> wd['peaklist'][0:5]
[63, 165, 264, 360, 460]
```

Now, the resolution is at max 10ms as that's the distance between data points. We can use the high precision mode for example to approximate a more precise position, for example if we had recorded at 1000Hz:

```
>>> wd = interpolate_peaks(data = data, peaks = wd['peaklist'],
... sample_rate = 100.0, desired_sample_rate = 1000.0, working_data = wd)
>>> wd['peaklist'][0:5]
[63.5, 165.4, 263.6, 360.4, 460.2]
```

As you can see the accuracy of peak positions has increased. Note that you cannot magically upsample nothing into something. Be reasonable.

### 3.2.7 Preprocessing

`heartpy.preprocessing.`**`scale_data`**(*data*, *lower=0*, *upper=1024*)

scales passed sequence between thresholds

Function that scales passed data so that it has specified lower and upper bounds.

**Parameters**

- **data** (*1-d array or list*) – Sequence to be scaled
- **lower** (*int or float*) – lower threshold for scaling default : 0
- **upper** (*int or float*) – upper threshold for scaling default : 1024

**Returns** **out** – contains scaled data

**Return type** 1-d array

---

**Examples**

When passing data without further arguments to the function means it scales 0-1024

```
>>> x = [2, 3, 4, 5]
>>> scale_data(x)
array([   0.        ,  341.33333333,  682.66666667, 1024.        ])
```

Or you can specify a range:

```
>>> scale_data(x, lower = 50, upper = 124)
array([ 50.        ,  74.66666667,  99.33333333, 124.        ])
```

---

heartpy.preprocessing.**scale_sections**(*data*, *sample_rate*, *windowsize=2.5*, *lower=0*, *upper=1024*)

scales data using sliding window approach

Function that scales the data within the defined sliding window between the defined lower and upper bounds.

**Parameters**

- **data** (*1-d array or list*) – Sequence to be scaled
- **sample_rate** (*int or float*) – Sample rate of the passed signal
- **windowsize** (*int or float*) – size of the window within which signal is scaled, in seconds default : 2.5
- **lower** (*int or float*) – lower threshold for scaling. Passed to scale_data. default : 0
- **upper** (*int or float*) – upper threshold for scaling. Passed to scale_data. default : 1024

**Returns** **out** – contains scaled data

**Return type** 1-d array

---

**Examples**

```
>>> x = [20, 30, 20, 30, 70, 80, 20, 30, 20, 30]
>>> scale_sections(x, sample_rate=1, windowsize=2, lower=20, upper=30)
array([20., 30., 20., 30., 20., 30., 20., 30., 20., 30.])
```

---

heartpy.preprocessing.**enhance_peaks**(*hrdata*, *iterations=2*)

enhances peak amplitude relative to rest of signal

Function thta attempts to enhance the signal-noise ratio by accentuating the highest peaks. Note: denoise first

**Parameters**

- **hrdata** (*1-d numpy array or list*) – sequence containing heart rate data
- **iterations** (*int*) – the number of scaling steps to perform default : 2

**Returns** **out** – array containing enhanced peaks

**Return type** 1-d numpy array

**Examples**

Given an array of data, the peaks can be enhanced using the function

```
>>> x = [200, 300, 500, 900, 500, 300, 200]
>>> enhance_peaks(x)
array([   0.        ,    4.31776016,   76.16528926, 1024.        ,
         76.16528926,    4.31776016,    0.        ])
```

heartpy.preprocessing.**enhance_ecg_peaks**(*hrdata*, *sample_rate*, *iterations=4*, *aggregation='mean'*, *notch_filter=True*)

enhances ECG peaks

Function that convolves synthetic QRS templates with the signal, leading to a strong increase signal-to-noise ratio. Function ends with an optional Notch filterstep (default : true) to reduce noise from the iterating convolution steps.

**Parameters**

- **hrdata** (*1-d numpy array or list*) – sequence containing heart rate data
- **sample_rate** (*int or float*) – sample rate with which the data is sampled
- **iterations** (*int*) – how many convolutional iterations should be run. More will result in stronger peak enhancement, but over a certain point (usually between 12-16) overtones start appearing in the signal. Only increase this if the peaks aren't amplified enough. default : 4
- **aggregation** (*str*) – how the data from the different convolutions should be aggregated. Can be either 'mean' or 'median'. default : mean
- **notch_filter** (*bool*) – whether to apply a notch filter after the last convolution to get rid of remaining low frequency noise. default : true

**Returns** **output** – The array containing the filtered data with enhanced peaks

**Return type** 1d array

**Examples**

First let's import the module and load the data

```
>>> import heartpy as hp
>>> data, timer = hp.load_exampledata(1)
>>> sample_rate = hp.get_samplerate_mstimer(timer)
```

After loading the data we call the function like so:

```
>>> filtered_data = enhance_ecg_peaks(data, sample_rate, iterations = 3)
```

By default the module uses the mean to aggregate convolutional outputs. It is also possible to use the median.

```
>>> filtered_data = enhance_ecg_peaks(data, sample_rate, iterations = 3,
... aggregation = 'median', notch_filter = False)
```

In the last example we also disabled the notch filter.

heartpy.preprocessing.**interpolate_clipping**(*data*, *sample_rate*, *threshold=1020*)
interpolate peak waveform

Function that interpolates peaks between the clipping segments using cubic spline interpolation. It takes the clipping start +/- 100ms to calculate the spline.

**Parameters**

- **data** (*1d list or numpy array*) – data section to be evaluated
- **sample_rate** (*int or float*) – sample rate with which the data array is sampled
- **threshold** (*int or float*) – the threshold for clipping, recommended to be a few data points below ADC or sensor max value, to compensate for signal noise default : 1020

**Returns** **out** – the output is an array with clipping segments replaced by interpolated segments

**Return type** array

**Examples**

First let's load some example data:

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(example=2)
>>> x = data[2000:3000]
>>> x[425:445]
array([948, 977, 977, 977, 977, 978, 978, 977, 978, 977, 977, 977, 977,
       914, 820, 722, 627, 536, 460, 394])
```

And interpolate any clipping segments as such:

```
>>> intp = interpolate_clipping(x, sample_rate=117, threshold=970)
>>> intp[425:445]
array([ 972, 1043, 1098, 1138, 1163, 1174, 1173, 1159, 1134, 1098, 1053,
        998,  934,  848,  747,  646,  552,  470,  402,  348])
```

heartpy.preprocessing.**flip_signal**(*data*, *enhancepeaks=False*, *keep_range=True*)
invert signal waveforms.

Function that flips raw signal with negative mV peaks to normal ECG. Required for proper peak finding in case peaks are expressed as negative dips.

**Parameters**

- **data** (*1d list or numpy array*) – data section to be evaluated
- **enhance_peaks** (*bool*) – whether to apply peak accentuation default : False
- **keep_range** (*bool*) – whether to scale the inverted data so that the original range is maintained

**Returns** **out**

**Return type** 1d array

**Examples**

Given an array of data

```
>>> x = [200, 300, 500, 900, 500, 300, 200]
```

We can call the function. If keep_range is False, the signal will be inverted relative to its mean.

```
>>> flip_signal(x, keep_range=False)
array([628.57142857, 528.57142857, 328.57142857, -71.42857143,
       328.57142857, 528.57142857, 628.57142857])
```

However, by specifying keep_range, the inverted signal will be put 'back in place' in its original range.

```
>>> flip_signal(x, keep_range=True)
array([900., 800., 600., 200., 600., 800., 900.])
```

It's also possible to use the enhance_peaks function:

```
>>> flip_signal(x, enhancepeaks=True)
array([1024.        ,  621.75746332,  176.85545623,    0.        ,
        176.85545623,  621.75746332, 1024.        ])
```

## 3.2.8 visualizeutils

Functions that help visualize results

`heartpy.visualizeutils.`**`plotter`**(*working_data*, *measures*, *show=True*, *title='Heart Rate Signal Peak Detection'*, *moving_average=False*)

plots the analysis results.

Function that uses calculated measures and data stored in the working_data{} and measures{} dict objects to visualise the fitted peak detection solution.

**Parameters**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function
- **show** (*bool*) – when False, function will return a plot object rather than display the results. default : True
- **title** (*string*) – title for the plot. default : "Heart Rate Signal Peak Detection"
- **moving_average** (*bool*) – whether to display the moving average on the plot. The moving average is used for peak fitting. default: False

**Returns** **out** – only returned if show == False.

**Return type** matplotlib plot object

**Examples**

First let's load and analyse some data to visualise

```
>>> import heartpy as hp
>>> data, _ = hp.load_exampledata(0)
>>> wd, m = hp.process(data, 100.0)
```

Then we can visualise

```
>>> plot_object = plotter(wd, m, show=False, title='some awesome title')
```

This returns a plot object which can be visualized or saved or appended. See matplotlib API for more information on how to do this.

