
pyHRV - OpenSource Python Toolbox for Heart Rate Variability Documentation

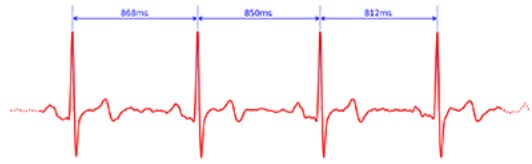
Release 0.4

Pedro Gomes

Jul 22, 2022

Contents:

1	Installation	3
2	R-Peak Detection with BioSPPy	5
3	Sample NNI Series	7
4	The biosppy.utils.ReturnTuple Object	9
5	What may help when matplotlib blocks your code from being executed	11
6	API Reference	13
6.1	pyHRV Function Levels	13
6.2	The HRV Function: hrv()	14
6.3	Time Domain Module	20
6.4	Frequency Domain Module	51
6.5	Nonlinear Module	92
6.6	Tools Module	104
6.7	Utils Module	124
7	Tutorials	133
7.1	Bulk Processing of Multiple NNI Series with pyHRV	133
7.2	ECG Acquisition & HRV Analysis with BITalino & pyHRV	137
8	License & Disclaimer	147
8.1	License	147
8.2	Disclaimer	147
9	Highlights	149
10	Installation	151
11	Disclaimer & Context	153
	Index	155



Python Toolbox for Heart Rate Variability

pyHRV is a toolbox for Heart Rate Variability (HRV) written in Python. The toolbox bundles a selection of functions to compute Time Domain, Frequency Domain, and nonlinear HRV parameters, along with other additional features designed to support your HRV research.

CHAPTER 1

Installation

Use the `pip` tool to download `pyHRV` from the [Python Package Index \(PyPi\)](#). If you are using macOS or a Linux-based operating system, open the *Terminal* and type in the command below. Windows users may use the *Command Prompt*.

```
$ pip install pyhrv
```

`pyHRV` depends on the following third-party packages, which will be automatically installed using the `pip` tool, if these are not already on your machine:

- `biosppy`
- `numpy`
- `scipy`
- `matplotlib`
- `nolds`
- `spectrum`

Note: This has been primarily developed for the Python 2.7 programming language. Running the `pip` command above may cause errors when trying to install the package using Python 3.

In this case, try to install the `pyHRV` dependencies first:

```
$ pip install biosppy
$ pip install matplotlib
$ pip install numpy
$ pip install scipy
$ pip install nolds
$ pip install spectrum
```

Note: Alternatively, it is recommended to install the [Anaconda](#) software, which comes with a compatible Python 2.7 distribution and all the necessary (and more) third-party packages for scientific computing.

R-Peak Detection with BioSPPy

BioSPPy is a toolbox for biosignal processing, and comes with built-in ECG processing and R-peak detection algorithms. These can be used to compute the NNI series upon which the HRV parameters can be computed.

An example of this procedure is demonstrated below, using ECG data acquired with the [BITalino \(r\)evolution](#) hardware and the [OpenSignals \(r\)evolution](#) software. The ECG signals are imported and converted to mV using the [opensignal-reader](#) package.

```
import biosppy
import numpy as np
import pyhrv.tools as tools
from opensignalsreader import OpenSignalsReader

# Load sample ECG signal & extract R-peaks using BioSppy
signal = OpenSignalsReader('./samples/SampleECG.txt').signal('ECG')
signal, rpeaks = biosppy.signals.ecg.ecg(signal, show=False)[1:3]

# Compute NNI
nni = tools.nn_intervals(rpeaks)
```

Note: pyHRV can of course be used with any ECG - or ECG Lead I like - signal and is not limited to signals acquired with specific devices or software. The instructions above are merely an example.

CHAPTER 3

Sample NNI Series

pyHRV comes with sample data available [here](#) folder.

The biosppy.utils.ReturnTuple Object

The results of the pyHRV parameter functions, wrapped and returned as `biosppy.utils.ReturnTuple` objects. This package-specific class combines the advantages of Python dictionaries (indexing using keywords) and Python tuples (immutable). Parameter values stored in the `ReturnTuple` object can be accessed as follows:

```
from biosppy import utils

# Store sample data in a ReturnTuple object
args = (500, 600, )
names = ('parameter1', 'parameter2', )
results = utils.ReturnTuple(args, names)

# Get and print 'parameter1'
print(results['parameter1'])
```

See also:

- [BioSPPy API Reference - ReturnTuple](#)
- [Note on ReturnTuple objects](#)

What may help when matplotlib blocks your code from being executed

The plots generated by the functions of pyHRV use matplotlib as the fundamental plotting library. The default backend configuration of this library can cause some unwanted behaviour, where your Python scripts are interrupted whenever a plot is shown.

Important: This issue can be solved by switching the matplotlib backend to a backend that supports the matplotlib `.interactive()` mode. This mode allows you to show the generated plots without interrupting your Python script. The Qt4Agg has shown to be a suitable backend to solve this issue on Windows and macOS.

Add the following lines of code **at the top of your script, before importing the other Python packages** to try to solve this issue:

```
# Import matplotlib and set the 'Qt4Agg' backend to support interactive mode on_
↪ Windows and macOS
import matplotlib
matplotlib.use('Qt4Agg')

# Activate interactive mode
import matplotlib.pyplot as plt
plt.ion()
```


6.1 pyHRV Function Levels

The HRV feature extraction functions of the pyHRV toolbox have been implemented and categorized into three levels, which are intended to facilitate the usage and increase the usability of the toolbox according to the needs of the user or programming background. This multilevel-architecture is illustrated in the Figure below.

Level 1 - HRV Level:

This level consists of a single function that allows you to compute the full range of HRV parameters using only a single line of code by calling the `hrv()` function found in the `hrv.py`. This function calls all underlying HRV parameter functions and returns the bundled results in a `biosppy.utils.ReturnTuple()` object. Custom settings or parameters for the computation of specific parameters can be passed to the `hrv()` function if needed using the `kwargs` input dictionaries. Otherwise, the default values will be used.

See also:

The HRV Function: `hrv()`

Level 2 - Domain Level:

This level of module/domain functions intended for users that want to compute all the parameters of a specific domain. Using the example of the Time Domain, the `time_domain()` function calls all the parameter function of this domain and returns the results bundled in a `biosppy.utils.ReturnTuple()` object. As in the previous level, user-specific settings and parameters can be passed to the domain functions using the available `kwargs` dictionaries. The module level function can be found in the respective domain modules.

See also:

- *Domain Level Function: `time_domain()`* (`time_domain.py`)
- *2D PSD Comparison Plot: `psd_comparison()`* (`frequency_domain.py`)
- *Domain Level Function: `nonlinear()`* (`nonlinear.py`)

Level 3 - Parameter Level:

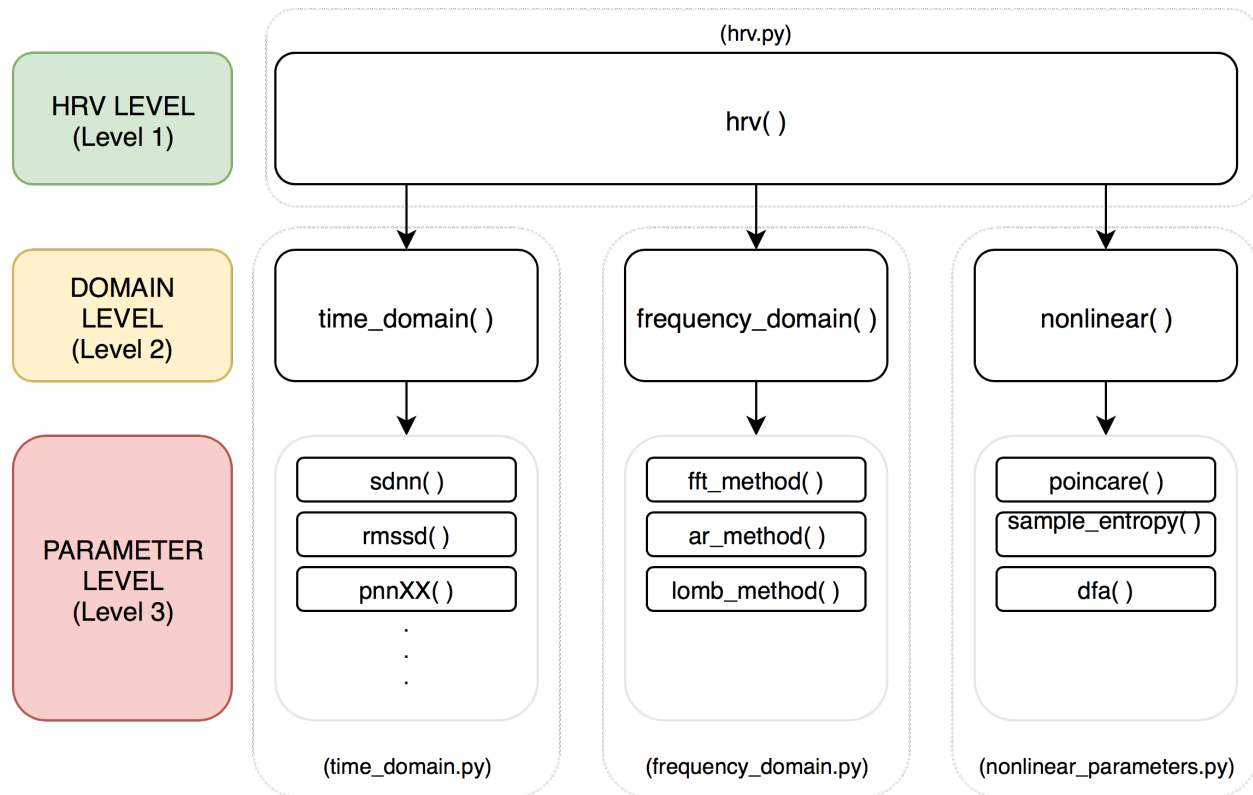


Fig. 1: Multilevel architecture of the pyHRV toolbox.

This level parameter-specific functions for the case that only want to compute single, specific parameters (individually) (e.g. `sdnn()` returns the SDNN parameter). This allows you to select only the parameters or features required for your specific application. User-specific settings and parameters can be made directly when overloading the function.

6.2 The HRV Function: `hrv()`

```
pyhrv.hrv.hrv(nni=None, rpeaks=None, signal=None, sampling_rate=1000., interval=[0,
10], plot_ecg=True, plot_Tachogram=True, show=True, fbands=None,
kwargs_ecg_plot=None, kwargs_tachogram=None, kwargs_)
```

Function Description

Computes all HRV parameters of the pyHRV toolkit (see list below).

See also:

- [Time Domain Module](#)
- [Frequency Domain Module](#)
- [Nonlinear Module](#)

Input Parameters

- `nni` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]
- `signal` (array): ECG signal

- `sampling_rate` (int, float, optional): Sampling rate in [Hz] used for the ECG acquisition (default: 1000Hz)
- `interval` (array, optional): Visualization interval of the Tachogram plot (default: None: [0s, 10s])
- `plot_ecg` (bool, optional): If True, plots ECG signal with specified interval ('signal' must not be None)
- `plot_tachogram` (bool, optional): If True, plots tachogram with specified interval
- `show` (bool, optional): If True, shows the ECG plot figure (default: True)
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `kwargs_ecg_plot` (dict, optional): ****kwargs** for the `plot_ecg()` function (see 'tools.py' module)
- `kwargs_tachogram` (dict, optional): ****kwargs** for the `plot_tachogram()` function (see 'tools.py' module)
- `kwargs_time` (dict, optional): ****kwargs** for the `time_domain()` function (see 'time_domain()' function)
- `kwargs_welch` (dict, optional): ****kwargs** for the 'welch_psd()' function (see 'frequency_domain.py' module)
- `kwargs_lomb` (dict, optional): ****kwargs** for the 'lomb_psd()' function (see 'frequency_domain.py' module)
- `kwargs_ar` (dict, optional): ****kwargs** for the 'ar_psd()' function (see 'frequency_domain.py' module)
- `kwargs_nonlinear` (dict, optional): ****kwargs** for the 'nonlinear()' function (see 'nonlinear.py' module)

Note: If `fbands` is none, the default values for the frequency bands will be set.

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below for more information on how to define custom frequency bands.

The `show` parameter is equally set for all plotting functions.

Important: This function computes the Time Domain parameters using either the `signal`, `nni`, or `rpeaks` data. Provide only one type of data, as it is not required to pass all three types at once.

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

Time Domain:

- `nni_counter` (int): Number of NNI (-)
- `nni_mean` (float): Mean NNI [ms]
- `nni_min` (int): Minimum NNI [ms]
- `nni_max` (int): Maximum NNI [ms]
- `nni_diff_mean` (float): Mean NNI difference [ms]

- `nni_diff_min` (int): Minimum NNI difference [ms]
- `nni_diff_max` (int): Maximum NNI difference [ms]
- `hr_mean` (float): Mean heart rate [bpm]
- `hr_min` (int): Minimum heart rate [bpm]
- `hr_max` (int): Maximum heart rate [bpm]
- `hr_std` (float): Standard deviation of the heart rate series [bpm]
- `sdnn` (float): Standard deviation of NN intervals [ms]
- `sdnn_index` (float): SDNN Index [ms]
- `sdann` (float): SDANN [ms]
- `rmssd` (float): Root mean of squared NNI differences [ms]
- `sdsd` (float): Standard deviation of NNI differences [ms]
- `nnXX` (int, optional): Number of NN interval differences greater than the specified threshold (-)
- `pnnXX` (float, optional): Ratio between nnXX and total number of NN interval differences (-)
- `nn50` (int): Number of NN interval differences greater 50ms
- `pnn50` (float): Ratio between NN50 and total number of NN intervals [ms]
- `nn20` (int): Number of NN interval differences greater 20ms
- `pnn20` (float): Ratio between NN20 and total number of NN intervals [ms]
- `nn_histogram` (matplotlib figure object): Histogram plot figure (only if input parameter `plot` is True)
- `tinn_n` (float): N value of the TINN computation (left corner of the interpolated triangle at (N, 0))
- `tinn_m` (float): M value of the TINN computation (right corner of the interpolated triangle at (M, 0))
- `tinn` (float): TINN (baseline width of the interpolated triangle) [ms]
- `tri_index` (float): Triangular index [ms]

Important: The XX in the `nnXX` and the `pnnXX` keys are substituted by the specified threshold.

For instance, `nnXX(nni, threshold=30)` returns the custom `nn30` and `pnn30` parameters. Applying `threshold=35` as `nnXX(nni, threshold=35)` returns the custom `nn35` and `pnn35` parameters.

These parameters are only returned if a custom threshold (`threshold`) has been defined in the input parameters.

Frequency Domain (X = one of the methods 'fft', 'ar', 'lomb'):

- `X_peak` (tuple): Peak frequencies of all frequency bands [Hz]
- `X_abs` (tuple): Absolute powers of all frequency bands [ms²]
- `X_rel` (tuple): Relative powers of all frequency bands [%]
- `X_log` (tuple): Logarithmic powers of all frequency bands [log]
- `X_norm` (tuple): Normalized powers of the LF and HF frequency bands [-]
- `X_ratio` (float): LF/HF ratio [-]
- `X_total` (float): Total power over all frequency bands [ms²]
- `X_plot` (matplotlib figure object): PSD plot figure object

- `fft_interpolation` (str): Interpolation method used for NNI interpolation (hard-coded to 'cubic')
- `fft_resampling_frequency` (int): Resampling frequency used for NNI interpolation [Hz] (hard-coded to 4Hz as recommended by the [HRV Guidelines](#))
- `fft_window` (str): Spectral window used for PSD estimation of the Welch's method
- `lomb_ma` (int): Moving average window size
- `ar_interpolation` (str): Interpolation method used for NNI interpolation (hard-coded to 'cubic')
- `ar_resampling_frequency` (int): Resampling frequency used for NNI interpolation [Hz] (hard-coded to 4Hz as recommended by the [HRV Guidelines](#))
- `ar_order` (int): Autoregressive model order

Nonlinear:

- `poincare_plot` (matplotlib figure object): Poincaré plot figure
- `sd1` (float): Standard deviation (SD1) of the major axis
- `sd2` (float): Standard deviation (SD2) of the minor axis
- `sd_ratio` (float): Ratio between SD1 and SD2 (SD2/SD1)
- `ellipse_area` (float): Area S of the fitted ellipse
- `sample_entropy` (float): Sample entropy of the NNI series
- `dfa_short` (float): Alpha value of the short-term fluctuations (alpha1)
- `dfa_long` (float): Alpha value of the long-term fluctuations (alpha2)

See also:

The `biosppy.utils.ReturnTuple` Object

Application Notes

It is not necessary to provide input data for `signal`, `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`signal`, `nni` **or** `rpeaks`). The input data will be prioritized in the following order, in case multiple inputs are provided:

1. `signal`, 2. `nni`, 3. `rpeaks`.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()` function.

See also:

- *[What may help when matplotlib blocks your code from being executed](#)*
- *[More information about the matplotlib Interactive Mode](#)*
- *[More information about matplotlib Backends](#)*

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

Use the `kwargs_ecg_plot` dictionary to pass function specific parameters for the `plot_ecg()` function. The following keys are supported:

- `rpeaks` (bool, optional): If True, marks R-peaks in ECG signal (default: True)
- `title` (str, optional): Plot figure title (default: None)

Use the `kwargs_tachogram` dictionary to pass function specific parameters for the `plot_tachogram()` function. The following keys are supported:

- `hr` (bool, optional): If True, plot HR series in [bpm] on second axis (default: True)
- `title` (str, optional): Optional plot figure title (default: None)

Use the `kwargs_time` dictionary to pass function specific parameters for the `time_domain()` function. The following keys are supported:

- `threshold` (int, optional): Custom threshold in [ms] for the optional NNXX and pNNXX parameters (default: None)
- `plot` (bool, optional): If True, creates histogram using matplotlib, else uses NumPy for histogram data only (geometrical parameters, default: True)
- `binsize` (float, optional): Bin size in [ms] of the histogram bins - (geometrical params, default: 7.8125ms).