A matplotlib plotting object is returned. This can be further processed and saved to a file.

---

`heartpy.visualizeutils.`**`segment_plotter`**(*working_data*, *measures*, *title='Heart Rate Signal Peak Detection'*, *figsize=(6, 6)*, *path="*, *start=0*, *end=None*, *step=1*)

plots analysis results

Function that plots the results of segmentwise processing of heart rate signal and writes all results to separate files at the path provided.

**Parameters**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function
- **title** (*str*) – the title used in the plot
- **figsize** (*tuple*) – figsize tuple to be passed to matplotlib
- **path** (*str*) – the path where the files will be stored, folder must exist.
- **start** (*int*) – what segment to start plotting with default : 0
- **end** (*int*) – last segment to plot. Must be smaller than total number of segments default : None, will plot until end
- **step** (*int*) – stepsize used when iterating over plots every step'th segment will be plotted default : 1

**Returns**

**Return type** None

---

**Examples**

This function has no examples. See documentation of heartpy for more info.

---

`heartpy.visualizeutils.`**`plot_poincare`**(*working_data*, *measures*, *show=True*, *title='Poincare plot'*)

visualize poincare plot

function that visualises poincare plot.

**Parameters**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function

- **measures** (*dict*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function
- **show** (*bool*) – whether to show the plot right away, or return a matplotlib object for further manipulation
- **title** (*str*) – the title used in the plot

**Returns** **out** – only returned if show == False.

**Return type** matplotlib plot object

**Examples**

This function has no examples. See documentation of heartpy for more info.

`heartpy.visualizeutils.`**`plot_breathing`**(*working_data*, *measures*, *show=True*)

plots extracted breathing signal and spectrogram

Function that plots the breathing signal extracted from RR-intervals alongside its computed spectrogram representation.

**Parameters**

- **working_data** (*dict*) – dictionary object that contains all heartpy's working data (temp) objects. will be created if not passed to function
- **measures** (*dict*) – dictionary object used by heartpy to store computed measures. Will be created if not passed to function
- **show** (*bool*) – whether to show the plot right away, or return a matplotlib object for further manipulation

**Returns** **out** – only returned if show == False.

**Return type** matplotlib plot object

**Examples**

This function has no examples. See documentation of heartpy for more info.

## 3.3 Heart Rate Analysis

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

### 3.3.1 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.

### 3.3.2 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.

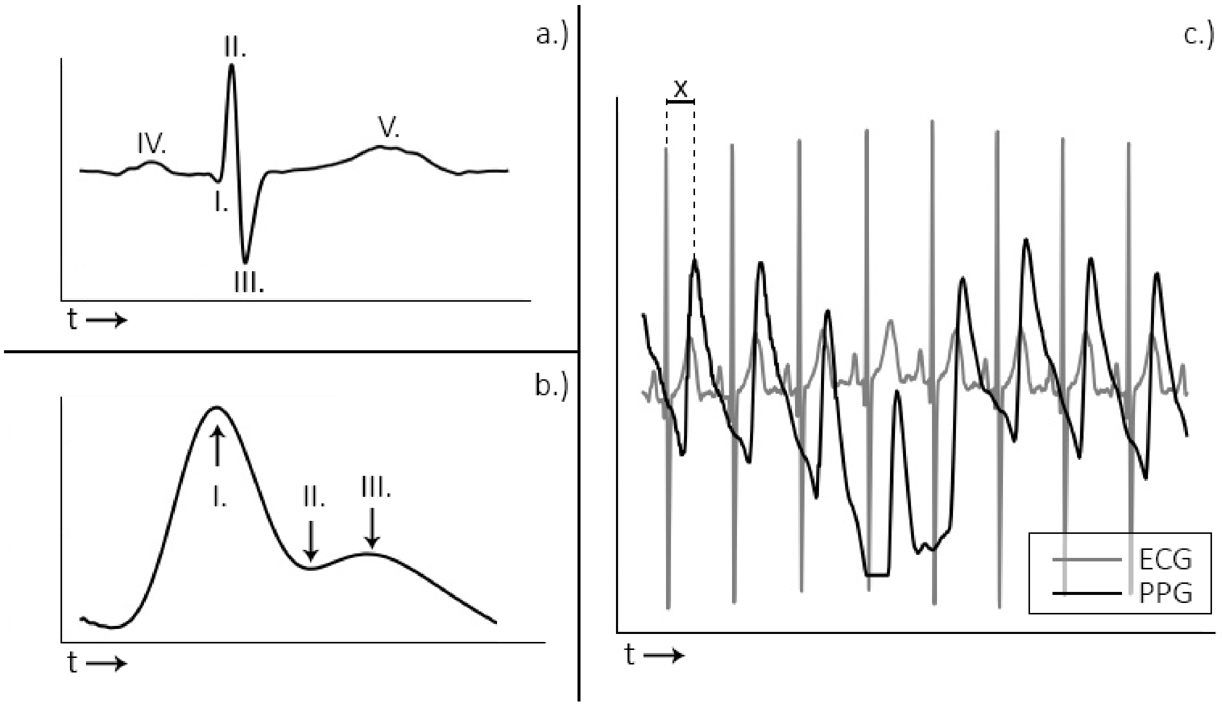


*Figure 1: a. and b. display the ECG and PPG waveform morphology, respectively. The ECG is divided into distinct waves (a, I-V), of which the R-wave (a, II) is used for heart beat extraction. With the PPG wave, the systolic peak (b, I) is used. The plot in c. shows the relationship between ECG and PPG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

[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

data, which is one of their main advantages. Usually the sensors are placed at the fingertip, earlobe, or on the wrist 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.

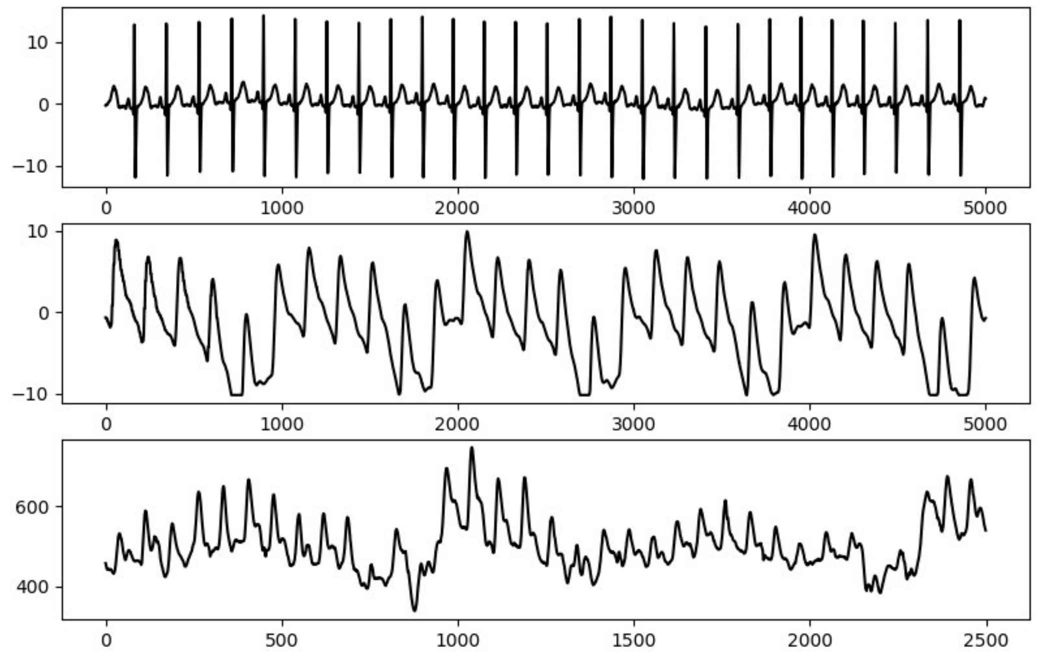


*Figure 2 – The ECG signal (a.) shows a strong QRS complex together with little amplitude variation. The PPG signal measured simultaneously while the patient is at rest in a hospital bed (b.) shows some amplitude variation but relatively stable morphology. When measuring PPG in a driving simulator using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

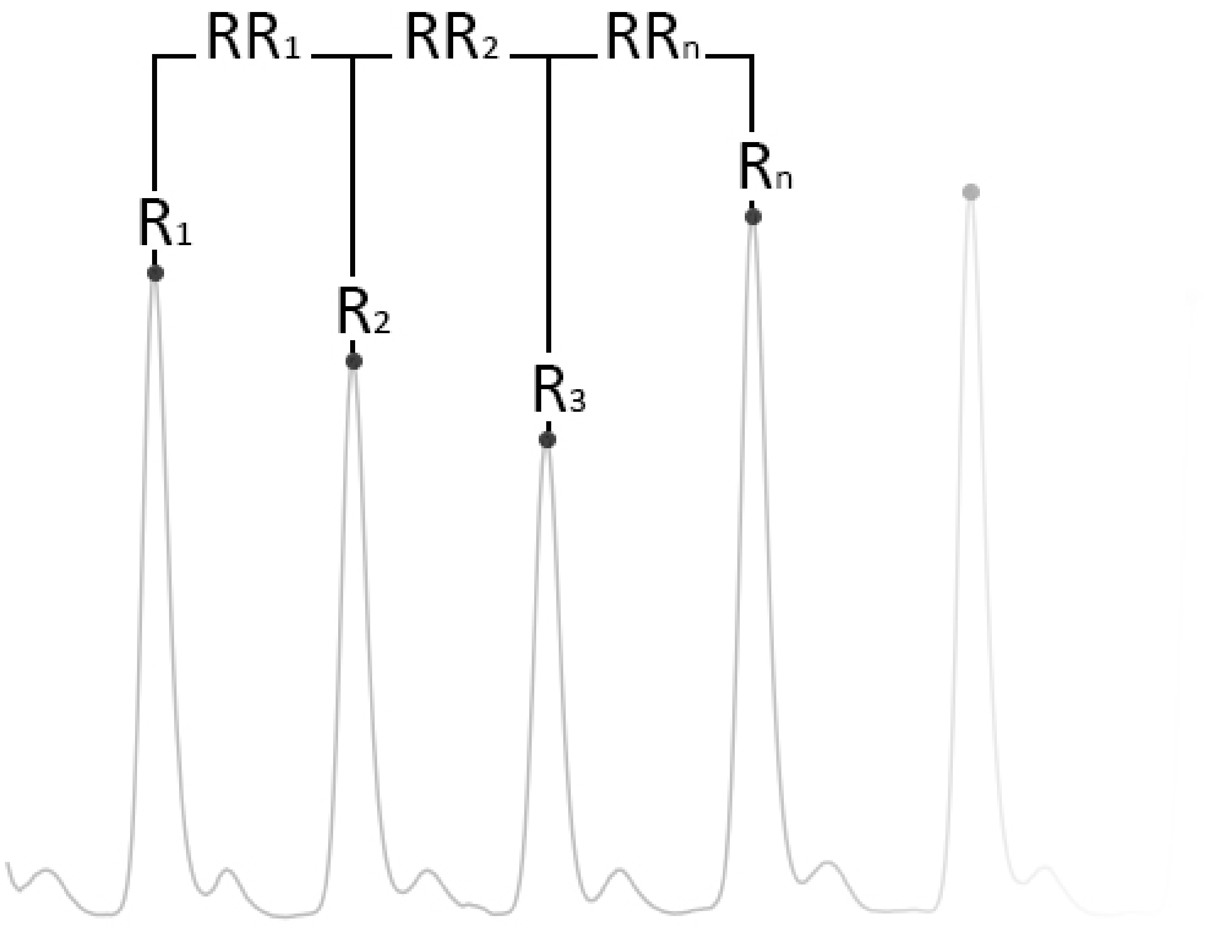
4

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.

### 3.3.3 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.



$$RMSSD = \sqrt{\frac{1}{n-2}\sum_{i=0}^{n-2}(RR_i - RR_{i+1})^2}$$

*n = number of R-peaks used in analysis*

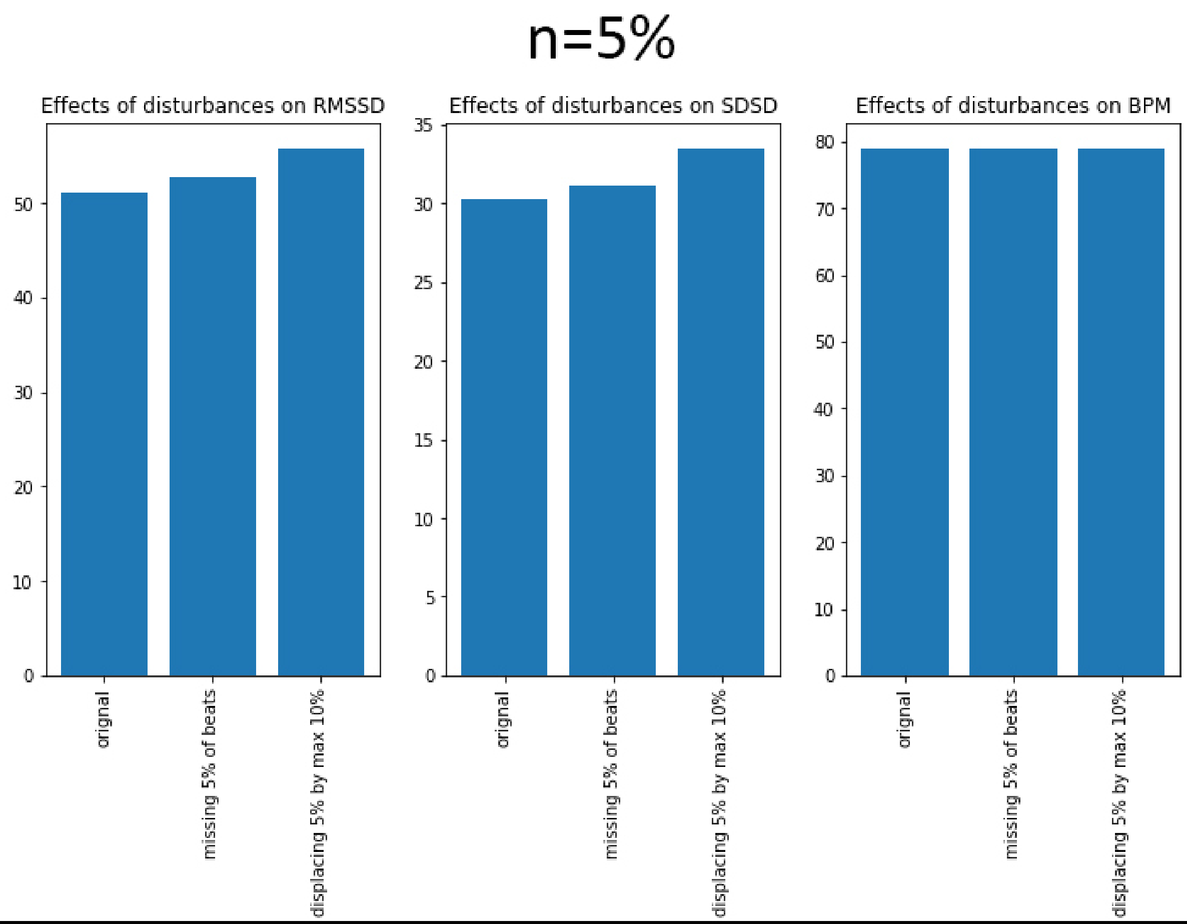
*Figure 3 - Image displaying the desired peak detection result, as well as the calculation of the RMSSD measure. The SDSD measure is the standard deviation between successive differences*