Use the `kwargs_welch` dictionary to pass function specific parameters for the `welch_psd()` method. The following keys are supported:

- `nfft` (int, optional): Number of points computed for the FFT result (default: 2**12)
- `detrend` (bool, optional): If True, detrend NNI series by subtracting the mean NNI (default: True)
- `window` (scipy.window function, optional): Window function used for PSD estimation (default: 'hamming')

Use the `lomb_psd` dictionary to pass function specific parameters for the `lombg_psd()` method. The following keys are supported:

- `nfft` (int, optional): Number of points computed for the Lomb-Scargle result (default: 2**8)
- `ma_order` (int, optional): Order of the moving average filter (default: None; no filter applied)

Use the `ar_psd` dictionary to pass function specific parameters for the `ar_psd()` method. The following keys are supported:

- `nfft` (int, optional): Number of points computed for the FFT result (default: 2**12)

- `order` (int, optional): Autoregressive model order (default: 16)

Use the `kwargs_nonlinear` dictionary to pass function specific parameters for the `nonlinear()` function. The following keys are supported:

- `ellipse` (bool, optional): If True, shows fitted ellipse in plot (default: True)
- `vectors` (bool, optional): If True, shows SD1 and SD2 vectors in plot (default: True)
- `legend` (bool, optional): If True, adds legend to the Poincaré plot (default: True)
- `marker` (str, optional): NNI marker in plot (must be compatible with the matplotlib markers (default: 'o'))
- `dim` (int, optional): Entropy embedding dimension (default: 2)
- `tolerance` (int, float, optional): Tolerance distance for which the two vectors can be considered equal (default: `std(NNI)`)
- `short` (array, optional): Interval limits of the short-term fluctuations (default: None: [4, 16])
- `long` (array, optional): Interval limits of the long-term fluctuations (default: None: [17, 64])

Examples & Tutorials & Tutorials

The following example codes demonstrate how to use the `hrv()` function.

You can choose either the ECG signal, the NNI series or the R-peaks as input data for the PSD estimation and parameter computation:

```
# Import packages
import biosppy
import pyhrv.tools as tools
from pyhrv.hrv import hrv

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNI series
nni = tools.nn_intervals(t[rpeaks])

# OPTION 1: Compute Time Domain parameters using the ECG signal
signal_results = hrv(signal=filtered_signal)

# OPTION 2: Compute Time Domain parameters using the R-peak series
rpeaks_results = hrv(rpeaks=t[rpeaks])

# OPTION 3: Compute Time Domain parameters using the NNI-series
nni_results = hrv(nni=nni)
```

The output of all three options above will be the same.

Note: If an ECG signal is provided, the signal will be filtered and the R-peaks will be extracted using the `biosppy.signals.ecg.ecg()` function. Finally, the NNI series for the PSD estimation will be computed from the extracted R-peak series. The ECG plot is only generated if an ECG signal is provided.

See also:

`biosppy.signals.ecg.ecg()`

You can now access the parameters using the output parameter keys (works the same for the `rpeaks_results` and `nni_results`):

```
# Print SDNN
print(signal_results['sdnn'])

# Print RMSSD
print(signal_results['rmssd'])
```

Use the *kwargs* input dictionaries to provide custom input parameters.

```
# Define custom input parameters using the kwargs dictionaries
kwargs_time = {'threshold': 35}
kwargs_nonlinear = {'vectors': False}
kwargs_welch = {'nfft': 2**8}
kwargs_lomb = {'nfft': 2**16}
kwargs_ar = {'nfft': 2**8}
kwargs_tachogram = {'hr': False}
kwargs_ecg_plot = {'title': 'My ECG Signal'}

# Compute HRV parameters
hrv(nni=nni, kwargs_time=kwargs_time, kwargs_nonlinear=kwargs_nonlinear, kwargs_
↪ar=kwargs_ar,
    kwargs_lomb=kwargs_lomb, kwargs_welch=kwargs_welch, kwargs_tachogram=kwargs_
↪tachogram)
```

pyHRV is robust against invalid parameter keys. For example, if an invalid input parameter such as *nfft* is provided with the *kwargs_time* dictionary, this parameter will be ignored and a warning message will be issued.

```
# Define custom input parameters using the kwargs dictionaries
kwargs_time = {
    'threshold': 35,      # Valid key, will be used
    'nfft': 2**8         # Invalid key for the time domain, will be ignored
}

# Compute HRV parameters
hrv(nni=nni, kwargs_time=kwargs_time)
```

This will trigger the following warning message.

Warning: *Unknown kwargs for ‘time_domain()’: nfft. These kwargs have no effect.*

6.3 Time Domain Module

The `time_domain.py` module contains all the functions to compute the HRV time domain parameters.

See also:

[pyHRV Time Domain Module source code](#)

Module Contents

- *Time Domain Module*

- *NNI Parameters: `nni_parameters()`*
- *NNI Parameters: `nni_differences_parameters()`*
- *Heart Rate Parameters: `hr_parameters()`*
- *SDNN: `sdnn()`*
- *SDNN Index: `sdnn_index()`*
- *SDANN: `sdann()`*
- *RMSSD: `rmssd()`*
- *SDSD: `sdsd()`*
- *NNXX: `nnXX()`*
- *NN50: `nn50()`*
- *NN20: `nn20()`*
- *Geometrical Parameters*
 - * *TINN: `tinn()`*
 - * *Triangular Index: `triangular_index()`*
 - * *Geometrical Parameters Function: `geometrical_parameters()`*
- *Domain Level Function: `time_domain()`*

6.3.1 NNI Parameters: `nni_parameters()`

`pyhrv.time_domain.nni_parameters (nni=None, rpeaks=None)`

Function Description

Computes basic statistical parameters from a series of NN intervals (# of intervals, mean, min, max).

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `nni_counter` (int): Number of NNI (-)
- `nni_mean` (float): Mean NNI [ms]
- `nni_min` (int): Minimum NNI [ms]
- `nni_max` (int): Maximum NNI [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute parameters
results = td.nni_parameters(nni)

# Print minimum NNI
print(results['nni_min'])
```

Alternatively, you can use R-peak series (`rpeaks`) data to compute the NNI parameters.

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameters using R-peak series
results = td.nni_parameters(rpeaks=t[rpeaks])
```

6.3.2 NNI Parameters: `nni_differences_parameters()`

`pyhrv.time_domain.nni_differences_parameters` (*nni=None, rpeaks=None*)

Function Description

Computes basic statistical parameters from a series of NN interval differences (# of intervals, mean, min, max).

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `nni_diff_mean` (float): Mean NNI difference [ms]

- `nni_diff_min` (int): Minimum NNI difference [ms]
- `nni_diff_max` (int): Maximum NNI difference [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute parameters
results = td.nni_differences_parameters(nni)

# Print maximum NNI difference
print(results['nni_diff_max'])
```

Alternatively, you can use R-peak series (`rpeaks`) data to compute the NNI parameters.

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameters using R-peak series
results = td.nni_differences_parameters(rpeaks=t[rpeaks])
```

6.3.3 Heart Rate Parameters: `hr_parameters()`

`pyhrv.time_domain.hr_parameters` (`nni=None`, `rpeaks=None`)

Function Description

Computes basic statistical parameters from a series of heart rate (HR) data (mean, min, max, standard deviation)

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `hr_mean` (float): Mean heart rate [bpm]
- `hr_min` (int): Minimum heart rate [bpm]
- `hr_max` (int): Maximum heart rate [bpm]
- `hr_std` (float): Standard deviation of the heart rate series [bpm]

See also:

The `biosppy.utils.ReturnTuple` Object

Computation

The Heart Rate series is computed as follows:

$$HR_j = \frac{60000}{NNI_j}$$

for $0 \leq j \leq n$

with:

- HR_j : Heart rate j (in [bpm])
- NNI_j : NN interval j (in [ms])
- n : Number of NN intervals

See also:

Heart Rate: `heart_rate()`

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
```

(continues on next page)

(continued from previous page)

```

nni = pyhrv.utils.load_sample_nni()

# Compute parameters
results = td.hr_parameters(nni)

# Print maximum HR
print(results['hr_max'])

```

Alternatively, you can use R-peak series (rpeaks) to compute the HR parameters.

```

# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameters using R-peak series
results = td.hr_parameters(rpeaks=t[rpeaks])

```

6.3.4 SDNN: sdnn()

`pyhrv.time_domain.sdnn(nni=None, rpeaks=None)`

Function Description

Computes the Standard Deviation of a NN interval series (SDNN).

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results:

- `sdnn` (float): Standard deviation of NN intervals [ms]

See also:

The biosppy.utils.ReturnTuple Object

Parameter Computation

The SDNN parameter is computed according to the following formula:

$$SDNN = \sqrt{\frac{1}{n-1} \sum_{j=1}^n (NNI_j - \overline{NNI})^2}$$

with:

- n : Number of NNI
- NNI_j : NNI_j

- \overline{NNI} : Mean of NNI series

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the SDNN parameter:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute SDNN parameter
results = td.sdnns(nni)

# Print SDNN
print(results['sdnn'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameter using R-peak series
results = td.sdnns(rpeaks=t[rpeaks])
```

6.3.5 SDNN Index: `sdnn_index()`

`pyhrv.time_domain.sdnns_index(nni=None, rpeaks=None, full=False, duration=300, warn=True)`

Function Description

Computes the SDNN Index of an NNI series with a specified segmentation duration of `duration` (300 seconds = 5 minutes by default).

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].

- `rpeaks` (array): R-peak times in [ms] or [s].^β
- `full` (bool, optional): If True, returns the last segment even if its duration is significantly shorter than `duration` (default: False).
- `duration` (int, optional): Maximum duration per segment in [s] (default: 300 seconds)
- `warn` (bool, optional): If True, raise a warning message if a segmentation could not be conducted (`duration` > NNI series duration)

Note: `full` is False by default which causes the last segment to be dropped.

For instance, if processing an NNI series of 12.5min and the default segment duration of 5min, the segmentation function would split this series into 3 segments of 5min, 5min and 2.5min in duration. In this case, the last segment greatly alters the SDNN Index. Set the `full` parameter to False to drop the last segment or to True to compute the SDNN Index even with shorter segments.

Use the `warn` input argument to decide whether you want to see warning messages in the Python terminal, which would appear if a segmentation of the signal could not be conducted (e.g. `duration` > NNI duration).

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results:

- `sdnn_index` (float): SDNN Index [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Parameter Computation

The SDNN Index is computed using the `pyhrv.time_domain.sdnn()` and the `pyhrv.tools.segmentation()` functions.

See also:

- *SDNN: `sdnn()`*
- *Segmentation: `segmentation()`*

First, the input NNI series is segmented into segments of ~5 minutes in duration. Second, the SDNN parameter of each segment is computed. Finally, the mean value of all computed SDNN values is computed.

These steps are presented in the flow chart below.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

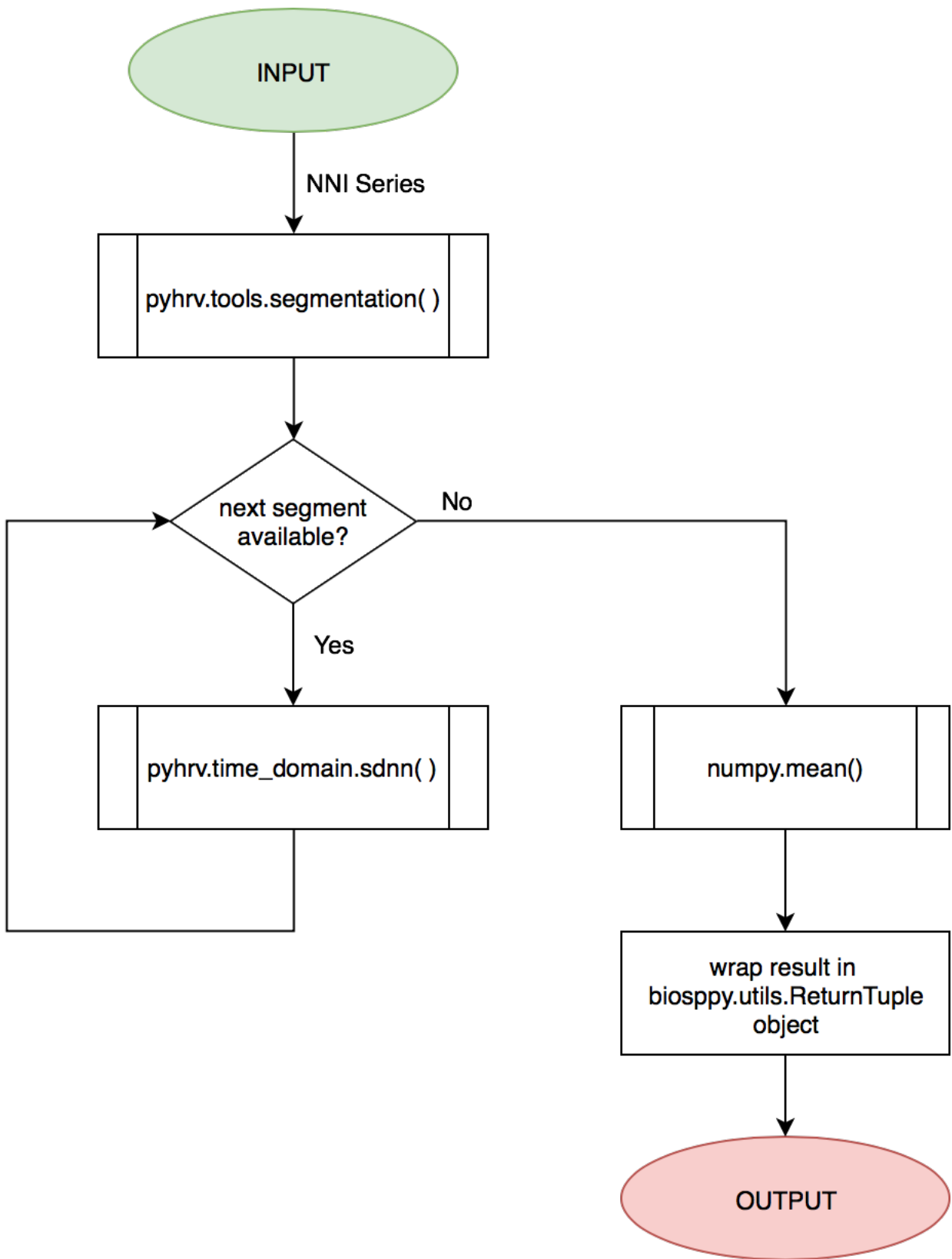
See also:

Section *NN Format: `nn_format()`* for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the SDNN parameter:



```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute SDNN Index parameter
results = td.sdnn_index(nni)

# Print SDNN index
print(results['sdnn_index'])
```

Alternatively, you can use R-peak series (`rpeaks`) to compute the SDNN Index:

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameter using R-peak series
results = td.sdnn_index(rpeaks=t[rpeaks])
```

6.3.6 SDANN: `sdann()`

`pyhrv.time_domain.sdann(nni=None, rpeaks=None, full=False, duration=300, warn=True)`

Function Description

Computes the SDANN of an NNI series with a specified segmentation duration of `duration` (300s=5min by default).

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `full` (bool, optional): If True, returns the last segment even if its duration is significantly shorter than `duration` (default: False).
- `duration` (int, optional): Maximum duration per segment in [s] (default: 300 seconds)
- `warn` (bool, optional): If True, raise a warning message if a segmentation could not be conducted (`duration > NNI series duration`)

Note: `full` is False by default which causes the last segment to be dropped.

For instance, if processing an NNI series of 12.5min and the default segment duration of 5min, the segmentation function would split this series into 3 segments of 5min, 5min and 2.5min in duration. In this case, the last segment greatly alter the SDNN Index. Set the `full` parameter to False to drop the last segment or to True to compute the SDNN Index even with shorter segments.

Use the `warn` input argument to decide whether you want to see warning messages in the Python terminal, which would appear if a segmentation of the signal could not be conducted (e.g. `duration > NNI duration`).

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results:

- `sdann` (float): SDANN [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Parameter Computation

The SDANN is computed using the `pyhrv.time_domain.sdnn()` and the `pyhrv.tools.segmentation()` functions.

See also:

- *SDNN: `sdnn()`*
- *Segmentation: `segmentation()`*

First, the input NNI series is segmented into segments of ~5 minutes in duration. Second, the mean of each segment is computed. Finally, the SDNN value of all computed mean values is computed.

These steps are presented in the flow chart below.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format: `nn_format()`* for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the SDANN parameter:

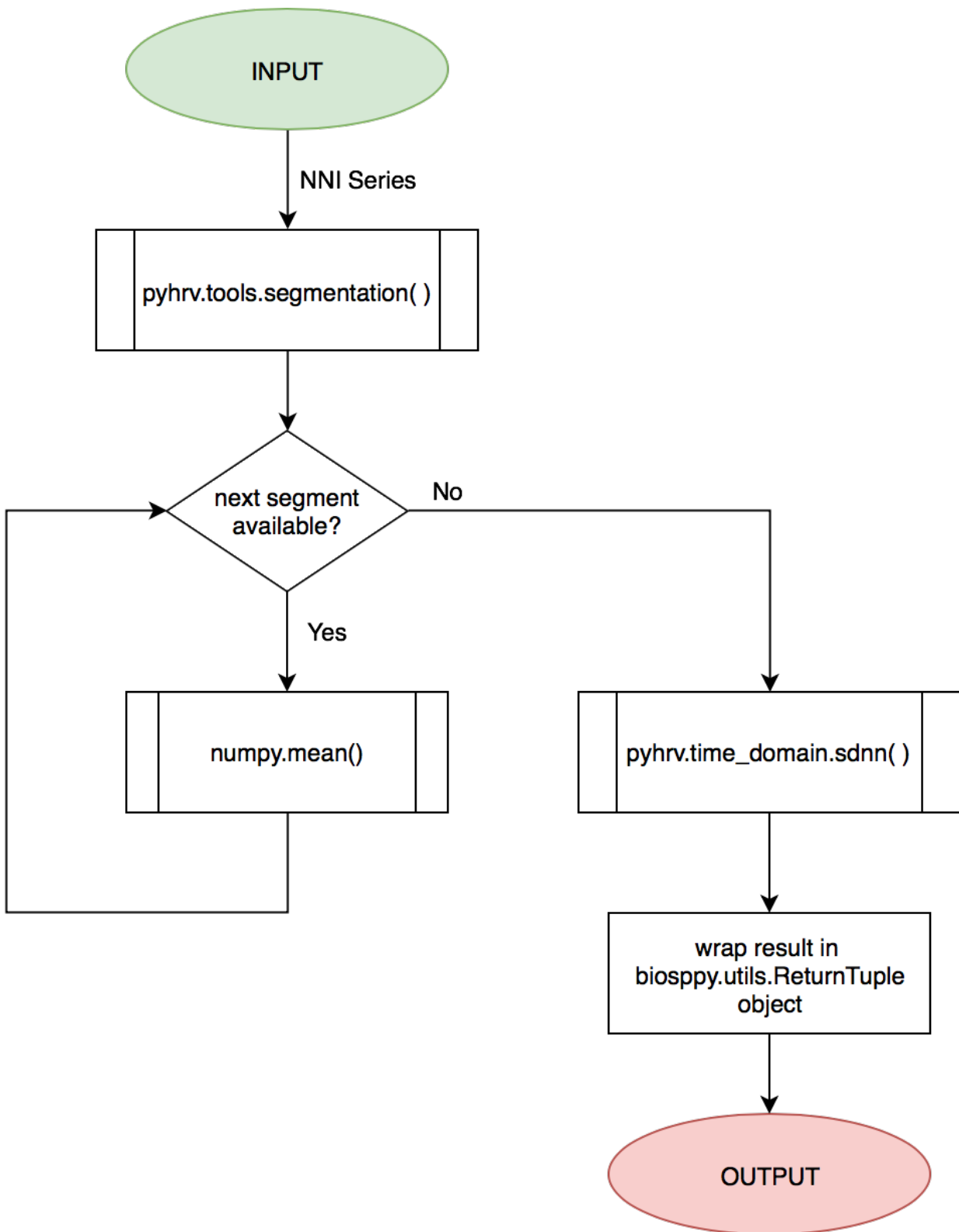
```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute SDANN parameter
results = td.sdann(nni)

# Print SDANN
print(results['sdann'])
```

Alternatively, you can use R-peak series (`rpeaks`) to compute the SDANN:



```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameter using R-peak series
results = td.sdann(rpeaks=t[rpeaks])
```

6.3.7 RMSSD: `rmssd()`

`pyhrv.time_domain.rmssd(nni=None, rpeaks=None)`

Function Description

Computes the root mean of squared NNI differences.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results: The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results:

- `rmssd` (float): Root mean of squared NNI differences [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Parameter Computation

The RMSSD parameter is computed according to the following formula:

$$RMSSD = \sqrt{\frac{1}{n-1} \sum_{j=1}^n \Delta NNI_j^2}$$

with:

- n : Number of NNI
- ΔNNI_j : NNI differences

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format: `nn_format()`* for more information.

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object using the output key 'rmssd'.

You can use NNI series (`nni`) to compute the RMSSD parameter:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute RMSSD parameter
results = td.rmssd(nni)

# Print RMSSD
print(results['rmssd'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameter using R-peak series
results = td.rmssd(rpeaks=t[rpeaks])
```

6.3.8 SDSD: `sdsd()`

`pyhrv.time_domain.sdsd(nni=None, rpeaks=None)`

Function Description

Standard deviation of NNI differences.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results:

- `sdsd` (float): Standard deviation of NNI differences [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Parameter Computation

The SDDS parameter is computed according to the following formula:

$$SDDS = \sqrt{\frac{1}{n-1} \sum_{j=1}^n (\Delta NNI_j - \overline{\Delta NNI})^2}$$

with:

- n : Number of NNI
- ΔNNI_j : NNI differences
- $\overline{\Delta NNI}$: Mean NNI

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object using the output key 'sdsd'.

You can use NNI series (`nni`) to compute the SDDS parameter:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute SDDS parameter
results = td.sdsd(nni)

# Print SDDS
print(results['sdsd'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute parameter using R-peak series
results = td.sdsd(rpeaks=t[rpeaks])
```

6.3.9 NNXX: nnXX()

`pyhrv.time_domain.nnXX(nni=None, rpeaks=None, threshold=None)`

Function Description

Derives the NNXX parameters: Finds number of NN interval differences greater than a specified threshold XX and the ratio between number of intervals > threshold and the total number of NN interval differences.

Hint: Other than the `nn50()` and the `NN20()` functions which derive the NNXX parameters based on 50 millisecond and 20 millisecond threshold, you can use this function to apply custom temporal thresholds.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `threshold` (int): threshold for nnXX values in [ms].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `nnXX` (int): Number of NN interval differences greater than the specified threshold
- `pnnXX` (float): Ratio between nnXX and total number of NN interval differences

See also:

The `biosppy.utils.ReturnTuple` Object

Important: The XX in the nnXX and the pnnXX keys are replaced by the specified threshold.

For example, `nnXX(nni, threshold=30)` returns the custom `nn30` and `pnn30` parameters. Applying `threshold=35` as `nnXX(nni, threshold=35)` returns the custom `nn35` and `pnn35` parameters.

```
# Code example with a threshold of 30ms
results30 = nnXX(nni, threshold=30)
print(results30['nn30'])

# Code example with a threshold of 35ms
results35 = nnXX(nni, threshold=35)
print(results35['nn35'])
```

Exceptions

- `TypeError`: If no threshold is specified.
- `ValueError`: Threshold <= 0.

Parameter Computation

This parameter computes the NNI differences series from the NNI (`nni`) or (`rpeaks`) data and derives the NNXX parameter (`nnXX`) where it counts all the NNI differences that are greater than the specified threshold (`threshold`).

The `pnnXX` parameters is computed as follows:

$$pnnXX = \frac{nnXX}{n}$$

with:

- *pnnXX*: Ratio of NNI differences > threshold and *n*
- *nnXX*: Number of NNI differences > threshold XX
- *n*: Number of NNI differences

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

Specify the threshold of your preference using the `threshold` input parameter.

You can use NNI series (`nni`) to compute the `nnXX` parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute NNXX parameters using the NNI series and a threshold of 30ms
results30 = nnXX(nni, threshold=30)
print(results30['nn30'])

# Compute NNXX parameters using the NNI series and a threshold of 35ms
results35 = nnXX(nni, threshold=35)
print(results35['nn35'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNXX parameters using the R-peak series and a threshold of 30ms
results30 = nnXX(rpeaks=t[rpeaks], threshold=30)
print(results30['nn30'])

# Compute NNXX parameters using the R-peak series and a threshold of 35ms
results35 = nnXX(rpeaks=r[rpeaks], threshold=35)
print(results35['nn35'])
```

6.3.10 NN50: nn50()

```
pyhrv.time_domain.nn50(nni=None, rpeaks=None)
```

Function Description

Derives the NN50 parameters: Finds number of NN interval differences greater than 50ms (NN50) and the ratio between NN50 and the total number of NN interval differences.

Hint: Use the `nnXX()` function (*NNXX: `nnXX()`*) to compute NNXX parameters with custom thresholds.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `nn50` (int): Number of NN interval differences greater 50 milliseconds
- `pnn50` (float): Ratio between NN50 and total number of NN intervals

See also:

The `biosppy.utils.ReturnTuple` Object

Parameter Computation

This parameter computes the NNI differences series from the NNI (`nni`) or (`rpeaks`) data and derives the NN50 parameter (`nn50`) where it counts all the NNI differences that are greater than 50ms.

The `pnn50` parameters is computed as follows:

$$pNN50 = \frac{NN50}{n}$$

with:

- `pNNXX`: Ratio of NNI differences > 50 milliseconds and n
- `NNXX`: Number of NNI differences > 50 milliseconds
- n : Number of NNI differences

Note: This function computes the parameters using the `nnXX()` function (*NNXX: `nnXX()`*).

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format: `nn_format()`* for more information.

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the nn50 parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute NN50 parameters using the NNI series
results30 = nn50(nni)
print(results50['nn50'])
print(results50['pnn50'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NN50 parameters using the R-peak series
results30 = nn50(rpeaks=t[rpeaks])
print(results['nn50'])
print(results['pnn50'])
```

6.3.11 NN20: `nn20()`

`pyhrv.time_domain.nn20 (nni=None, rpeaks=None)`

Function Description

Derives the NN20 parameters: Finds number of NN interval differences greater than 20ms (NN20) and the ratio between NN20 and the total number of NN interval differences.

Hint: Use the `nnXX()` function (*NNXX: `nnXX()`*) to compute NNXX parameters with custom thresholds.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `nn20` (int): Number of NN interval differences greater 20 milliseconds
- `pNN20` (float): Ratio between NN20 and total number of NN intervals

See also:*The biosppy.utils.ReturnTuple Object***Parameter Computation**

This parameter computes the NNI differences series from the NNI (`nni`) or (`rpeaks`) data and derives the NN20 parameter (`nn20`) where it counts all the NNI differences that are greater than 20ms.

The `pnn20` parameters is computed as follows:

$$pNN20 = \frac{NN20}{n}$$

with:

- `pNNXX`: Ratio of NNI differences > 20 milliseconds and n
- `NNXX`: Number of NNI differences > 20 milliseconds
- n : Number of NNI differences

Note: This function computes the parameters using the `nnXX()` function (*NNXX: `nnXX()`*).

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format: `nn_format()`* for more information.

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the `nn20` parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute NN20 parameters using the NNI series
results = nn20(nni)
print(results['nn20'])
print(results['pnn20'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]
```

(continues on next page)

(continued from previous page)

```
# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NN20 parameters using the R-peak series
results = nn20(rpeaks=t[rpeaks])
print(results['nn20'])
print(results['pnn20'])
```

6.3.12 Geometrical Parameters

The geometrical parameters are computed based on the NNI histogram distribution. The TINN and Triangular Index are, in most cases, provided together. However, pyHRV provides individual functions to individually compute the TINN (`pyhrv.time_domain.tinn()`) and Triangular Index (`pyhrv.time_domain.triangular_index()`) parameters.

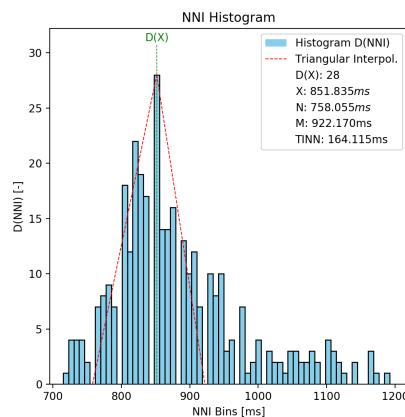
Additionally, the `pyhrv.time_domain.geometrical_parameters()` function allows you to compute all geometrical parameters and to join them in a single NNI histogram using only a single function.

TINN: `tinn()`

`pyhrv.time_domain.tinn(nni=None, rpeaks=None, binsize=7.8125, plot=True, show=True, figsize=None, legend=True)`

Function Description

This function fits an interpolated triangle to the NNI histogram and computes its baseline width. See *Parameter Computation* below for detailed information about the computation. As result, an NNI histogram (plot) as shown below is computed.



Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `binsize` (int, float, optional): Bin size of the histogram bins in [ms] (default: 7.8125 milliseconds).
- `plot` (bool, optional): If True, create the histogram plot figure using `matplotlib`. If False, the histogram data is computed using `numpy` with generating a histogram plot figure (default: True).

- `show` (bool, optional): If True, shows the histogram plot figure (default: True).
- `figsize` (array, optional): 2-element array with the `matplotlib` figure size `figsize`. Format: `figsize=(width, height)` (default: will be set to (6, 6) if input is None).
- `legend` (bool, optional): If True, adds legend to the histogram plot figure (default: True).

Note: The `binsize` is pre-defined at 7.8125ms and is determined from the minimum suitable sampling frequency for ECG signals of 128Hz as recommended by the [HRV Guidelines](#).

At this sampling frequency, the temporal resolution of the signal used to derive NNI series is limited at 7.8125ms (= 1/128Hz).

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `tinn_histogram` (matplotlib figure object): Histogram plot figure (only if input parameter `plot` is True)
- `tinn_n` (float): N value of the TINN computation (left corner of the interpolated triangle at (N, 0))
- `tinn_m` (float): M value of the TINN computation (right corner of the interpolated triangle at (M, 0))
- `tinn` (float): TINN (baseline width of the interpolated triangle) [ms]

See also:

The `biosppy.utils.ReturnTuple` Object

Parameter Computation

The TINN parameters are computed based on the interpolation of a triangle into the NNI distribution. The positioning of the triangle's edges are determined by the following procedure: The first edge is positioned at the point $(D(X), X)$ with $D(X)$ being the histogram's maximum and X the bin containing the maximum. The other two edges are positioned at the points $(N, 0)$ and $(M, 0)$. Finally, N and M are determined by finding the interpolated triangle with the best fit to the NNI histogram using the least squares method, as presented by the following formula:

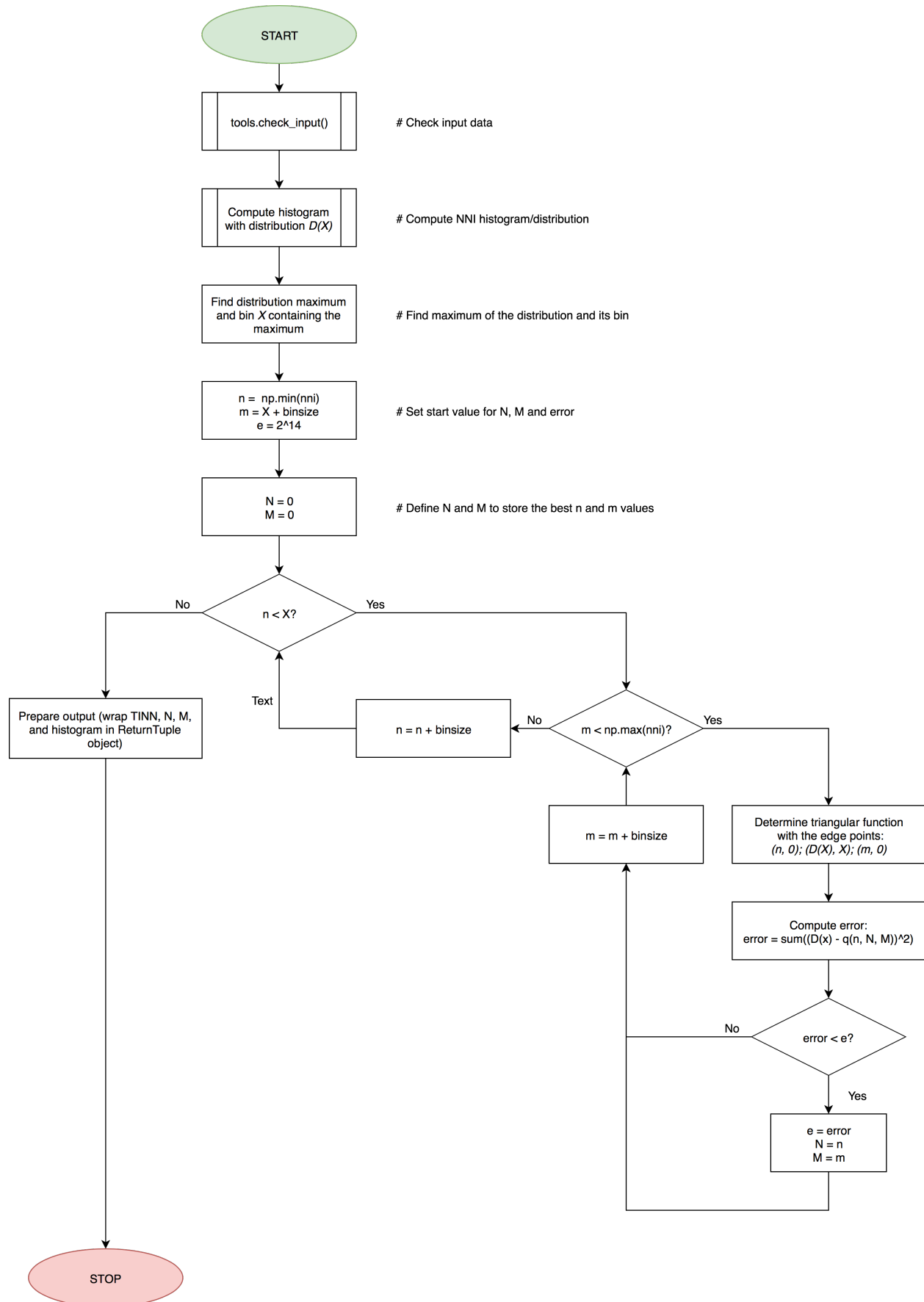
$$E(n, N, M) = \min \sum_{N_{min}}^{M_{max}} (D(X) - q(n, N, M))^2$$

with:

- $E(n)$: Error of the triangular interpolation with the best fit to the distribution
- $D(X)$: NNI distribution
- $q(n, N, m)$: Triangular interpolation function
- n : Bin
- N : N value determining the left corner of the interpolated triangle
- N_{min} : Lowest bin where $D(x) \neq 0$
- M : M value determining the right corner of the interpolated triangle
- M_{min} : Highest bin where $D(x) \neq 0$

The main flow of this function is presented in the following flowchart:

Application Notes



It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Use the `legend` input parameter to show or hide the legend in the histogram figure.