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%

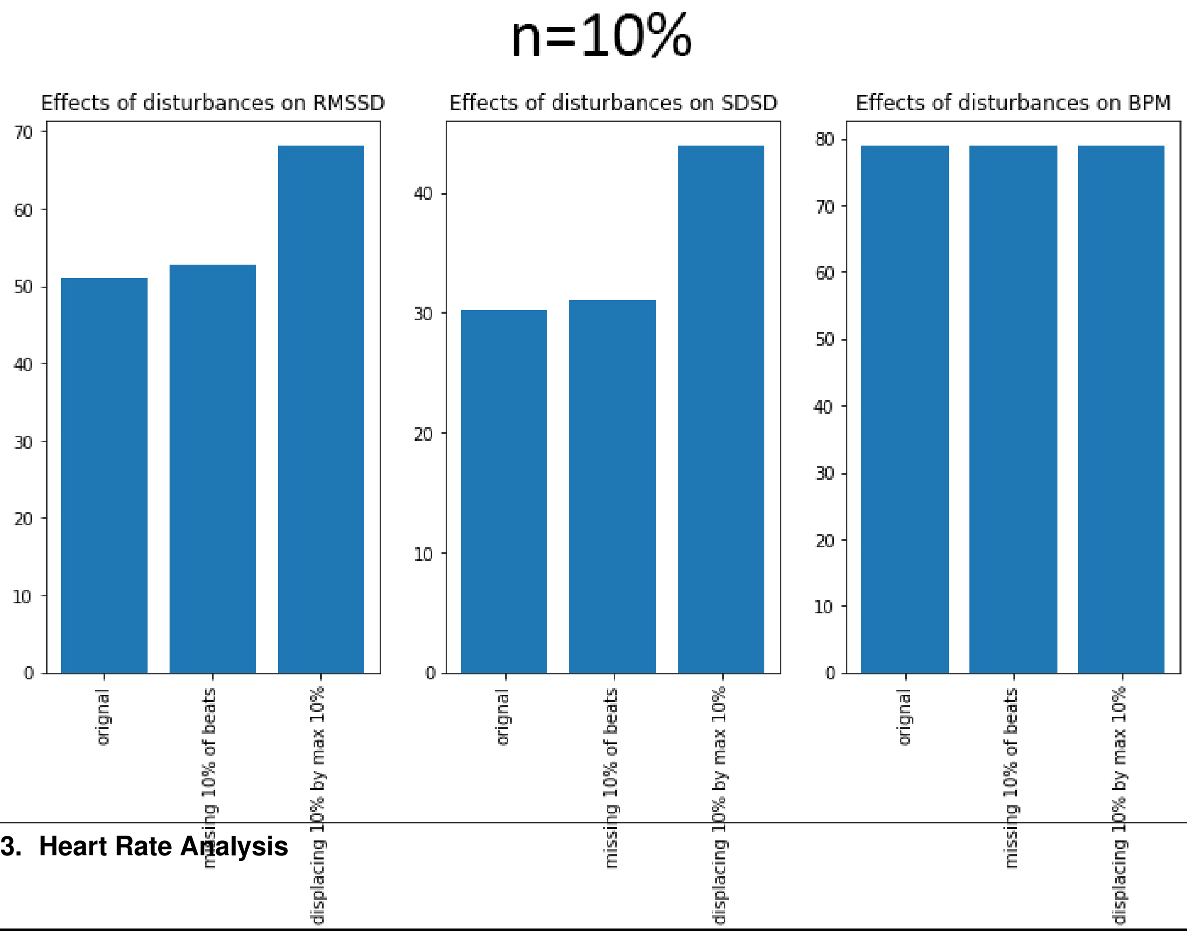

n=10%
Effects of disturbances on RMSSD
Effects of disturbances on SDSD
Effects of disturbances on BPM
orignal
missing 10% of beats
displacing 10% by max 10%

*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 in the technical paper here[6]. Information on the valiation can be found in[5].

### 3.3.4 References

## 3.4 Algorithm functioning

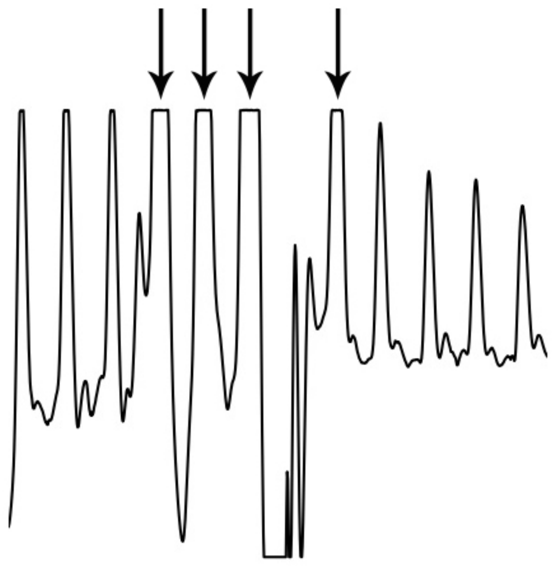
This section describes the details of the algorithm functionality.

### 3.4.1 Pre-processing

Various options are available for pre-processing. These are described below

#### Clipping detection and interpolation

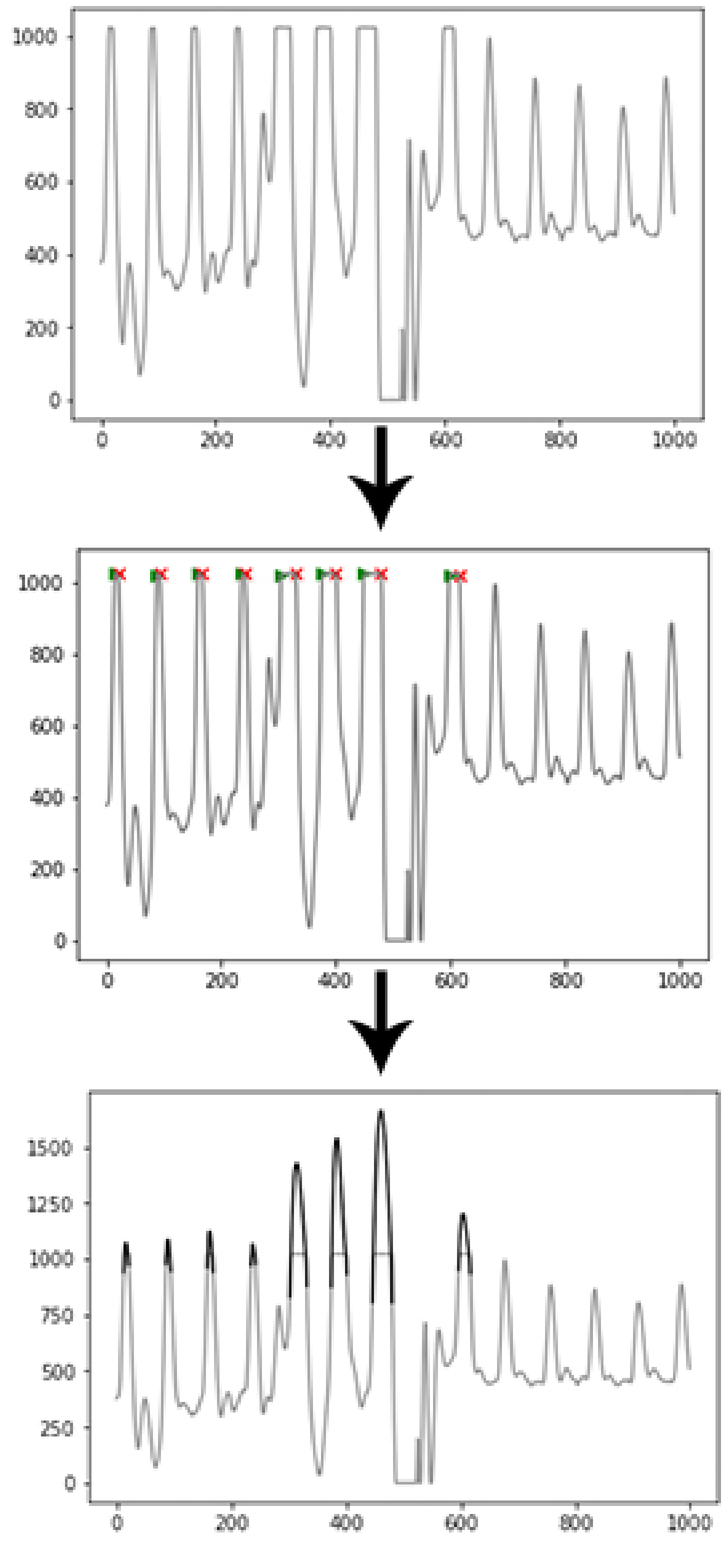
Whenever a measured property exceeds a sensor's sensitivity range, or when digitising an analog signal, clipping can occur. Clipping in this case means the peaks are flattened off because the signal continues outside the boundaries of the sensor you're using:



[6] van Gent, P., Farah, H., van Nes, N., & van Arem, B. (2018). Analysing Noisy Driver Physiology Real-Time Using Off-the-Shelf Sensors: Heart rate analysis software from the Taking the Fast Lane Project. http://doi.org/10.13140/RG.2.2.24895.56485

[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

Clipping functions by detecting (almost) flat parts of the signal near its maximum, preceded and followed by a steep angle on both ends. The ‘missing’ signal peak is interpolated using a cubic spline, which takes into account 100ms of data on both ends of the clipping portion of the signal. The reconstructed R-peak is overlaid on the original signal and used for further analysis.