The `show` parameter only has effect if `plot` is set to `True`. If `plot` is `False`, no plot figure will be generated, therefore, no figure can be shown using the `show` parameter.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the TINN parameters:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute TINN parameters using the NNI series
results = td.tinn(nni)

# Print TINN and th N value
print(results['tinn'])
print(results['tinn_n'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
```

(continues on next page)

(continued from previous page)

```
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

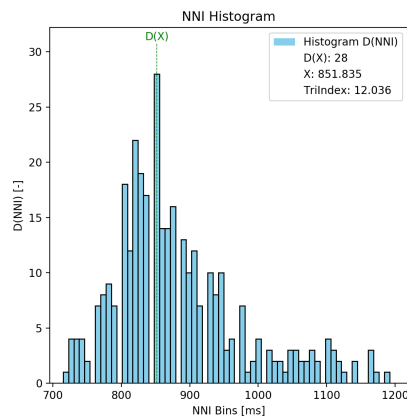
# Compute TINN parameters using the R-peak series
results = td.tinn(rpeaks=t[rpeaks])
```

Triangular Index: `triangular_index()`

```
pyhrv.time_domain.triangular_index(nni=None, rpeaks=None, binsize=7.8125, plot=True,
                                   show=True, figsize=None, legend=True)
```

Function Description

Computes the triangular index based on the NN interval histogram.



Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `binsize` (int, float, optional): Bin size of the histogram bins (default: 7.8125ms).
- `plot` (bool, optional): If True, create the histogram plot figure using `matplotlib`. If False, the histogram data is computed using `numpy` with generating a histogram plot figure (default: True).
- `show` (bool, optional): If True, shows the histogram plot figure (default: True).
- `figsize` (array, optional): 2-element array with the `matplotlib` figure size `figsize`. Format: `figsize=(width, height)` (default: will be set to (6, 6) if input is None).
- `legend` (bool, optional): If True, adds legend to the histogram plot figure (default: True).

Note: The `binsize` is pre-defined at 7.8125ms and is determined from the minimum suitable sampling frequency for ECG signals of 128Hz as recommended by the [HRV Guidelines](#).

At this sampling frequency, the temporal resolution of the signal used to derive NNI series is limited at 7.8125ms (= 1/128Hz).

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object (see also [The biosppy.utils.ReturnTuple Object](#)). Use the following keys below (on the left) to index the results.

- `tri_histogram` (matplotlib figure object): Histogram figure (only if input parameter 'plot' is True).
- `tri_index` (double): Triangular index.

See also:

[The biosppy.utils.ReturnTuple Object](#)

Parameter Computation

The Triangular Index is computed as the ratio between the total number of NNIs and the maximum of the NNI histogram distribution ($D(x)$).

$$Tri = \frac{n}{D(X)}$$

with:

- *Tri*: Triangular index
- *n*: Number of NNI
- $D(X)$: Maximum of the NNI distribution

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section [NN Format: nn_format\(\)](#) for more information.

Use the `legend` input parameter to show or hide the legend in the histogram figure.

The `show` parameter only has effect if `plot` is set to True. If `plot` is False, no plot figure will be generated, therefore, no figure can be shown using the `show` parameter.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the Triangular Index:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute Triangular Index using the NNI series
results = td.triangular_index(nni)

# Print Triangular Index
print(results['tri_index'])
```

Alternatively, you can use R-peak series (rpeaks):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

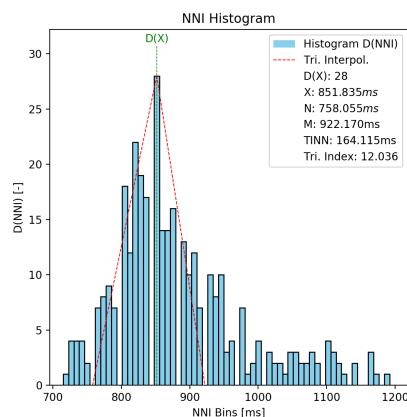
# Compute Triangular Index using the R-peak series
results = td.triangular_index(rpeaks=t[rpeaks])
```

Geometrical Parameters Function: `geometrical_parameters()`

```
pyhrv.time_domain.geometrical_parameters(nni=None, rpeaks=None, binsize=7.8125,
                                          plot=True, show=True, figsize=None, legend=True,
                                          end=True)
```

Function Description

Computes all the geometrical parameters based on the NNI histogram (Triangular Index, TINN, N, M) and returns them in a single histogram plot figure.



Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `binsize` (int, float, optional): Bin size of the histogram bins (default: 7.8125ms).
- `plot` (bool, optional): If True, create the histogram plot figure using `matplotlib`. If False, the histogram data is computed using `numpy` with generating a histogram plot figure (default: True).
- `show` (bool, optional): If True, shows the histogram plot figure (default: True).
- `figsize` (array, optional): 2-element array with the `matplotlib` figure size `figsize`. Format: `figsize=(width, height)` (default: will be set to (6, 6) if input is None).
- `legend` (bool, optional): If True, adds legend to the histogram plot figure (default: True).

Note: The `binsize` is pre-defined at 7.8125ms and is determined from the minimum suitable sampling frequency for ECG signals of 128Hz as recommended by the [HRV Guidelines](#).

At this sampling frequency, the temporal resolution of the signal used to derive NNI series is limited at 7.8125ms (= 1/128Hz).

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object (see also [The biosppy.utils.ReturnTuple Object](#)). Use the following keys below (on the left) to index the results.

- `nn_histogram` (matplotlib figure object): Histogram plot figure (only if input parameter `plot` is True)
- `tinn_n` (float): N value of the TINN computation (left corner of the interpolated triangle at (N, 0))
- `tinn_m` (float): M value of the TINN computation (right corner of the interpolated triangle at (M, 0))
- `tinn` (float): TINN (baseline width of the interpolated triangle) [ms]
- `tri_index` (float): Triangular index [ms]

See also:

[The biosppy.utils.ReturnTuple Object](#)

Parameter Computation

See *TINN*: [tinn\(\)](#) and *Triangular Index*: [triangular_index\(\)](#) for detailed information.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: [nn_format\(\)](#) for more information.

Use the `legend` input parameter do show or hide the legend in the histogram figure.

The `show` parameter only has effect if `plot` is set to True. If `plot` is False, no plot figure will be generated, therefore, no figure can be shown using the `show` parameter.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following examples demonstrate how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the Triangular Index:

```
# Import packages
import pyhrv
import pyhrv.time_domain as td

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute Geometrical Parameters using the NNI series
results = td.geometrical_parameters(nni)

# Print Geometrical Parameters
print(results['tri_index'])
print(results['tinn'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute Geometrical Parameters using the R-peak series
results = td.geometrical_parameters(rpeaks=t[rpeaks])
```

6.3.13 Domain Level Function: `time_domain()`

```
pyhrv.time_domain.time_domain()
```

Function Description

Computes all time domain parameters of the HRV Time Domain module and returns them in a single `ReturnTuple` object.

See also:

The individual parameter functions of this module for more detailed information about the computed parameters.

Input Parameters

- `signal` (array): ECG signal
- `nni` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]
- `sampling_rate` (int, float, optional): Sampling rate in [Hz] used for the ECG acquisition (default: 1000Hz)
- `threshold` (int, optional): Custom threshold in [ms] for the optional NNXX and pNNXX parameters (default: None)
- `plot` (bool, optional): If True, creates histogram using matplotlib, else uses NumPy for histogram data only (geometrical parameters, default: True)
- `show` (bool, optional): If True, shows histogram plots.
- `binsize` (float, optional): Bin size in [ms] of the histogram bins - (geometrical params, default: 7.8125ms).

Important: This function computes the Time Domain parameters using either the `signal`, `nni`, or `rpeaks` data. Provide only one type of data, as it is not required to pass all three types at once.

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `nni_counter` (int): Number of NNI (-)
- `nni_mean` (float): Mean NNI [ms]
- `nni_min` (int): Minimum NNI [ms]
- `nni_max` (int): Maximum NNI [ms]
- `nni_diff_mean` (float): Mean NNI difference [ms]
- `nni_diff_min` (int): Minimum NNI difference [ms]
- `nni_diff_max` (int): Maximum NNI difference [ms]
- `hr_mean` (float): Mean heart rate [bpm]
- `hr_min` (int): Minimum heart rate [bpm]
- `hr_max` (int): Maximum heart rate [bpm]
- `hr_std` (float): Standard deviation of the heart rate series [bpm]
- `sdnn` (float): Standard deviation of NN intervals [ms]
- `sdnn_index` (float): SDNN Index [ms]
- `sdann` (float): SDANN [ms]
- `rmssd` (float): Root mean of squared NNI differences [ms]
- `sdsd` (float): Standard deviation of NNI differences [ms]
- `nnXX` (int, optional): Number of NN interval differences greater than the specified threshold (-)
- `pnnXX` (float, optional): Ratio between nnXX and total number of NN interval differences (-)
- `nn50` (int): Number of NN interval differences greater 50ms

- `pnn50` (float): Ratio between NN50 and total number of NN intervals [ms]
- `nn20` (int): Number of NN interval differences greater 20ms
- `pnn20` (float): Ratio between NN20 and total number of NN intervals [ms]
- `nn_histogram` (matplotlib figure object): Histogram plot figure (only if input parameter `plot` is True)
- `tinn_n` (float): N value of the TINN computation (left corner of the interpolated triangle at (N, 0))
- `tinn_m` (float): M value of the TINN computation (right corner of the interpolated triangle at (M, 0))
- `tinn` (float): TINN (baseline width of the interpolated triangle) [ms]
- `tri_index` (float): Triangular index [ms]

Important: The XX in the `nnXX` and the `pnnXX` keys are substituted by the specified threshold.

For instance, `nnXX(nni, threshold=30)` returns the custom `nn30` and `pnn30` parameters. Applying `threshold=35` as `nnXX(nni, threshold=35)` returns the custom `nn35` and `pnn35` parameters.

These parameters are only returned if a custom threshold (`threshold`) has been defined in the input parameters.

See also:

The `biosppy.utils.ReturnTuple` Object

Application Notes

It is not necessary to provide input data for `signal`, `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`signal`, `nni` **or** `rpeaks`). The input data will be prioritized in the following order, in case multiple inputs are provided:

1. `signal`, 2. `nni`, 3. `rpeaks`.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- *[What may help when matplotlib blocks your code from being executed](#)*
 - *[More information about the matplotlib Interactive Mode](#)*
 - *[More information about matplotlib Backends](#)*
-

Examples & Tutorials & Tutorials

The following example codes demonstrate how to use the `time_domain()` function.

You can choose either the ECG signal, the NNI series or the R-peaks as input data for the PSD estimation and parameter computation:

```
# Import packages
import biosppy
import pyhrv.time_domain as td
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNI series
nni = tools.nn_intervals(t[rpeaks])

# OPTION 1: Compute Time Domain parameters using the ECG signal
signal_results = td.time_domain(signal=filtered_signal)

# OPTION 2: Compute Time Domain parameters using the R-peak series
rpeaks_results = td.time_domain(rpeaks=t[rpeaks])

# OPTION 3: Compute Time Domain parameters using the NNI-series
nni_results = td.time_domain(nni=nni)
```

The output of of all three options above will be the same.

Note: If an ECG signal is provided, the signal will be filtered and the R-peaks will be extracted using the `biosppy.signals.ecg.ecg()` function. Finally, the NNI series for the PSD estimation will be computed from the extracted R-peak series.

See also:

`biosppy.signals.ecg.ecg()`

You can now access the parameters using the output parameter keys (works the same for the `rpeaks_results` and `nni_results`):

```
# Print SDNN
print(signal_results['sdnn'])

# Print RMSSD
print(signal_results['rmssd'])
```

6.4 Frequency Domain Module

The *Frequency Domain Module* contains all functions to compute the frequency domain parameters derived from the PSD estimation using the Welch's method, the Lomb-Scargle Periodogram and the Autoregressive method.

See also:

[pyHRV Frequency Domain Module source code](#)

Module Contents

- *Frequency Domain Module*
 - *Welch's Method:* `welch_psd()`
 - *Lomb-Scargle Periodogram:* `lomb_psd()`
 - *Autoregressive Method:* `ar_psd()`
 - *2D PSD Comparison Plot:* `psd_comparison()`
 - *3D PSD Waterfall Plot:* `psd_waterfall()`
 - *Domain Level Function:* `frequency_domain()`
 - *Frequency Parameters*

See also:

- [Sample NNI Series \(docs\)](#)
- [Sample NNI Series on GitHub](#)
- `series_1.npy` (file used in the examples below)

6.4.1 Welch's Method: `welch_psd()`

```
pyhrv.frequency_domain.welch_psd(nn=None, rpeaks=None, fbands=None, nfft=2**12,
                                   detrend=True, window='hamming', show=True,
                                   show_param=True, legend=True)
```

Function Description

Computes a Power Spectral Density (PSD) estimation from the NNI series using the Welch's method and computes all frequency domain parameters from this PSD according to the specified frequency bands.

If no frequency bands are specified, the default frequency band limits for the *Very Low Frequency (VLF)*, *Low Frequency (LF)*, and *High Frequency (HF)* bands as recommended by the [HRV Guidelines](#) are applied:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

Use the `fbands` parameter to specify custom frequency bands and the possibility to add the *Ultra Low Frequency (ULF)* band (see **Application Notes & Examples & Tutorials** below for more information).

The following parameters are computed from the PSD and the specified frequency bands:

- Peak frequencies [Hz]
- Absolute powers [ms²]
- Relative powers [ms²]
- Logarithmic powers [log]
- Normalized powers (LF & HF only)[-]
- Total power of all frequency bands [ms²]

An example of a PSD plot generated by this function can be seen [here](#):

Input Parameters

- `nn` (array): NN intervals in [ms] or [s]

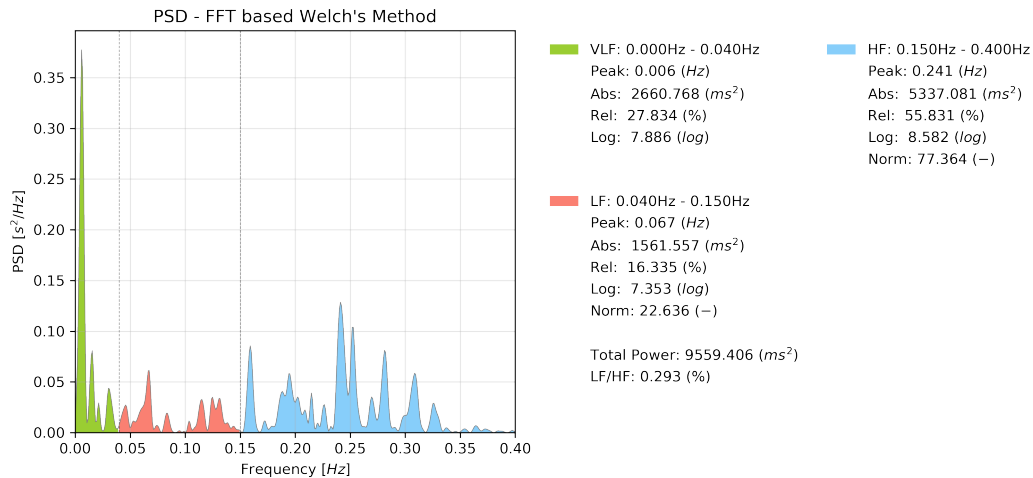


Fig. 2: Sample Welch PSD.

- `rpeaks` (array): R-peak times in [ms] or [s]
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `nfft` (int, optional): Number of points computed for the FFT result (default: 2×12)
- `detrend` (bool, optional): If True, detrend NNI series by subtracting the mean NNI (default: True)
- `window` (scipy.window function, optional): Window function used for PSD estimation (default: 'hamming')
- `show` (bool, optional): If True, show PSD plot figure (default: True)
- `show_param` (bool, optional): If true, list all computed PSD parameters next to the plot (default: True)
- `legend` (bool, optional): If true, add a legend with frequency bands to the plot (default: True)

Note: If `fbands` is none, the default values for the frequency bands will be set.

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below for more information on how to define custom frequency bands.

Important: The specified `nfft` refers to the overall number of samples computed for the entire PSD estimation regardless of frequency bands, i.e. the number of samples within the lowest and the highest frequency band limit is not necessarily equal to the specified `nfft`.

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the keys below (on the left) to index the results:

- `fft_peak` (tuple): Peak frequencies of all frequency bands [Hz]

- `fft_abs` (tuple): Absolute powers of all frequency bands [ms^2]
- `fft_rel` (tuple): Relative powers of all frequency bands [%]
- `fft_log` (tuple): Logarithmic powers of all frequency bands [log]
- `fft_norm` (tuple): Normalized powers of the LF and HF frequency bands [-]
- `fft_ratio` (float): LF/HF ratio [-]
- `fft_total` (float): Total power over all frequency bands [ms^2]
- `fft_interpolation` (str): Interpolation method used for NNI interpolation (hard-coded to 'cubic')
- `fft_resampling_frequency` (int): Resampling frequency used for NNI interpolation [Hz] (hard-coded to 4Hz as recommended by the [HRV Guidelines](#))
- `fft_window` (str): Spectral window used for PSD estimation of the Welch's method
- `fft_plot` (matplotlib figure object): PSD plot figure object

See also:

The `biosppy.utils.ReturnTuple` Object

Computation Method

This functions computes the PSD estimation using the `scipy.signals.lomb()` ([docs](#), [source](#)) function.

The flowchart below visualizes the structure of this function. The NNI series are interpolated at a new sampling frequency of 4Hz before the PSD computation as per the HRV guidelines.

See also:

Section `ref-freqparams` for detailed information about the computation of the individual parameters.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

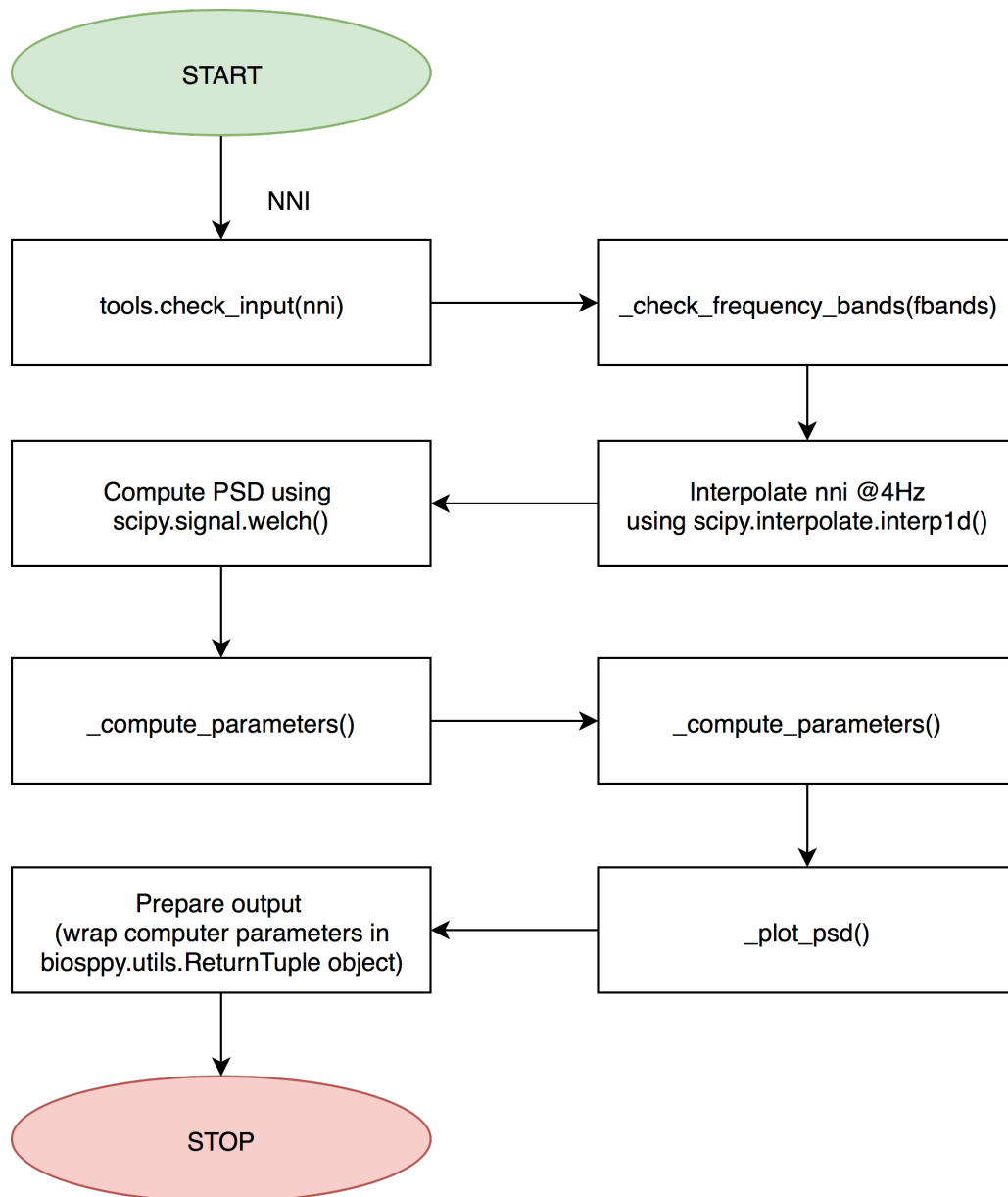
```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

This issue is shown in the following PSD plot where the corrected frequency bands above were used and there is no frequency band covering the range between 0.2Hz and 0.25Hz:

Fig. 3: Flowchart of the `welch_psd()` function.

The resampling frequency and the interpolation methods used for this method are hardcoded to 4Hz and the cubic spline interpolation of the `scipy.interpolate.interp1d()` ([docs](#), [source](#)) function.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the PSD:

```
# Import packages
import pyhrv
import pyhrv.frequency_domain as fd

# Load NNI sample series
pyhrv.utils.load_sample_nni()

# Compute the PSD and frequency domain parameters using the NNI series
result = fd.welch_psd(nni)

# Access peak frequencies using the key 'fft_peak'
print(result['fft_peak'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute the PSD and frequency domain parameters using the R-peak series
result = fd.welch_psd(rpeaks=t[rpeaks])
```

The plot of these examples should look like the following plot:

If you want to specify custom frequency bands, define the limits in a Python dictionary as shown in the following example:

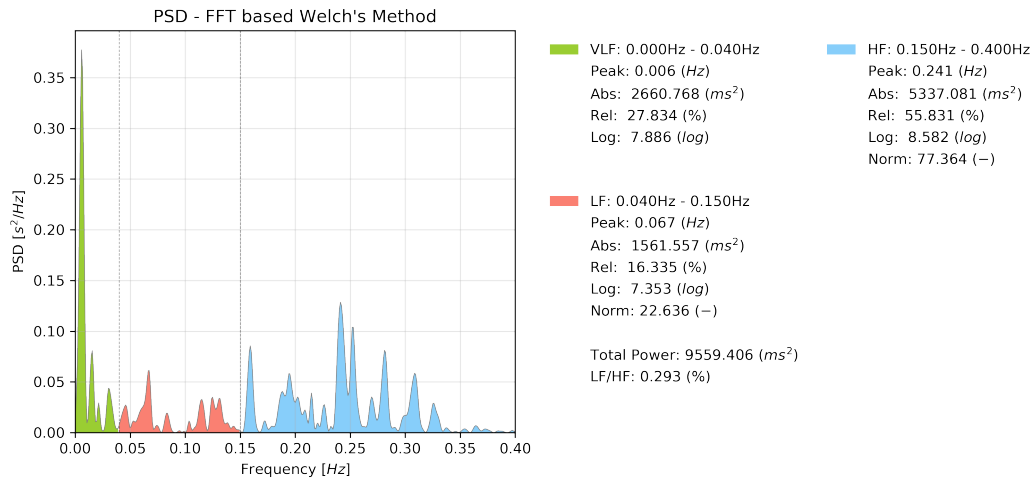


Fig. 5: Welch PSD with default frequency bands.

```
# Define custom frequency bands and add the ULF band
fbands = {'ulf': (0.0, 0.1), 'vlf': (0.1, 0.2), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}

# Compute the PSD with custom frequency bands
result = fd.welch_psd(nni, fbands=fbands)
```

The plot of this example should look like the following plot:

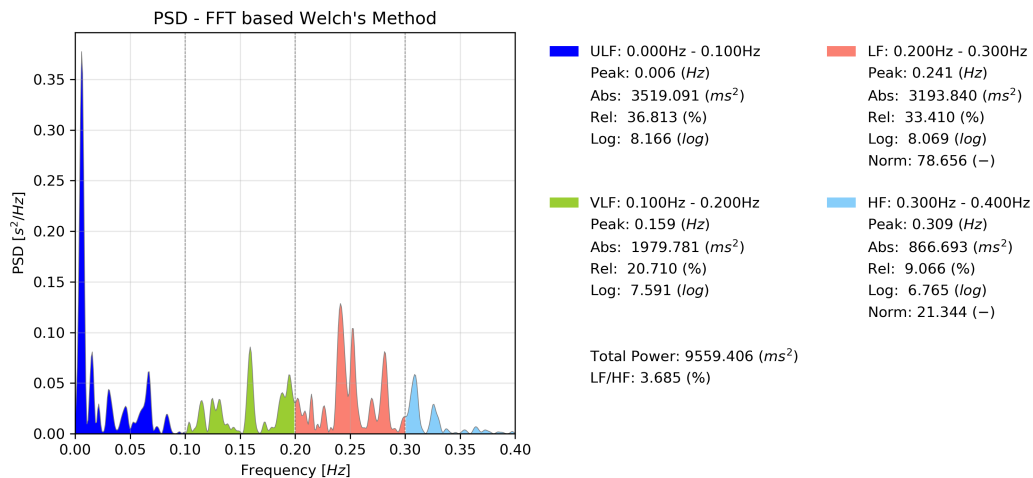


Fig. 6: Welch PSD with custom frequency bands.

By default, the figure will contain the PSD plot on the left and the computed parameter results on the right side of the figure. Set the `show_param` to `False` if only the PSD is needed in the figure.

```
# Compute the PSD without the parameters being shown on the right side of the figure
result = fd.welch_psd(nni, show_param=False)
```

The plot for this example should look like the following plot:

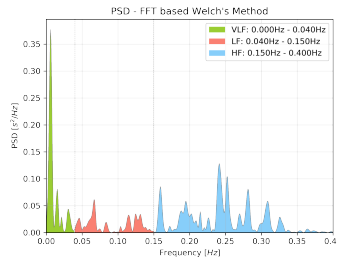


Fig. 7: PSD plot without parameters.

6.4.2 Lomb-Scargle Periodogram: `lomb_psd()`

```
pyhrv.frequency_domain.lomb_psd(nn=None, rpeaks=None, fbands=None, nfft=2**8,
                                ma_size=None, show=True, show_param=True, leg-
                                end=True)
```

Function Description

Computes a Power Spectral Density (PSD) estimation from the NNI series using the Lomb-Scargle Periodogram and computes all frequency domain parameters from this PSD according to the specified frequency bands.

If no frequency bands are specified, the default frequency band limits for the *Very Low Frequency (VLF)*, *Low Frequency (LF)*, and *High Frequency (HF)* bands as recommended by the [HRV Guidelines](#) are applied:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

Use the `fbands` parameter to specify custom frequency bands and the possibility to add the *Ultra Low Frequency (ULF)* band (see [Application Notes & Examples & Tutorials](#) below for more information).

The following parameters are determined from the PSD and the specified frequency bands:

- Peak frequencies [Hz]
- Absolute powers [ms²]
- Relative powers [ms²]
- Logarithmic powers [log]
- Normalized powers (LF & HF only)[-]
- Total power of all frequency bands [ms²]

An example of a PSD plot generated by this function can be seen here:

Input Parameters

- `nn` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `nfft` (int, optional): Number of points computed for the Lomb-Scargle result (default: 2**8)
- `ma_order` (int, optional): Order of the moving average filter (default: None; no filter applied)
- `show` (bool, optional): If True, show PSD plot figure (default: True)

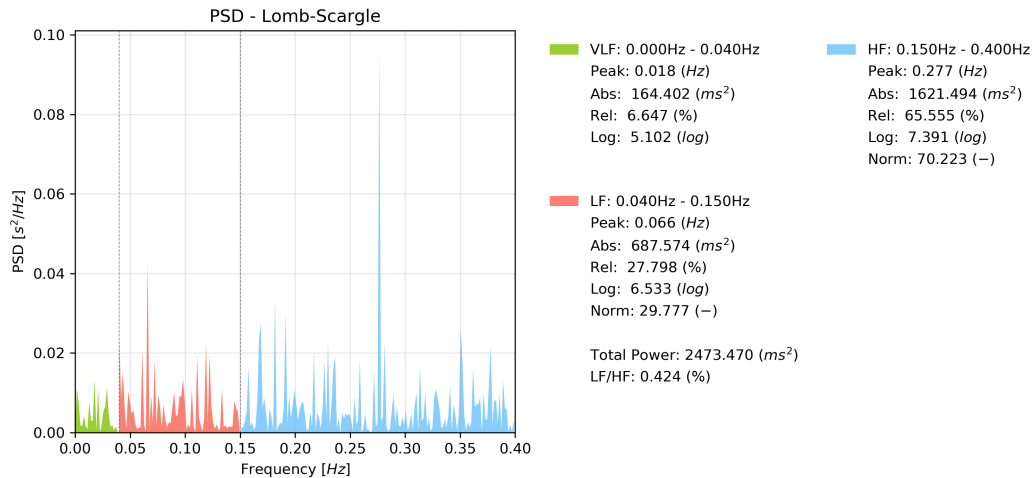


Fig. 8: Sample Lomb PSD.

- `show_param` (bool, optional): If true, list all computed PSD parameters next to the plot (default: True)
- `legend` (bool, optional): If true, add a legend with frequency bands to the plot (default: True)

Note: If `fbands` is none, the default values for the frequency bands will be set:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below to learn how to specify custom frequency bands.

Important: The specified `nfft` refers to the overall number of samples computed for the entire PSD estimation regardless of frequency bands, i.e. the number of samples within the lowest and the highest frequency band limit is not necessarily equal to the specified `nfft`.

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `lomb_peak` (tuple): Peak frequencies of all frequency bands [Hz]
- `lomb_abs` (tuple): Absolute powers of all frequency bands [ms²]
- `lomb_rel` (tuple): Relative powers of all frequency bands [%]
- `lomb_log` (tuple): Logarithmic powers of all frequency bands [log]
- `lomb_norm` (tuple): Normalized powers of the LF and HF frequency bands [-]
- `lomb_ratio` (float): LF/HF ratio [-]
- `lomb_total` (float): Total power over all frequency bands [ms²]
- `lomb_ma` (int): Moving average filter order [-]

- `lomb_plot` (matplotlib figure object): PSD plot figure object

See also:

The `biosppy.utils.ReturnTuple` Object

Computation Method

This functions computes the PSD estimation using the `scipy.signals.lombscargle()` ([docs](#) , [source](#)) function.

See also:

Section `ref-freqparams` for detailed information about the computation of the individual parameters.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

This issue is shown in the following PSD plot where the corrected frequency bands above were used and there is no frequency band covering the range between 0.2Hz and 0.25Hz:

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()` function.

See also:

- *What may help when matplotlib blocks your code from being executed*
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the PSD:

```
# Import packages
import pyhrv
import pyhrv.frequency_domain as fd

# Load NNI sample series
pyhrv.utils.load_sample_nni()

# Compute the PSD and frequency domain parameters using the NNI series
result = fd.lomb_psd(nni)

# Access peak frequencies using the key 'lomb_peak'
print(result['lomb_peak'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute the PSD and frequency domain parameters using the R-peak series
result = fd.lomb_psd(rpeaks=t[rpeaks])
```

The plot of these examples should look like the following plot:

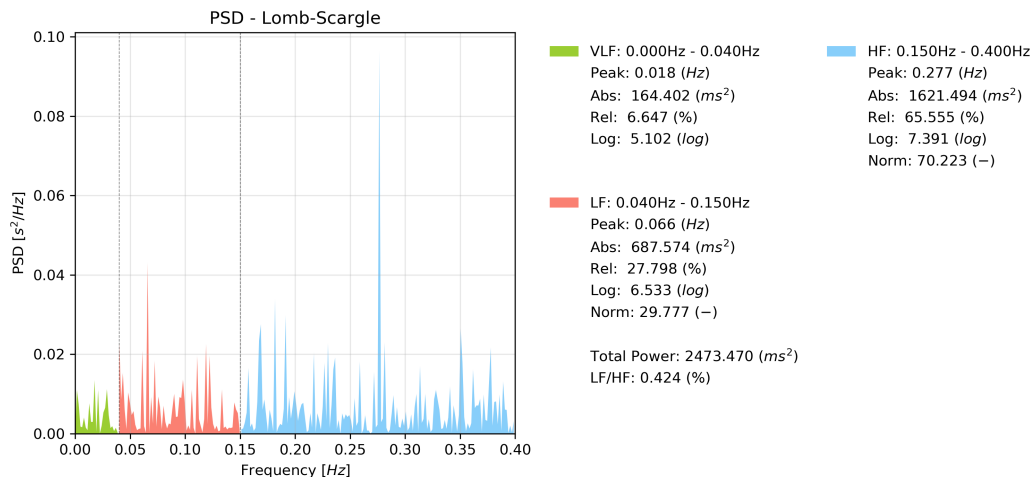


Fig. 10: Lomb PSD with default frequency bands.