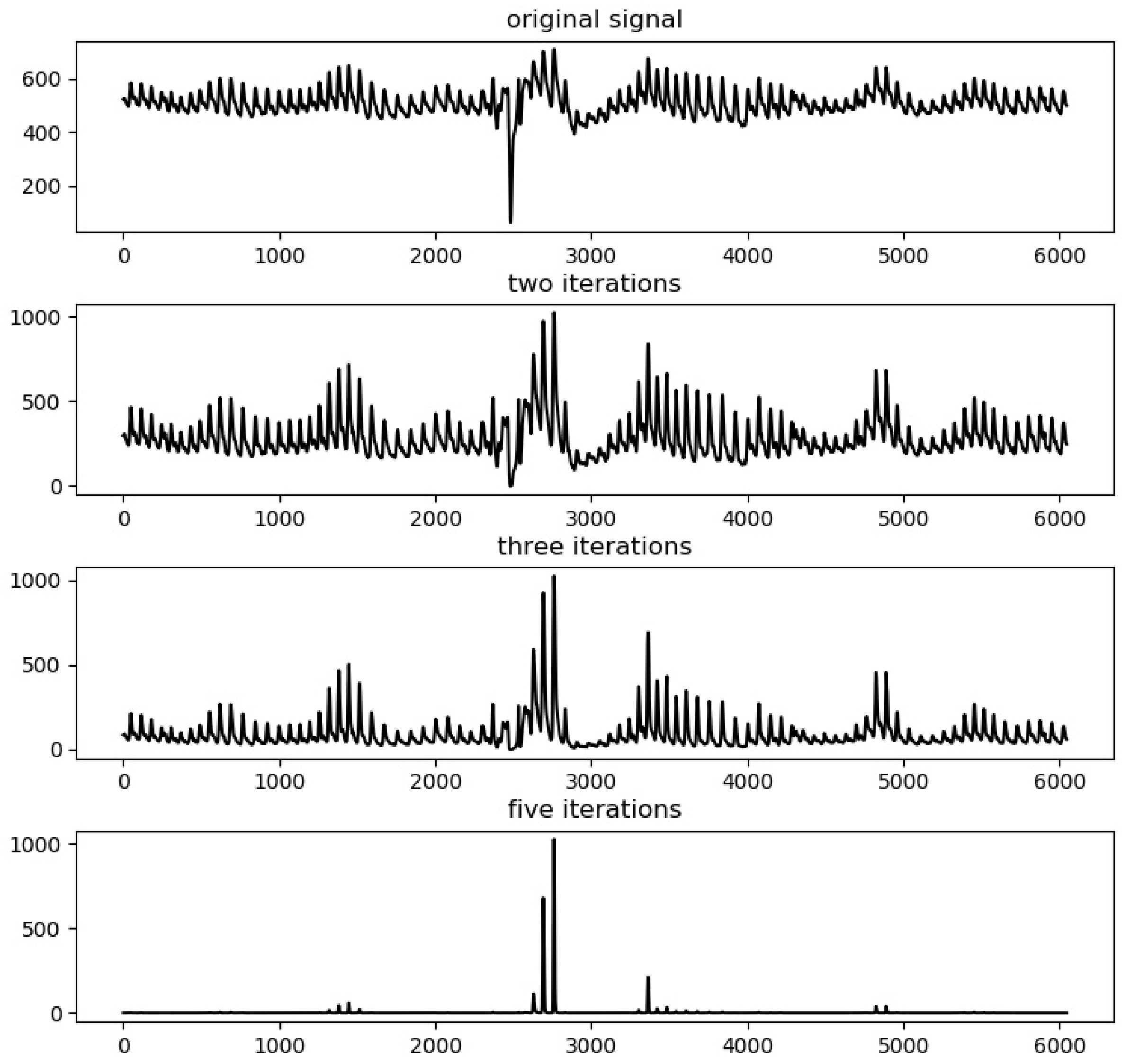

1000
800
600
400
200
0
0
200
400
600
800
1000
1500
1250
1000
750
500
250
0

### Peak enhancement

A peak enhancement function is available that attempts to normalise the amplitude, then increase R-peak amplitude relative to the rest of the signal. It only uses linear transformations, meaning absolute peak positions are not disturbed (in contrast to FIR filters). It runs a predefined number of iterations. Generally two iterations are sufficient. Be cautious not to over-iterate as this will start to suppress peaks of interest as well.

```python
import heartpy as hp

enhanced = hp.enhance_peaks(data, iterations=2)
```

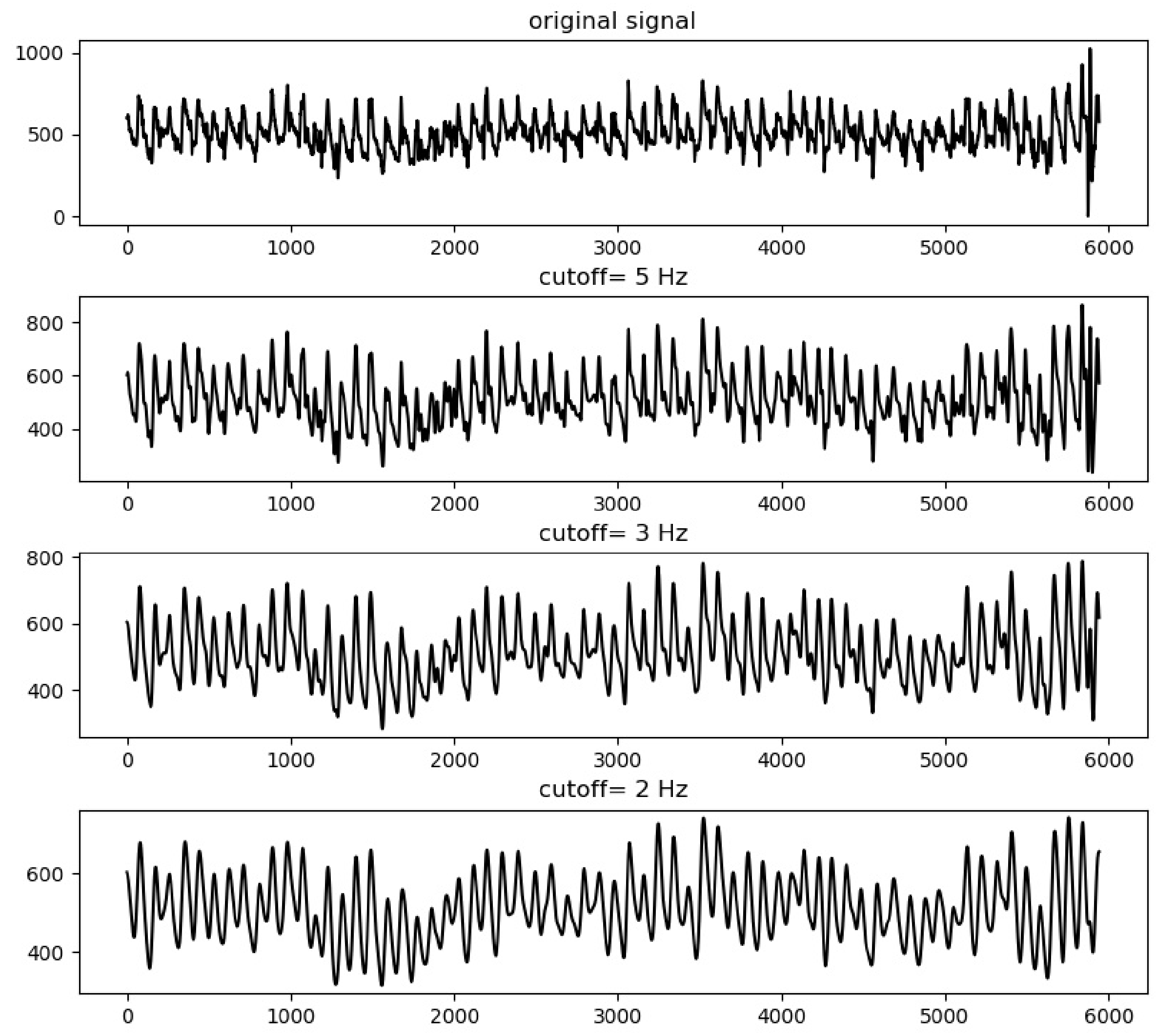


### Butterworth filter

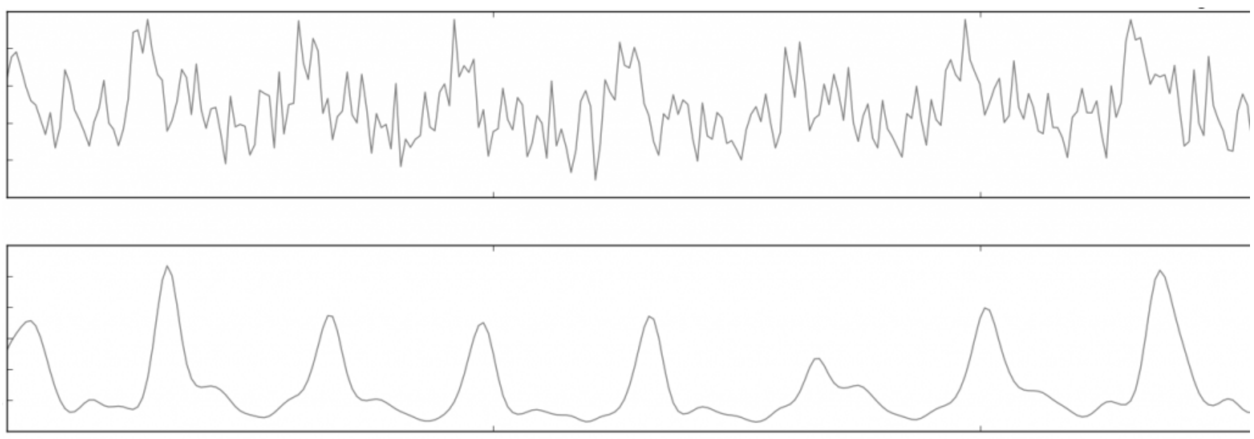
A Butterworth filter implementation is available to remove high frequency noise. Note that this will disturb the absolute peak positions slightly, influencing the output measures. However, in cases of heavy noise this is the only way useful information can be extracted from the signal.

```python
import heartpy as hp

filtered = hp.butter_lowpass_filter(data, cutoff=5, sample_rate=100.0, order=3)
```



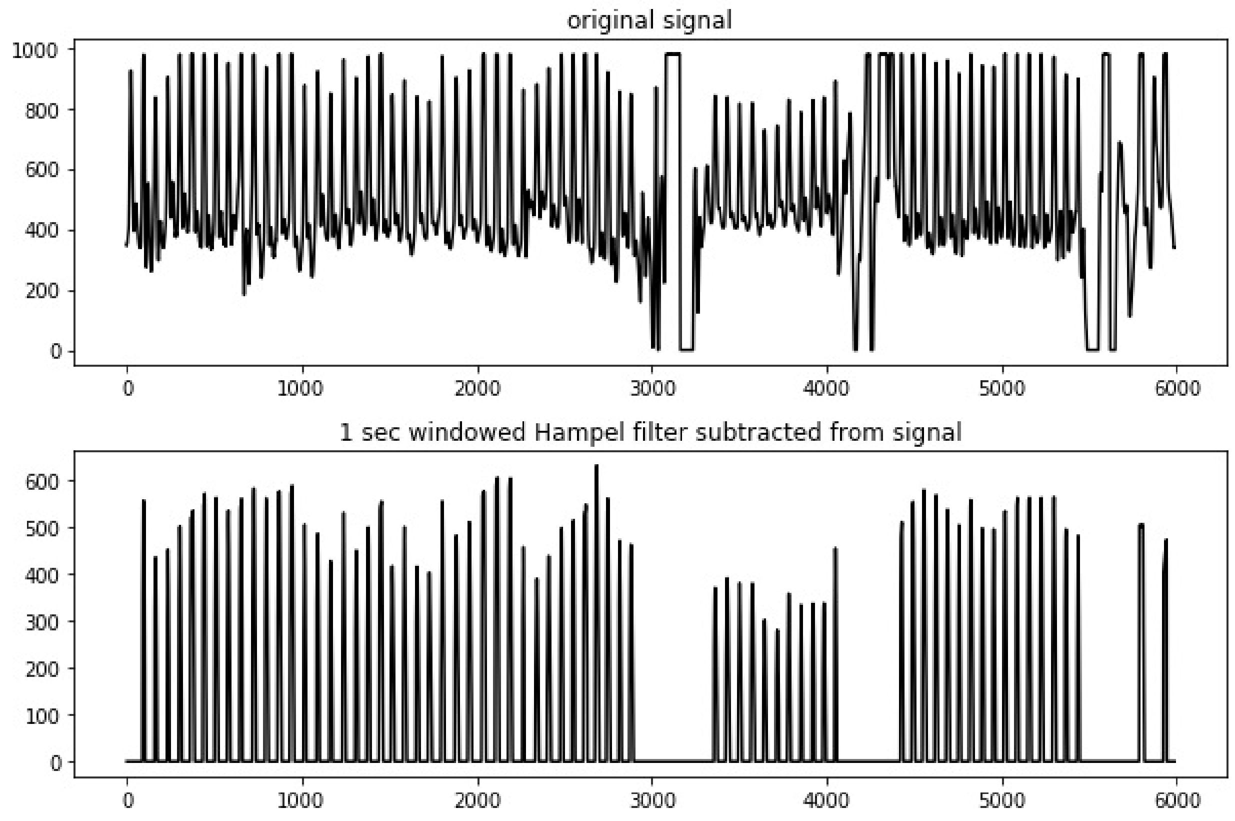
Filtering is generally not recommended unless there is high noise present in the signal. An extreme example is displayed below:



### Hampel Correction

The Hampel Correction functions as an extended version of a Hampel Filter, with a larger window size than the standard datapoint + 3 datapoints on each side (=7). The downside is that it (the current implementation at least) takes significant processing time since a window median and median absolute deviation needs to be computed for each datapoint.

In the current implementation, if called, a median filter is taken over a 1-sec window of the heart rate signal. The filter output is subsequently subtracted from the original signal. When doing so, the property of noise suppression arises:



Note that the absolute peak positions will shift slightly when using this type of filter. With it, the output measures will start to deviate as error is induced. Generally the error is not high, but by default hampel filtering is disabled. It should only be used when encountering segments of heavy noise that the algorithm cannot handle properly.

## 3.4.2 Peak detection

The peak detection phase attempts to accommodate amplitude variation and morphology changes of the PPG complexes by using an adaptive peak detection threshold (Fig 3, III), followed by several steps of outlier detection and rejection. To identify heartbeats, a moving average is calculated using a window of 0.75 seconds on both sides of each data point. The first and last 0.75 seconds of the signal are populated with the signal's mean, no moving average is generated for these sections. Regions of interest (ROI) are marked between two points of intersection where the signal amplitude is larger than the moving average (Fig 3, I-II), which is a standard way of detecting peaks. R-peaks are marked at the maximum of each ROI.

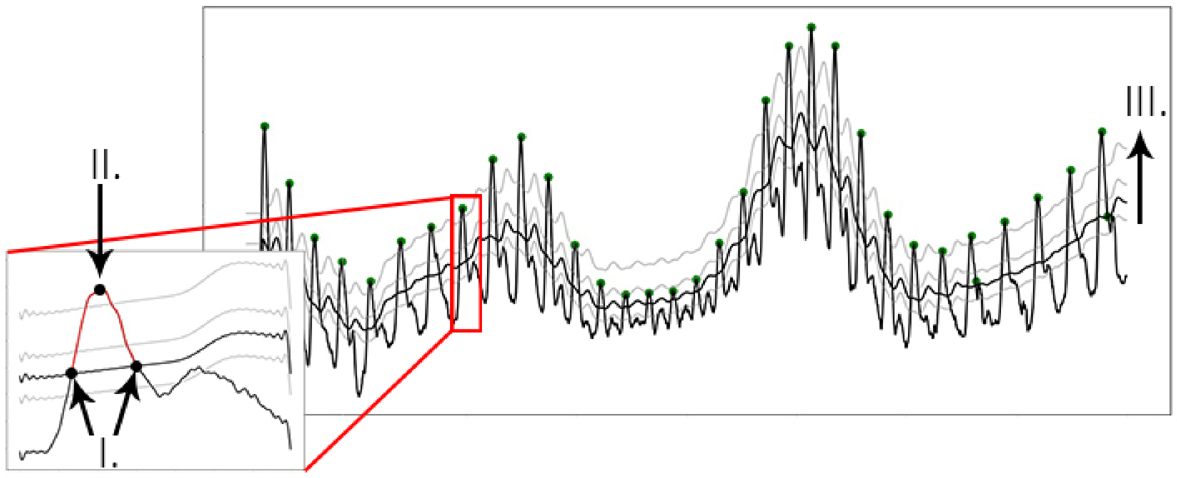


*Figure showing the process of peak extraction. A moving average is used as an intersection threshold (II). Candidate peaks are marked at the maximum between intersections (III). The moving average is adjusted stepwise to compensate for varying PPG waveform morphology (I).*

During the peak detection phase, the algorithm adjusts the amplitude of the calculated threshold stepwise. To find the best fit, the standard deviation between successive differences (SDSD, see also 2.2) is minimised and the signal's BPM is checked. This represents a fast method of approximating the optimal threshold by exploiting the relative regularity of the heart rate signal. As shown in the figure below, missing one R-peak (III.) already leads to a substantial increase in SDSD compared to the optimal fit (II.). Marking incorrect R-peaks also leads to an increase in SDSD (I.). The lowest SDSD value that is not zero, in combination with a likely BPM value, is selected as the best fit. The BPM must lie within a predetermined range (default: 40 <= BPM <= 180, range settable by user).

The figure below displays how the SDSD relates to peak fitting. In essence the fitting function exploits the strong regularity expected in the heart rate signal.