If you want to specify custom frequency bands, define the limits in a Python dictionary as shown in the following example:

```
# Define custom frequency bands and add the ULF band
fbands = {'ulf': (0.0, 0.1), 'vlf': (0.1, 0.2), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}

# Compute the PSD with custom frequency bands
result = fd.lomb_psd(nni, fbands=fbands)
```

The plot of this example should look like the following plot:

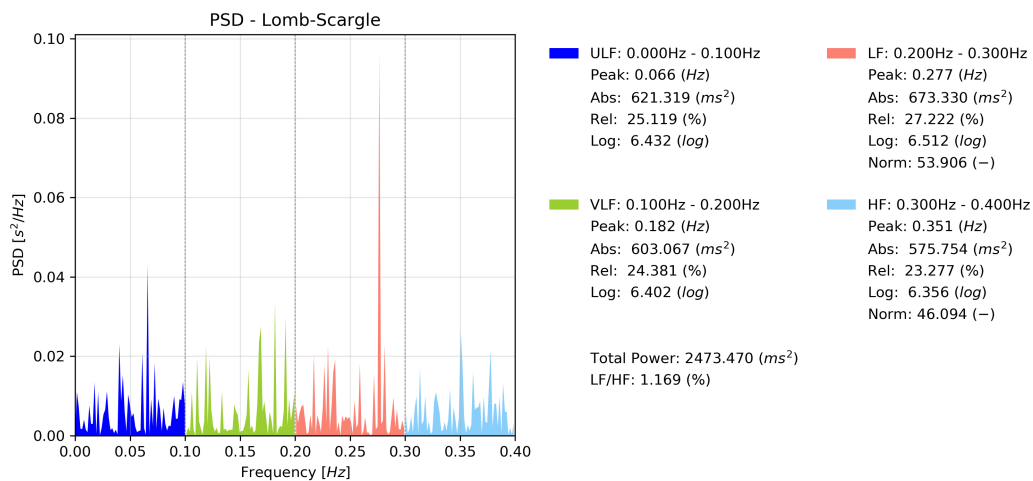


Fig. 11: Lomb PSD with custom frequency bands.

By default, the figure will contain the PSD plot on the left and the computed parameter results on the right side of the figure. Set the `show_param` to `False` if only the PSD is needed in the figure.

```
# Compute the PSD without the parameters being shown on the right side of the figure
result = fd.lomb_psd(nni, show_param=False)
```

The plot for this example should look like the following plot:

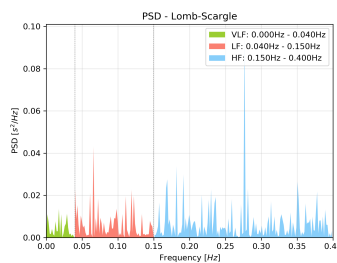


Fig. 12: Lomb PSD without parameters.

6.4.3 Autoregressive Method: `ar_psd()`

```
pyhrv.frequency_domain.ar_psd(nn=None, rpeaks=None, fbands=None, nfft=2**12, order=16,
                               show=True, show_param=True, legend=True)
```

Function Description

Computes a Power Spectral Density (PSD) estimation from the NNI series using the Autoregressive method and computes all frequency domain parameters from this PSD according to the specified frequency bands.

If no frequency bands are specified, the default frequency band limits for the *Very Low Frequency (VLF)*, *Low Frequency (LF)*, and *High Frequency (HF)* bands as recommended by the [HRV Guidelines](#) are applied:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

Use the `fbands` parameter to specify custom frequency bands and the possibility to add the *Ultra Low Frequency (ULF)* band (see **Application Notes & Examples & Tutorials** below for more information).

The following parameters are computed from the PSD and the specified frequency bands:

- Peak frequencies [Hz]
- Absolute powers [ms^2]
- Relative powers [ms^2]
- Logarithmic powers [log]
- Normalized powers (LF & HF only)[-]
- Total power of all frequency bands [ms^2]

An example of a PSD plot generated by this function can be seen here:

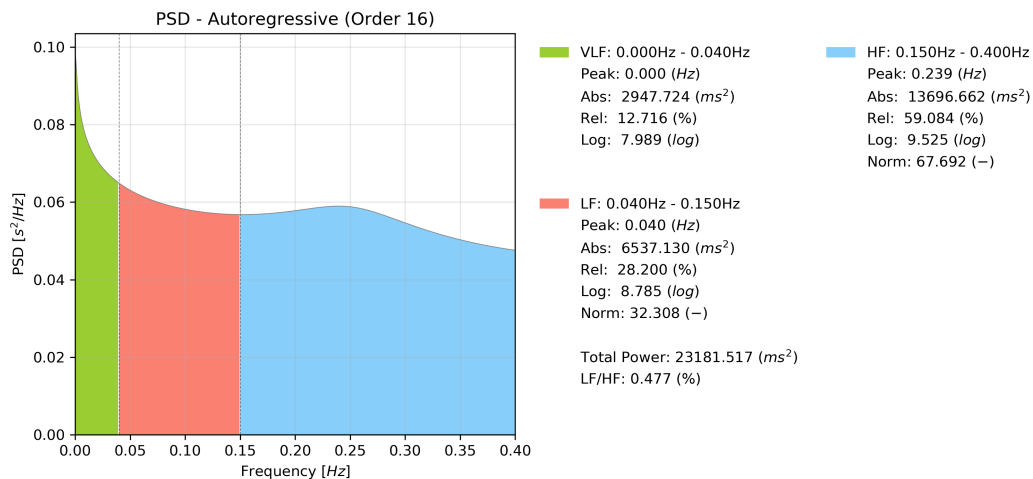


Fig. 13: Sample Autoregressive PSD.

Input Parameters

- `nn` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `nfft` (int, optional): Number of points computed for the FFT result (default: 2×12)
- `order` (int, optional): Autoregressive model order (default: 16)
- `show` (bool, optional): If True, show PSD plot figure (default: True)

- `show_param` (bool, optional): If true, list all computed PSD parameters next to the plot (default: True)
- `legend` (bool, optional): If true, add a legend with frequency bands to the plot (default: True)

Note: If `fbands` is none, the default values for the frequency bands will be set.

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below for more information on how to define custom frequency bands.

Important: The specified `nfft` refers to the overall number of samples computed for the entire PSD estimation regardless of frequency bands, i.e. the number of samples within the lowest and the highest frequency band limit is not necessarily equal to the specified `nfft`.

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `ar_peak` (tuple): Peak frequencies of all frequency bands [Hz]
- `ar_abs` (tuple): Absolute powers of all frequency bands [ms^2]
- `ar_rel` (tuple): Relative powers of all frequency bands [%]
- `ar_log` (tuple): Logarithmic powers of all frequency bands [log]
- `ar_norm` (tuple): Normalized powers of the LF and HF frequency bands [-]
- `ar_ratio` (float): LF/HF ratio [-]
- `ar_total` (float): Total power over all frequency bands [ms^2]
- `ar_interpolation` (str): Interpolation method used for NNI interpolation (hard-coded to 'cubic')
- `ar_resampling_frequency` (int): Resampling frequency used for NNI interpolation [Hz] (hard-coded to 4Hz as recommended by the [HRV Guidelines](#))
- `ar_window` (str): Spectral window used for PSD estimation of the Welch's method
- `ar_order` (int): Autoregressive model order
- `ar_plot` (matplotlib figure object): PSD plot figure object

See also:

The `biosppy.utils.ReturnTuple` Object

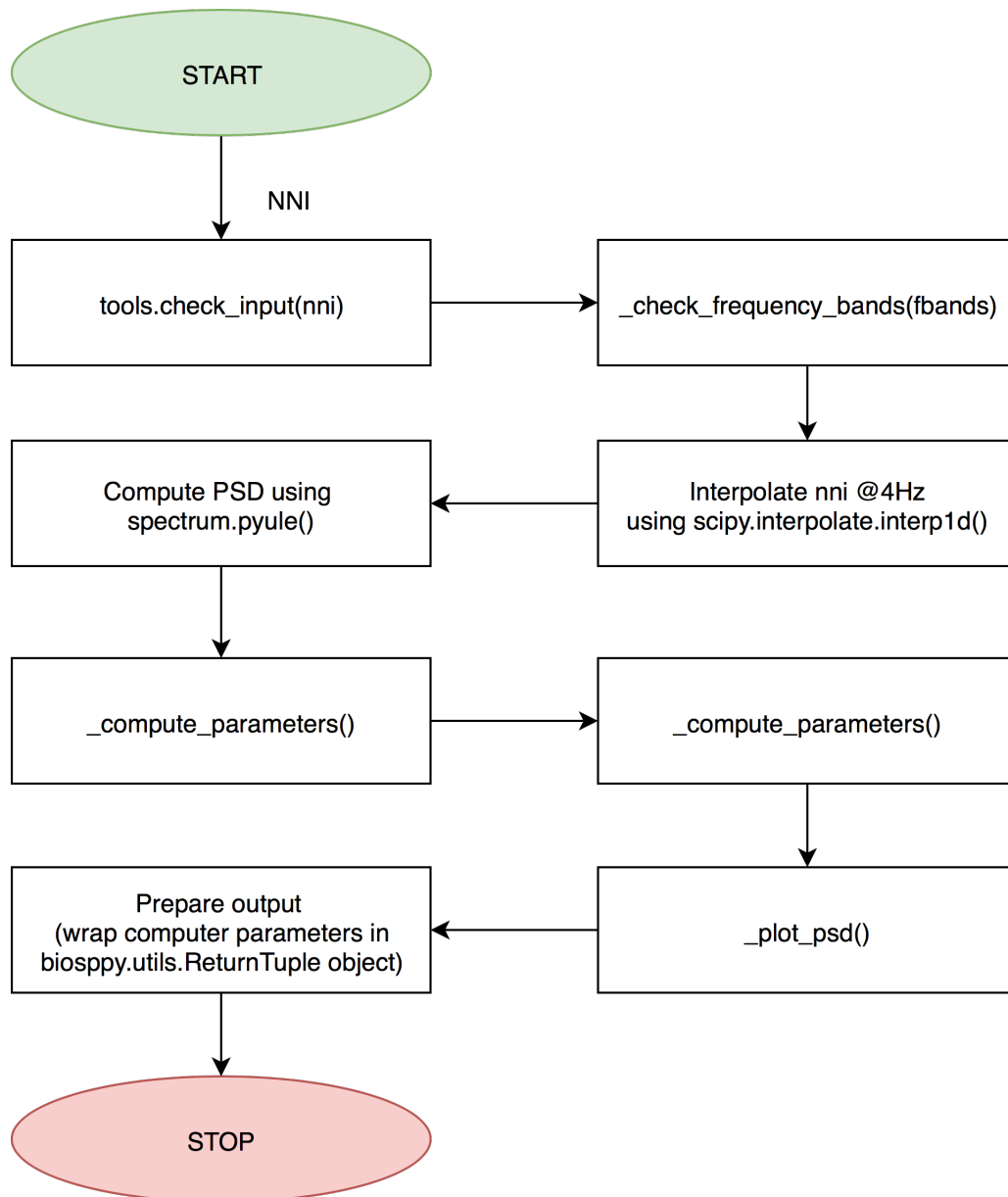
Computation Method

This functions computes the PSD estimation using the `spectrum.pyule()` ([docs](#), [source](#)) function.

The flowchart below visualizes the structure of the `ar_psd()` function. The NNI series are interpolated at a new sampling frequency of 4Hz before the PSD computation is conducted as the unevenly sampled NNI series would distort the PSD.

See also:

Section `ref-freqparams` for detailed information about the computation of the individual parameters.

Fig. 14: Flowchart of the `ar_psd()` function.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

This issue is shown in the following PSD plot where the corrected frequency bands above were used and there is no frequency band covering the range between 0.2Hz and 0.25Hz:

The resampling frequency and the interpolation methods used for this method are hardcoded to 4Hz and the cubic spline interpolation of the `scipy.interpolate.interpld()` ([docs](#), [source](#)) function.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()` function.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
- [More information about the matplotlib Interactive Mode](#)
- [More information about matplotlib Backends](#)

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the PSD:

```
# Import packages
import pyhrv
import pyhrv.frequency_domain as fd
```

(continues on next page)

(continued from previous page)

```
# Load NNI sample series
pyhrv.utils.load_sample_nni()

# Compute the PSD and frequency domain parameters using the NNI series
result = fd.ar_psd(nni)

# Access peak frequencies using the key 'ar_peak'
print(result['ar_peak'])
```

Alternatively, you can use R-peak series (rpeaks):

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute the PSD and frequency domain parameters using the R-peak series
result = fd.ar_psd(rpeaks=t[rpeaks])
```

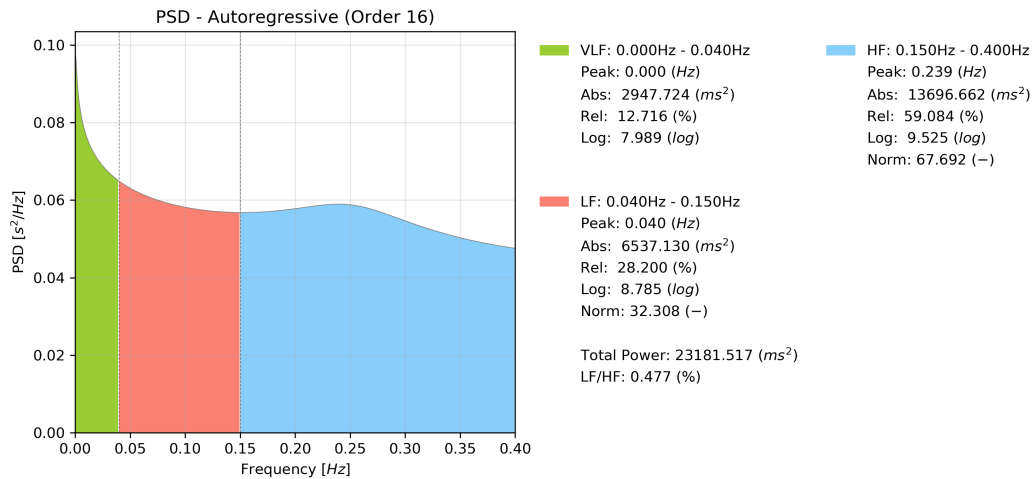


Fig. 16: Autoregressive PSD with default frequency bands.

If you want to specify custom frequency bands, define the limits in a Python dictionary as shown in the following example:

```
# Define custom frequency bands and add the ULF band
fbands = {'ulf': (0.0, 0.1), 'vlf': (0.1, 0.2), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}

# Compute the PSD with custom frequency bands
result = fd.ar_psd(nni, fbands=fbands)

# Access peak frequencies using the key 'ar_peak'
print(result['ar_peak'])
```

The plot of this example should look like the following plot:

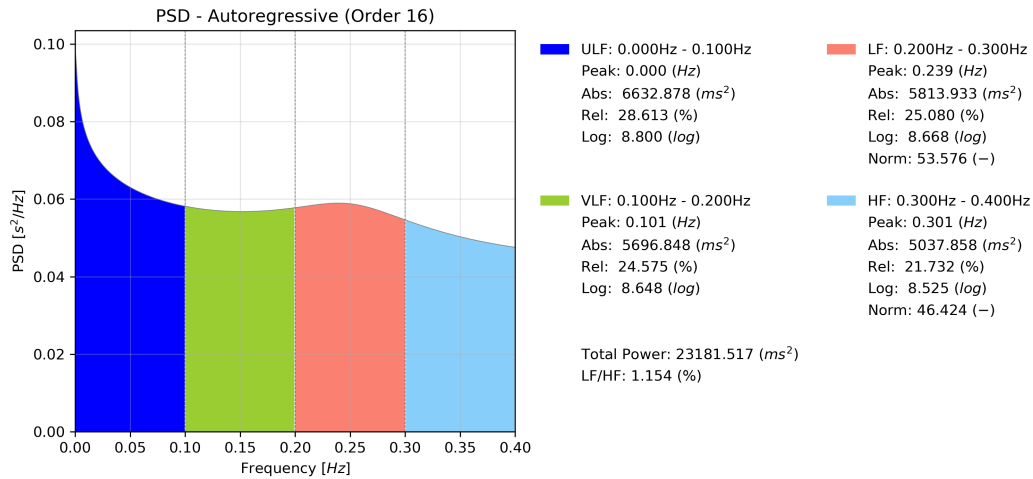


Fig. 17: Autoregressive PSD with custom frequency bands.

By default, the figure will contain the PSD plot on the left and the computed parameter results on the left side of the figure. Set the `show_param` to `False` if only the PSD is needed in the figure.

```
# Compute the PSD without the parameters being shown on the right side of the figure
result = fd.ar_psd(nni, show_param=False)

# Access peak frequencies using the key 'ar_peak'
print(result['ar_peak'])
```

The plot for this example should look like the following plot:

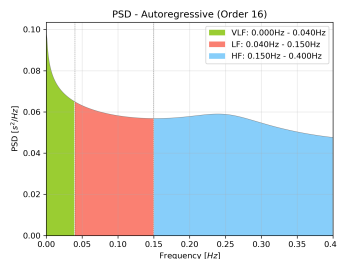


Fig. 18: PSD plot without parameters.

6.4.4 2D PSD Comparison Plot: `psd_comparison()`

```
pyhrv.frequency_domain.psd_comparison(nni=None, rpeaks=None, segments=None,
                                       method='welch', fbands=None, duration=300,
                                       show=True, kwargs=None)
```

Function Description

Computes a series of PSDs from NNI segments extracted from a NNI/R-Peak input series or a series of input NNI segments and plots the result in a single plot. The PSDs are computed using the `welch_psd()`, `lomb_psd()`, or `ar_psd()` functions presented above.

This function aims to facilitate the visualization, comparison, and analysis of PSD evolution over time or NNI segments. An example of a PSD comparison plot generated by this function can be seen here:

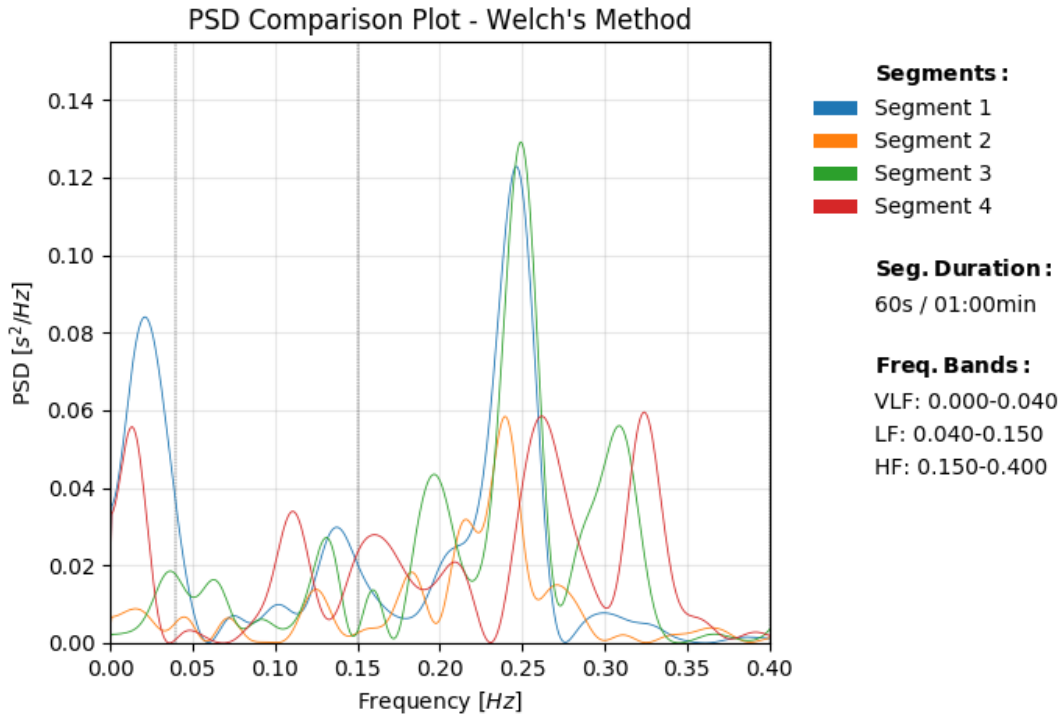


Fig. 19: Sample PSD comparison plot.

See also:

- *Welch's Method:* `welch_psd()`
- *Lomb-Scargle Periodogram:* `lomb_psd()`
- *Autoregressive Method:* `ar_psd()`

If no frequency bands are specified, the default frequency band limits for the *Very Low Frequency (VLF)*, *Low Frequency (LF)*, and *High Frequency (HF)* bands as recommended by the [HRV Guidelines](#) are applied:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

Use the `fbands` parameter to specify custom frequency bands and the possibility to add the *Ultra Low Frequency (ULF)* band (see **Application Notes & Examples & Tutorials** below for more information).

The following parameters are computed from the PSDs and the specified frequency bands for each segment:

- Peak frequencies [Hz]
- Absolute powers [ms²]
- Relative powers [ms²]
- Logarithmic powers [log]

- Normalized powers (LF & HF only)[-]
- Total power of all frequency bands [ms²]

Input Parameters

- `nni` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]
- `segments` (array of arrays): Array containing pre-selected segments for the PSD computation in [ms] or [s]
- `method` (str): PSD estimation method ('welch', 'ar' or 'lomb')
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `duration` (int): Maximum duration duration per segment in [s] (default: 300s)
- `show` (bool, optional): If True, show PSD plot figure (default: True)
- `kwargs_method` (dict): Dictionary of kwargs for the PSD computation functions 'welch_psd()', 'ar_psd()' or 'lomb_psd()'

Note: If `fbands` is none, the default values for the frequency bands will be set.

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below for more information on how to define custom frequency bands.

Returns (ReturnTuple Object)

The results of this function are returned in a nested `biosppy.utils.ReturnTuple` object with the following structure:

- `psd_comparison_plot` (matplotlib figure): Plot figure of the 2D comparison plot
- `segN` (dict): Plot data and PSD parameters of the segment N

The `segN` contains the Frequency Domain parameter results computed from the segment N. The segments have number keys (e.g. first segment = `seg0`, second segment = `seg0`, ..., last segment = `segN`).

Example of a 2-segment output:

```
'seg0': {  
    # Frequency Domain parameters of the first segment (e.g., 'fft_peak', 'fft_abs  
    ↳', 'fft_log', etc.)  
}  
'seg1': {  
    # Frequency Domain parameters of the second segment (e.g., 'fft_peak', 'fft_  
    ↳abs', 'fft_log', etc.)  
}  
'psd_comparison_plot': # matplotlib figure of the comparison plot
```

See also:

The `biosppy.utils.ReturnTuple` Object

Important: If the the selected `duration` exceeds the overall duration of the input NNI series, the standard PSD plot and frequency domain results of the selected PDS method will be returned.

Keep an eye for warnings indicating if this is the case, as the output of this function will then provide the same output as the `welch_psd()`, `lomb_psd()` or `ar_psd()`.

The `kwargs_method` input parameter will not have any effect in such cases.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

Segments will be chosen over 'nni' or 'rpeaks'.

See also:

Section *NN Format*: `nn_format()` for more information.

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the PSD comparison plot:

```
# Import packages
import pyhrv
import pyhrv.frequency_domain as fd

# Load NNI sample series
nni = pyhrv.utils.load_sample_nni()

# Compute the PSDs and the comparison plot using the Welch's method and 60s segments
result = fd.psd_comparison(nni=nni, duration=60, method='welch')

# Access peak frequencies of the first segment using the key 'fft_peak'
print(result['seg1']['fft_peak'])
```

Alternatively, you can use R-peak series (rpeaks), too:

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute the PSDs and the comparison plot using the Welch's method and 60s segments
result = fd.psd_comparison(rpeaks=rpeaks, duration=60, method='welch')
```

The plot of these examples should look like the following plot:

If you want to specify custom frequency bands, define the limits in a Python dictionary as shown in the following example:

```
# Define custom frequency bands and add the ULF band
fbands = {'ulf': (0.0, 0.1), 'vlf': (0.1, 0.2), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}

# Compute the PSDs with custom frequency bands
result = fd.psd_comparison(nni=nni, duration=60, method='welch', fbands=fbands)
```

You can also use the Autoregressive method and the Lomb-Scargle methods:

```
# Compute the PSDs and the comparison plot using the AR method and 60s segments
result = fd.psd_comparison(rpeaks=rpeaks, duration=60, method='ar')

# Compute the PSDs and the comparison plot using the Lomb-Scargle method and 60s
↪ segments
result = fd.psd_comparison(rpeaks=rpeaks, duration=60, method='lomb')
```

This should produce the following results:

Using the `psd_comparison()` function does not restrict you in specifying input parameters for the individual PSD methods. Define the compatible input parameters in Python dictionaries and pass them to the `kwargs` input dictionary of this function.

```
# Define input parameters for the 'welch_psd()' function & plot the PSD comparison
kwargs_welch = {'nfft': 2**8, 'detrend': False, 'window': 'hann'}
result = fd.psd_comparison(nni=nni, duration=60, method='welch', kwargs_method=kwargs_
↪ welch)
```

(continues on next page)

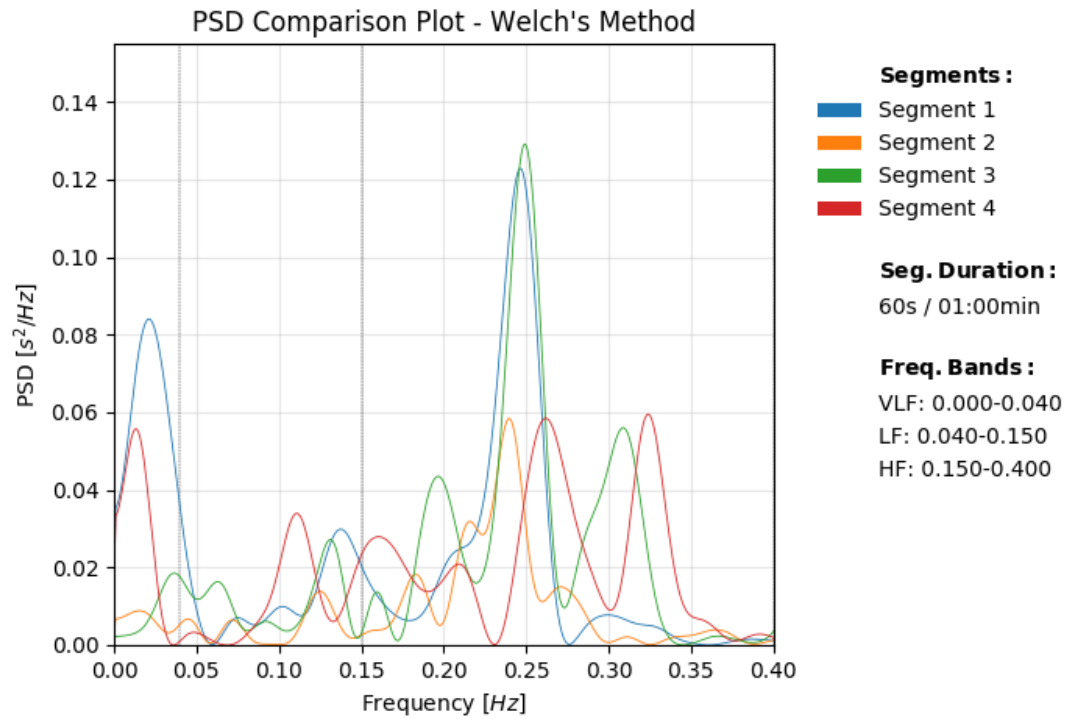


Fig. 20: Comparison of PSDs computing the Welch's method with default frequency bands.

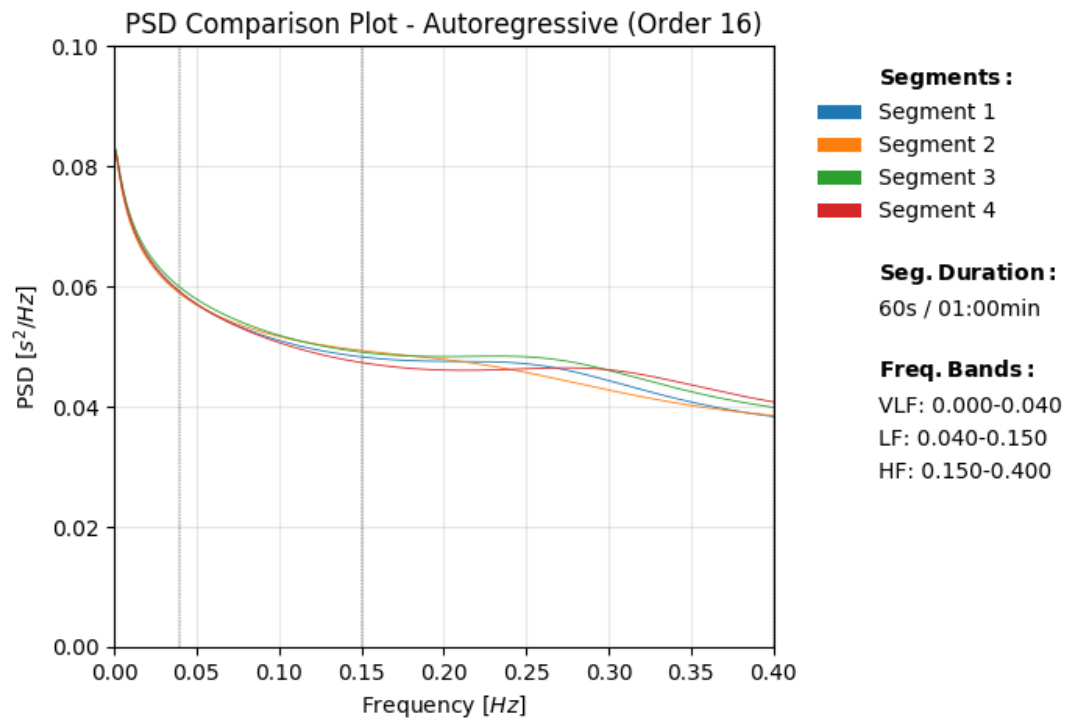


Fig. 21: Comparison of PSDs computing the Autoregressive method with default frequency bands.

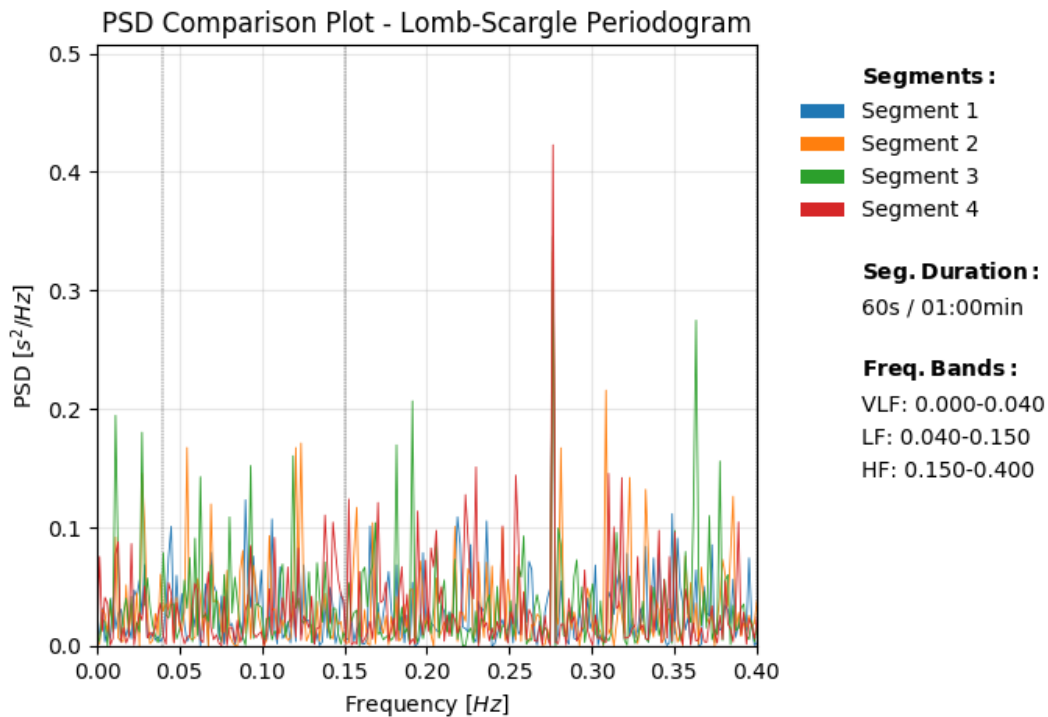


Fig. 22: Comparison of PSDs computing the Lomb-Scargle method with default frequency bands.

(continued from previous page)

```
# Define input parameters for the 'lomb_psd()' function & plot the PSD comparison
kwargs_lomb = {'nfft': 2**8, 'ma_order': 5}
result = fd.psd_comparison(nni=nni, duration=60, method='lomb', kwargs_method=kwargs_
    ↪lomb)

# Define input parameters for the 'ar_psd()' function & plot the PSD comparison
kwargs_ar = {'nfft': 2**8, 'order': 30}
result = fd.psd_comparison(nni=nni, duration=60, method='ar', kwargs_method=kwargs_ar)
```

Note: Some input parameters of the `welch_psd()`, `ar_psd()`, or `lomb_psd()` will be ignored when provided via the `'kwargs_method'` input parameter to ensure the functionality this function

pyHRV is robust against invalid parameter keys. For example, if an invalid input parameter such as `'threshold'` is provided, this parameter will be ignored and a warning message will be issued.

```
# Define custom input parameters using the kwargs dictionaries
kwargs_welch = {
    'nfft': 2**8,          # Valid key, will be used
    'threshold': 2**8      # Invalid key for the Welch's method domain, will be ignored
}

# Generate PSD comparison plot
result = fd.psd_comparison(nni=nni, duration=60, method='welch', kwargs_method=kwargs_
    ↪welch)
```

(continues on next page)

(continued from previous page)

This will trigger the following warning message.

Warning: *Unknown kwargs for 'welch_psd()': threshold. These kwargs have no effect.*

6.4.5 3D PSD Waterfall Plot: `psd_waterfall()`

```
pyhrv.frequency_domain.psd_comparison(nni=None, rpeaks=None, segments=None,
                                       method='welch', fbands=None, kwargs_method={},
                                       duration=300, show=True, legend=True)
```

Function Description

Computes a series of PSDs from NNI segments extracted from a NNI/R-Peak input series or a series of input NNI segments and plots the result in a single plot 3D plot. The PSDs are computed using the `welch_psd()`, `lomb_psd()`, or `ar_psd()` functions presented above.

This function aims to facilitate the visualization, comparison, and analysis of PSD evolution over time or NNI segments.

An example of a 3D waterfall plot generated by this function can be seen here:

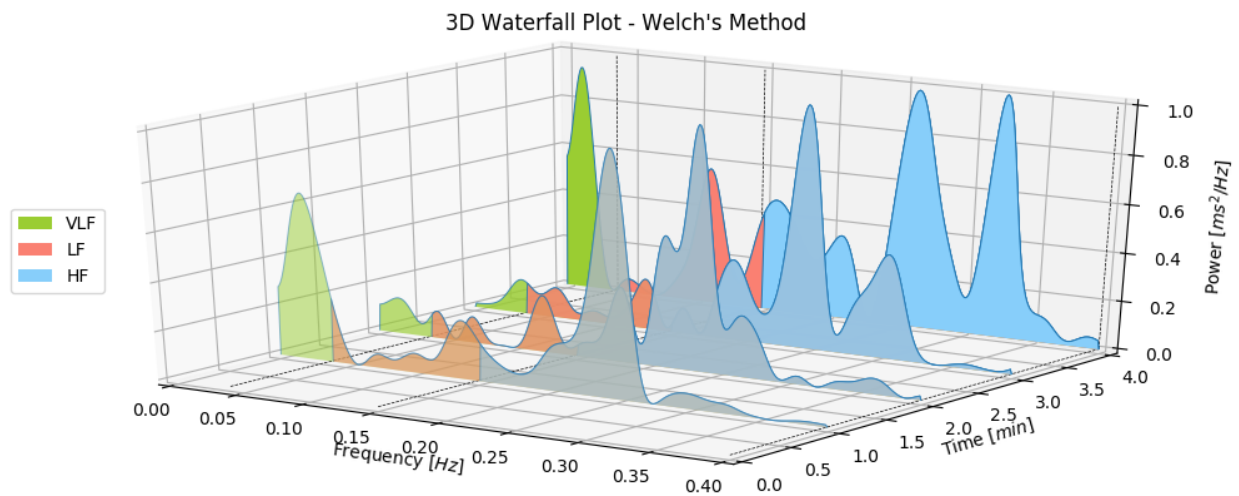


Fig. 23: Sample PSD comparison plot.