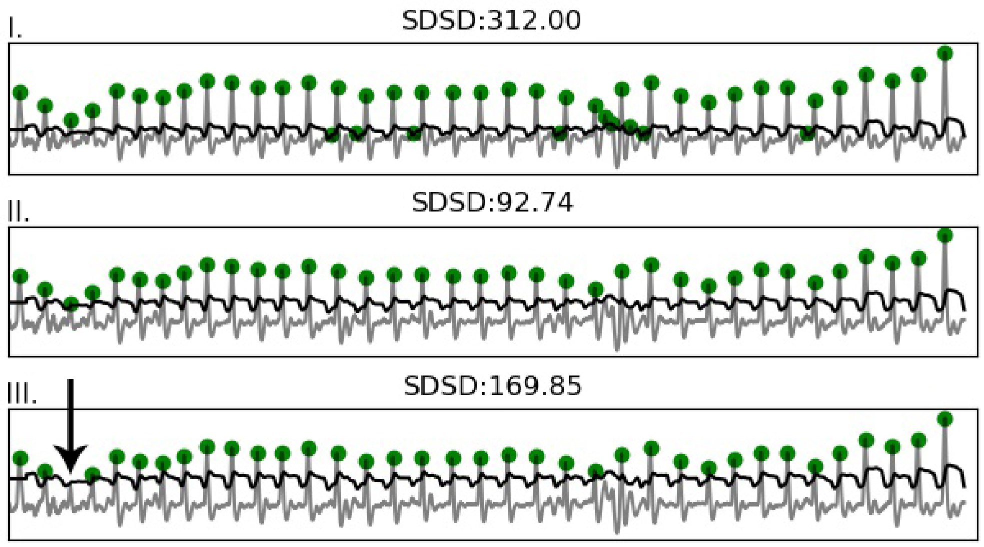


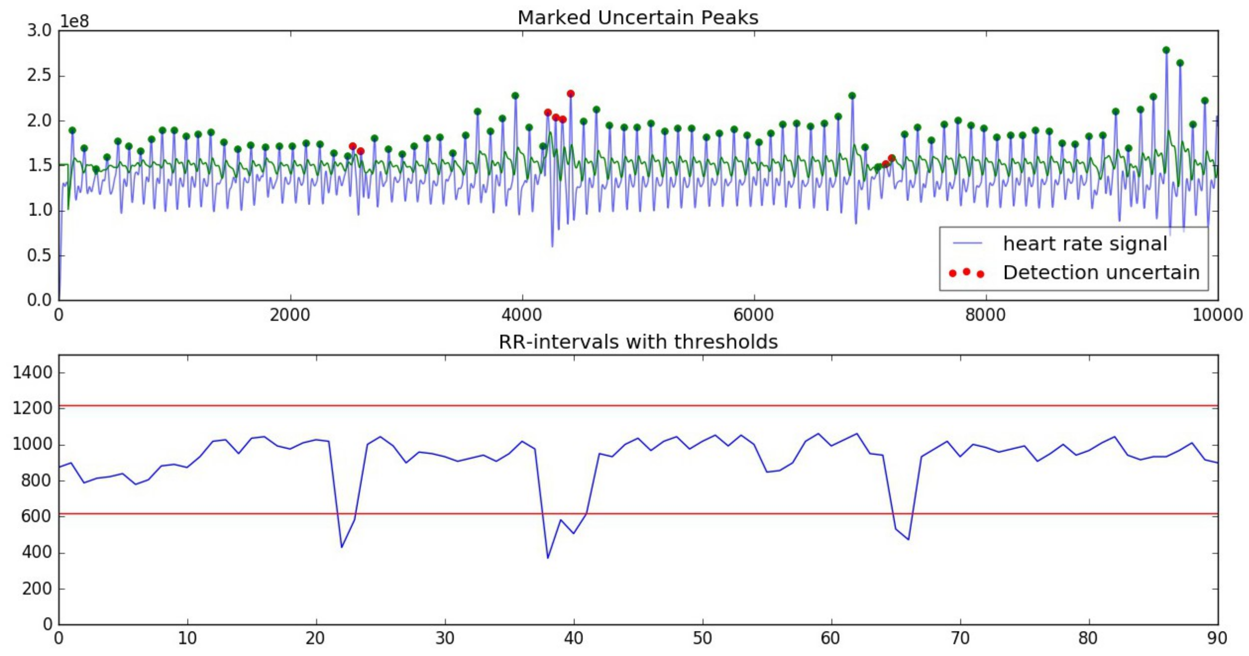
*Figure showing how the SDSD responds strongly even to a single missed beat (bottom plot), and is lowest when all peaks are properly detected (middle plot).*

Whenever clipping occurs, the algorithm detects this and will attempt to reconstruct the waveform by spline interpolation. This is discussed under *Clipping detection and interpolation*

An optional 'high precision mode' is available that takes the signal surrounding each detected peak (+/- 100ms on both ends), and upsamples it to simulate a higher sampling rate for more accurate peak position estimation. By default it upsamples to 1000Hz to provide ms-accurate peak position estimations.

### 3.4.3 Peak rejection

After the fitting phase, several incorrectly detected peaks may still remain due to various factors. These are tested and rejected based on a thresholded value for the RR-intervals in the section:



Thresholds are computed based on the mean of the RR-intervals in the segments. 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.

## 3.4.4 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

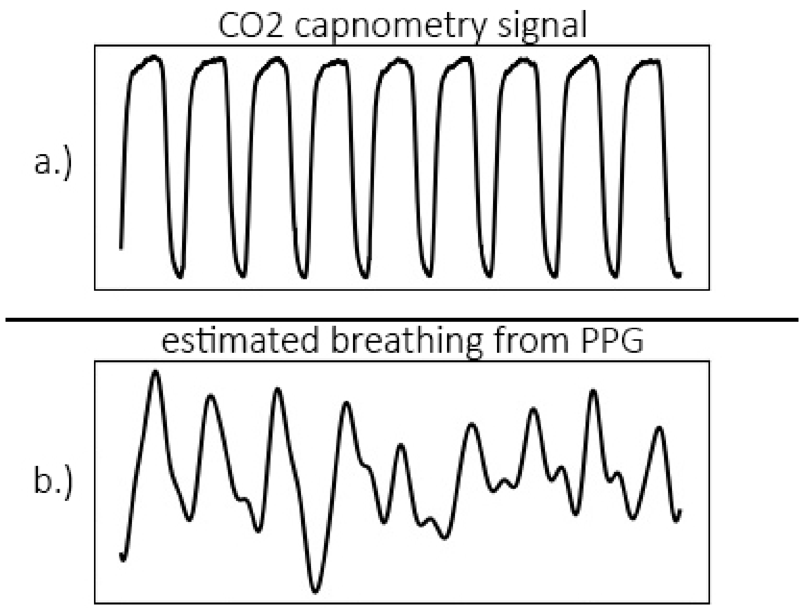
Frequency domain measures computed are:

- 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.

### Estimating breathing rate

One interesting property of the heart is that the frequency with which it beats is strongly influenced by breathing, through the autonomous nervous system. It is one of the reasons why deep breaths can calm nerves. We can also exploit this relationship to extract breathing rate from a segment of heart rate data. For example, using a dataset from[1] which contains both CO2 capnometry signals as well as PPG signals, we can see the relationship between breathing and the RR-intervals clearly. Below are plotted the CO2 capnometry signal (breathing signal measured at the nose), as well as the (upsampled) signal created by the RR-intervals:



The problem is now reduced to one of peak finding. Breathing rate can be extracted using the toolkit. After calling the 'process' function, breathing rate (in Hz) is available in the dict{} object that is returned.

```python
import heartpy as hp

data = hp.get_data('data.csv')
```

(continues on next page)

[1] W. Karlen, S. Raman, J. M. Ansermino, and G. A. Dumont, "Multiparameter respiratory rate estimation from the photoplethysmogram," IEEE transactions on bio-medical engineering, vol. 60, no. 7, pp. 1946–53, 2013. DOI: 10.1109/TBME.2013.2246160 PMED: http://www.ncbi.nlm.nih.gov/pubmed/23399950

(continued from previous page)

```
fs = 100.0
working_data, measures = hp.process(data, fs, report_time=True)
print('breathing rate is: %s Hz' %measures['breathingrate'])
```

This will result in:

```
breathing rate is: 0.16109544905356424 Hz
```

### 3.4.5 References

## 3.5 Development

### 3.5.1 Release Notes

#### V0.8.1

- Added changelog to repository
- Implemented clipping detection and interpolation functionality
- Changed FFT calculation flag to default False, as in some cases the FFT takes very long to compute. Possible causes and fixes to be investigated
- Pushed readthedocs.io documentation source structure to repository
- Added encoding argument to get_data function, per the NumPy deprecation of not using encoding. For more info: https://docs.scipy.org/doc/numpy-1.14.0/release.html#encoding-argument-for-text-io-functions

#### V0.8.2

- RR_difference interval no longer taken into account when RR-intervals are not technically adjacent due to rejected peak presence in between
- Moved matplotlib import statement so that it is no longer necessary unless calling the plot functionality, reduces need to install irrelevant dependencies when plotting functionality not needed
- Added Hampel Filter with settable filtersize
- Added method to suppress noisy segments called 'Hampel Corrector', called such as it's simply a Hampel Filter with large window size. Computationally on the expensive side so disabled by default, but very good at suppressing noisy segments without influencing peak positions in the rest of the signal.
- Added breathing rate extraction method. Stores estimated breathing rate in measures['breathingrate']
- Made BPM threshold values settable
- Added Periodogram- and Welch-based PSD estimation
- Added support for edge case where clipping segment starts early in signal, meaning there is insufficient data to interpolate accurately.

### V1.0

- Released Version 1.0
- Added flag to disable scaling when interpolating clipping segments. Useful for data with large amplitude variations.
- Added marking of rejected segments in plotter
- Added automatic peak rejection when first peak occurs within 150ms, since the signal might start just after a peak, which creates slight inaccuracy.
- Added segment rejection based on percentage of incorrect peaks.

### V1.0.1

- Changed segmentwise rejection API to simplify plotting

### V1.1

- We are now officially called HeartPy
- Changed overall structure to get rid of global dicts, allows for modular or multithreaded use easier.
- Changed docs to reflect changes

### V1.1.2

- Added high-pass and band-pass Butterworth filters
- Fixed case where no peak-peak differences over 20ms in a signal caused an exception
- Fixed case where intermittent noisy signals resulted in exception when calculating breathing rate
- Added scale_sections() function that uses local scaling rather than global
- Added preprocess_ecg(data, sample_rate) function that attempts to preprocess ecg data. Note: doubles sampling rate of returned data.
- Added highpass and bandpass filtering options. Called through filtersignal function with argument filtertype= lowpass/highpass/bandpass.
- Changed way peak fitting works by adding extra peak validation step between fitting phases

### V1.1.3

- Added functions to allow for continous measure output
- Added make_windows() function to divide input data into evenly sized segments with settable windowsize and settable overlap
- Added two functions to remove outliers from continous set: outliers_modified_z(), and outliers_iqr_method(). Both take a list of numpy array of one continous measure and remove outliers due to incorrectly analysed sections, if any outliers are persent.

### V1.1.4

- Added wrapper function 'process_segmentwise()' that splits hrdata in sections (overlap between sections is settable), and analyses each section separately. Returns two dict objects with all outputs.
- Changed rolling mean function to no longer flatten off the more it is raised, turned out to be more robust.
- Removed peak validation step implemented in V1.1.2 -> after further testing it turned out to be detrimental to many cases.
- Updated docs to reflect the changes to the codebase.

### V1.1.5

- Adapted *make_windows()* to accept tail end of data. Updated *process_segmentise()* to make use of this.
- Updated docs explaining the new functionality
- Fixed error where 'fast' segmentwise method returned single instance in dict rather than sequence
- Fixed error where 'fast' segmentwise method returned empty working_data object
- Started properly structuring module.

### V1.1.6

- moved nn20/nn50 temp containers to 'working_data' dict in stead of output measures (see issue #15).
- fixed wrongly unpacked kwargs in process_segmentwise
- deprecated doctests.txt for now - no longer functional and need updating.
- process_segmentwise() now returns the indices of the slices made in the original data, these are appended to both the returned measures{} and working_data{}. Closes #14
- updated process_segmentwise to allow control over last (likely incomplete) segment. Setting min_size to -1 now puts the tail end of the data into the last bin, making it larger than the others. Closes #16
- fixed sample_rate not being passed to rolmean() when esitmating the breathing rate

### V1.1.7

- added peak interpolation (high precision mode) method 'interpolate_peaks' that allows more accurate estimation of peak positions in signal of low sampling frequency
- in segmentwise processing, fixed bug where the interquartile-range was also used when modified z-score approach was requested.
- fixed mistake in argument order in process_segmentwise function docstring
- implemented 'segment_plotter()' function. This will plot segments and save plots to folder after running 'process_segmentwise()'.
- updated docs to include new functionality.

### V1.1.7a

- hotfix for process_segmentwise issue where multiple copies of the same index range were placed in the output.

#### V1.2

- Changed organisation HeartPy, it is now split into multiple modules to keep the growing library ordered. This opens the way to the planned addition of a GUI.
- Added examples that also function as doctests to all functions
- Added extensive documentation docstrings to all functions
- Added function load_exampledata() that loads the available example data files directly from github.
- Added several jupyter notebooks in Examples folder, illustrating how to work with different types of data.
- Added function to reject outliers in RR-list and compute measures based on cleaned list. See: clean_rr_intervals()

### 3.5.2 Questions

contact me at P.vanGent@tudelft.nl

# Python Module Index

h

heartpy.analysis, 28
heartpy.datautils, 34
heartpy.exceptions, 38
heartpy.filtering, 38
heartpy.peakdetection, 41
heartpy.preprocessing, 45
heartpy.visualizeutils, 49

# Index

A

append_dict() (*in module heartpy.peakdetection*), 41

B

BadSignalWarning, 38

C

calc_breathing() (*in module heartpy.analysis*), 33
calc_fd_measures() (*in module heartpy.analysis*), 32
calc_rr() (*in module heartpy.analysis*), 28
calc_rr_segment() (*in module heartpy.analysis*), 29
calc_ts_measures() (*in module heartpy.analysis*), 31
check_binary_quality() (*in module heartpy.peakdetection*), 44
check_peaks() (*in module heartpy.peakdetection*), 44
clean_rr_intervals() (*in module heartpy.analysis*), 30

D

detect_peaks() (*in module heartpy.peakdetection*), 42

E

enhance_ecg_peaks() (*in module heartpy*), 20
enhance_ecg_peaks() (*in module heartpy.preprocessing*), 47
enhance_peaks() (*in module heartpy*), 20
enhance_peaks() (*in module heartpy.preprocessing*), 46

F

filter_signal() (*in module heartpy*), 26
filter_signal() (*in module heartpy.filtering*), 38
fit_peaks() (*in module heartpy.peakdetection*), 43
flip_signal() (*in module heartpy*), 21
flip_signal() (*in module heartpy.preprocessing*), 48

G

get_data() (*in module heartpy*), 23
get_data() (*in module heartpy.datautils*), 34
get_samplerate_datetime() (*in module heartpy*), 25
get_samplerate_datetime() (*in module heartpy.datautils*), 35
get_samplerate_mstimer() (*in module heartpy*), 25
get_samplerate_mstimer() (*in module heartpy.datautils*), 35

H

hampel_correcter() (*in module heartpy*), 27
hampel_correcter() (*in module heartpy.filtering*), 40
hampel_filter() (*in module heartpy*), 27
hampel_filter() (*in module heartpy.filtering*), 39
heartpy.analysis (*module*), 28
heartpy.datautils (*module*), 34
heartpy.exceptions (*module*), 38
heartpy.filtering (*module*), 38
heartpy.peakdetection (*module*), 41
heartpy.preprocessing (*module*), 45
heartpy.visualizeutils (*module*), 49

I

interpolate_clipping() (*in module heartpy.preprocessing*), 48
interpolate_peaks() (*in module heartpy.peakdetection*), 45

L

load_exampledata() (*in module heartpy*), 24
load_exampledata() (*in module heartpy.datautils*), 37

M

`MAD()` (*in module heartpy.datautils*), 37
`make_windows()` (*in module heartpy.peakdetection*), 41

O

`outliers_iqr_method()` (*in module heartpy.datautils*), 36
`outliers_modified_z()` (*in module heartpy.datautils*), 36

P

`plot_breathing()` (*in module heartpy.visualizeutils*), 51
`plot_poincare()` (*in module heartpy.visualizeutils*), 50
`plotter()` (*in module heartpy*), 18
`plotter()` (*in module heartpy.visualizeutils*), 49
`process()` (*in module heartpy*), 14
`process_segmentwise()` (*in module heartpy*), 17

R

`remove_baseline_wander()` (*in module heartpy*), 21
`rolling_mean()` (*in module heartpy.datautils*), 36

S

`scale_data()` (*in module heartpy*), 22
`scale_data()` (*in module heartpy.preprocessing*), 45
`scale_sections()` (*in module heartpy*), 22
`scale_sections()` (*in module heartpy.preprocessing*), 46
`segment_plotter()` (*in module heartpy*), 19
`segment_plotter()` (*in module heartpy.visualizeutils*), 50
`smooth_signal()` (*in module heartpy.filtering*), 40

U

`update_rr()` (*in module heartpy.analysis*), 29