See also:

- *Welch's Method:* `welch_psd()`
- *Lomb-Scargle Periodogram:* `lomb_psd()`
- *Autoregressive Method:* `ar_psd()`

If no frequency bands are specified, the default frequency band limits for the *Very Low Frequency (VLF)*, *Low Frequency (LF)*, and *High Frequency (HF)* bands as recommended by the [HRV Guidelines](#) are applied:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

Use the `fbands` parameter to specify custom frequency bands and the possibility to add the *Ultra Low Frequency (ULF)* band (see **Application Notes & Examples & Tutorials** below for more information).

The following parameters are computed from the PSDs and the specified frequency bands for each segment:

- Peak frequencies [Hz]
- Absolute powers [ms²]
- Relative powers [ms²]
- Logarithmic powers [log]
- Normalized powers (LF & HF only)[-]
- Total power of all frequency bands [ms²]

Input Parameters

- `nni` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]
- `segments` (array of arrays): Array containing pre-selected segments for the PSD computation in [ms] or [s]
- `method` (str): PSD estimation method ('welch', 'ar' or 'lomb')
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `kwargs_method` (dict): Dictionary of kwargs for the PSD computation functions 'welch_psd()', 'ar_psd()' or 'lomb_psd()'
- `duration` (int): Maximum duration duration per segment in [s] (default: 300s)
- `show` (bool, optional): If True, show PSD plot figure (default: True)
- `legend` (bool, optional): If True, add a legend with frequency bands to the plat (default: True)

Note: If `fbands` is none, the default values for the frequency bands will be set.

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below for more information on how to define custom frequency bands.

Returns (ReturnTuple Object)

The results of this function are returned in a nested `biosppy.utils.ReturnTuple` object with the following structure:

- `psd_waterfall_plot` (matplotlib figure): Plot figure of the 3D waterfall plot
- `segN` (dict): Plot data and PSD parameters of the segment N

The `segN` contains the Frequency Domain parameter results computed from the segment N. The segments have number keys (e.g. first segment = `seg0`, second segment = `seg0`, ..., last segment = `segN`).

Example of a 2-segment output:

```
'seg0': {  
    # Frequency Domain parameters of the first segment (e.g., 'fft_peak', 'fft_abs  
    ↳', 'fft_log', etc.)  
}  
'seg1': {  
    # Frequency Domain parameters of the second segment (e.g., 'fft_peak', 'fft_  
    ↳abs', 'fft_log', etc.)  
}  
'psd_waterfall_plot': # matplotlib figure of the 3D waterfall plot
```

See also:

The `biosppy.utils.ReturnTuple` Object

Important: If the the selected duration exceeds the overall duration of the input NNI series, the standard PSD plot and frequency domain results of the selected PDS method will be returned.

Keep an eye for warnings indicating if this is the case, as the output of this function will then provide the same output as the `welch_psd()`, `lomb_psd()` or `ar_psd()`.

The `kwargs_method` input parameter will not have any effect in such cases.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()`.

See also:

- *What may help when matplotlib blocks your code from being executed*
 - *More information about the matplotlib Interactive Mode*
 - *More information about matplotlib Backends*
-

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the PSD comparison plot:

```
# Import packages
import pyhrv
import pyhrv.frequency_domain as fd

# Load NNI sample series
nni = pyhrv.utils.load_sample_nni()

# Compute the PSDs and the comparison plot using the Welch's method and 60s segments
result = fd.psd_waterfall(nni=nni, duration=60, method='welch')

# Access peak frequencies of the first segment using the key 'fft_peak'
print(result['psd_data']['seg1']['fft_peak'])
```

Alternatively, you can use R-peak series (`rpeaks`), too:

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute the PSDs and the comparison plot using the Welch's method and 60s segments
result = fd.psd_waterfall(rpeaks=rpeaks, duration=60, method='welch')
```

The plot of these examples should look like the following plot:

If you want to specify custom frequency bands, define the limits in a Python dictionary as shown in the following example:

```
# Define custom frequency bands and add the ULF band
fbands = {'ulf': (0.0, 0.1), 'vlf': (0.1, 0.2), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}

# Compute the PSDs with custom frequency bands
result = fd.psd_waterfall(nni=nni, duration=60, method='welch', fbands=fbands)
```

You can also use the Autoregressive method and the Lomb-Scargle methods:

```
# Compute the PSDs and the waterfall plot using the AR method and 60s segments
result = fd.psd_waterfall(rpeaks=rpeaks, duration=60, method='ar')
```

(continues on next page)

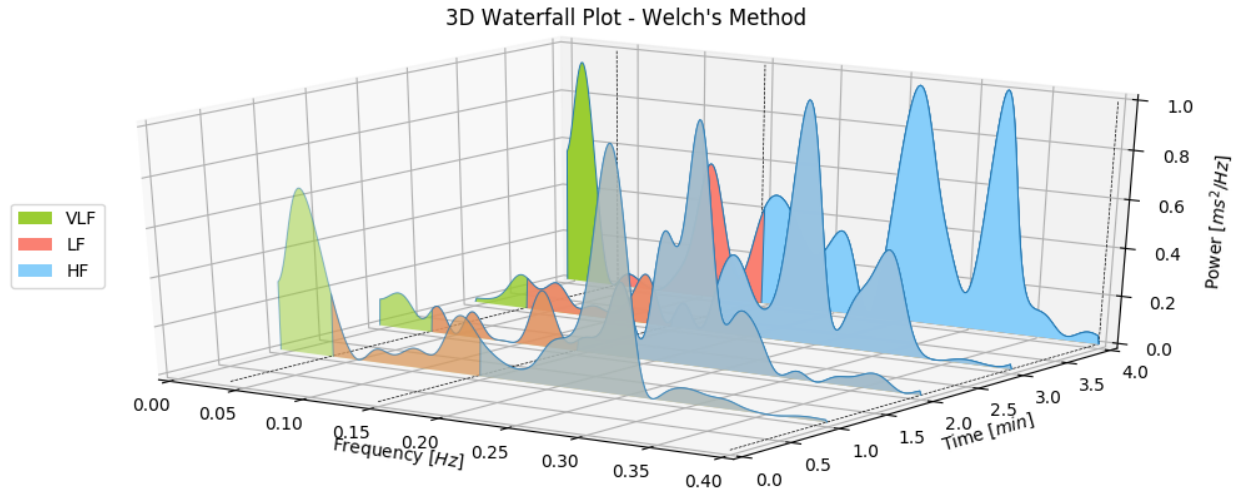


Fig. 24: PSD waterfall computed using the Welch's method with default frequency bands.

(continued from previous page)

```
# Compute the PSDs and the waterfall plot using the Lomb-Scargle method and 60s
↳ segments
result = fd.psd_waterfall(rpeaks=rpeaks, duration=60, method='lomb')
```

This should produce the following results:

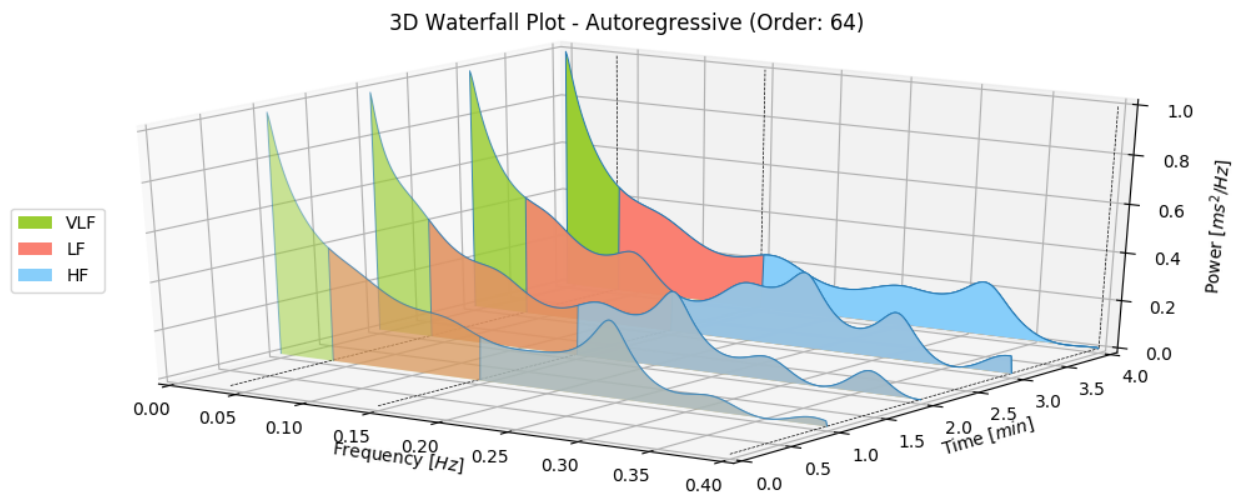


Fig. 25: PSD waterfall computed using the Autoregressive method with default frequency bands.

Using the `psd_waterfall()` function does not restrict you in specifying input parameters for the individual PSD methods. Define the compatible input parameters in Python dictionaries and pass them to the `kwargs` input dictionary of this function.

```
# Define input parameters for the 'welch_psd()' function & plot the PSD comparison
kwargs_welch = {'nfft': 2**8, 'detrend': False, 'window': 'hann'}
result = fd.psd_waterfall(nni=nni, duration=60, method='welch', kwargs_method=kwargs_
↳ welch)
```

(continues on next page)

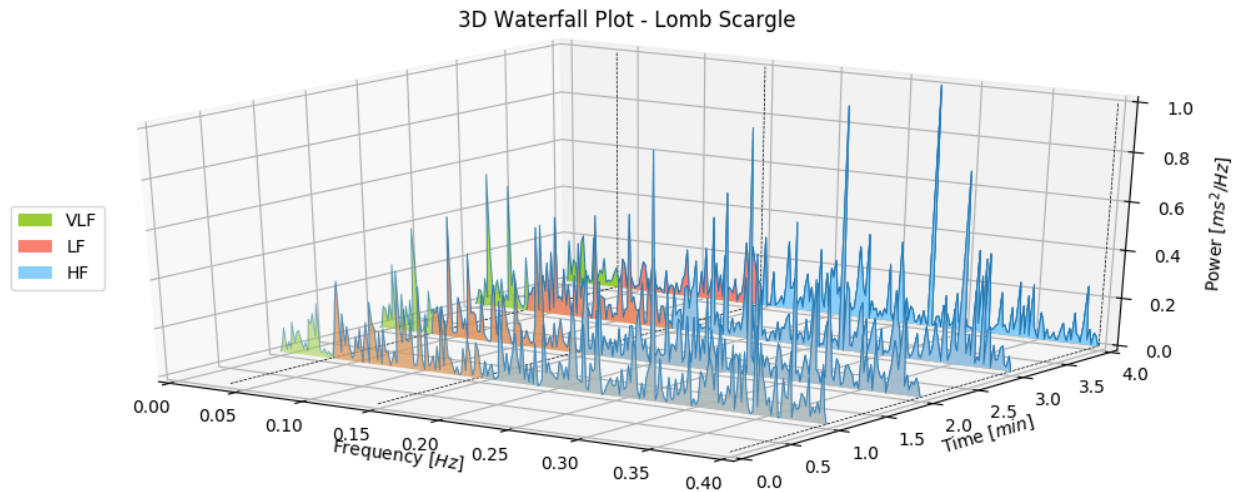


Fig. 26: PSD waterfall computed using the Lomb-Scargle method with default frequency bands.

(continued from previous page)

```
# Define input parameters for the 'lomb_psd()' function & plot the PSD comparison
kwargs_lomb = {'nfft': 2**8, 'ma_order': 5}
result = fd.psd_waterfall(nni=nni, duration=60, method='lomb', kwargs_method=kwargs_
→lomb)

# Define input parameters for the 'ar_psd()' function & plot the PSD comparison
kwargs_ar = {'nfft': 2**8, 'order': 30}
result = fd.psd_waterfall(nni=nni, duration=60, method='ar', kwargs_method=kwargs_ar)
```

Note: Some input parameters of the `welch_psd()`, `ar_psd()`, or `lomb_psd()` will be ignored when provided via the `'kwargs_method'` input parameter to ensure the functionality this function

pyHRV is robust against invalid parameter keys. For example, if an invalid input parameter such as `'threshold'` is provided, this parameter will be ignored and a warning message will be issued.

```
# Define custom input parameters using the kwargs dictionaries
kwargs_welch = {
    'nfft': 2**8,          # Valid key, will be used
    'threshold': 2**8      # Invalid key for the Welch's method domain, will be ignored
}

# Generate PSD comparison plot
result = fd.psd_waterfall(nni=nni, duration=60, method='welch', kwargs=kwargs_welch)
```

This will trigger the following warning message.

Warning: Unknown kwargs for `'welch_psd()'`: `threshold`. These kwargs have no effect.

6.4.6 Domain Level Function: `frequency_domain()`

```
pyhrv.frequency_domain.frequency_domain(signal=None,      nn=None,      rpeaks=None,
                                         sampling_rate=1000.,      fbands=None,
                                         show=False, show_param=True, legend=True,
                                         kwargs_welch=None,      kwargs_lomb=None,
                                         kwargs_ar=None)
```

Function Description

Computes PSDs using the Welch, Lomb, and Autoregressive methods by calling the `welch_psd()`, `lomb_psd()`, and `ar_psd()` functions, computes frequency domain parameters, and returns the results in a single `biosppy.utils.ReturnTuple` object.

See also:

- *Welch's Method:* `welch_psd()`
- *Lomb-Scargle Periodogram:* `lomb_psd()`
- *Autoregressive Method:* `ar_psd()`

If no frequency bands are specified, the default frequency band limits for the *Very Low Frequency (VLF)*, *Low Frequency (LF)*, and *High Frequency (HF)* bands as recommended by the [HRV Guidelines](#) are applied:

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

Use the `fbands` parameter to specify custom frequency bands and the possibility to add the *Ultra Low Frequency (ULF)* band (see **Application Notes & Examples & Tutorials** below for more information).

The following parameters are computed from the PSD and the specified frequency bands:

- Peak frequencies [Hz]
- Absolute powers [ms^2]
- Relative powers [ms^2]
- Logarithmic powers [log]
- Normalized powers (LF & HF only)[-]
- Total power of all frequency bands [ms^2]

Input Parameters

- `signal` (array): ECG signal
- `nni` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `show` (bool, optional): If true, show all PSD plots.
- `show_param` (bool, optional):
- `window` (scipy.window function, optional): Window function used for PSD estimation (default: 'hamming')
- `show` (bool, optional): If True, show PSD plot figure (default: True)
- `show_param` (bool, optional): If true, list all computed parameters next to the plot (default: True)

- `kwargs_welch` (dict, optional): Dictionary containing the kwargs for the ‘`welch_psd`’ function
- `kwargs_lomb` (dict, optional): Dictionary containing the kwargs for the ‘`lomb_psd`’ function
- `kwargs_ar` (dict, optional): Dictionary containing the kwargs for the ‘`ar_psd`’ function

Important: This function calls the PSD using either the `signal`, `nni`, or `rpeaks` data. Provide only one type of data, as it is not required to pass all three types at once.

Note: If `fbands` is none, the default values for the frequency bands will be set.

- VLF: [0.00Hz - 0.04Hz]
- LF: [0.04Hz - 0.15Hz]
- HF: [0.15Hz - 0.40Hz]

See **Application Notes & Examples & Tutorials** below for more information on how to define custom frequency bands.

Returns (ReturnTuple Object) The results of this function are returned in a `biosppy.utils.ReturnTuple` object. This function returns the frequency parameters computed with all three PSD estimation methods. You can access all the parameters using the following keys (X = one of the methods ‘`fft`’, ‘`ar`’, ‘`lomb`’):

- `X_peak` (tuple): Peak frequencies of all frequency bands [Hz]
- `X_abs` (tuple): Absolute powers of all frequency bands [ms^2]
- `X_rel` (tuple): Relative powers of all frequency bands [%]
- `X_log` (tuple): Logarithmic powers of all frequency bands [log]
- `X_norm` (tuple): Normalized powers of the LF and HF frequency bands [-]
- `X_ratio` (float): LF/HF ratio [-]
- `X_total` (float): Total power over all frequency bands [ms^2]
- `X_plot` (matplotlib figure object): PSD plot figure object
- `fft_interpolation` (str): Interpolation method used for NNI interpolation (hard-coded to ‘`cubic`’)
- `fft_resampling_frequency` (int): Resampling frequency used for NNI interpolation [Hz] (hard-coded to 4Hz as recommended by the [HRV Guidelines](#))
- `fft_window` (str): Spectral window used for PSD estimation of the Welch’s method
- `lomb_ma` (int): Moving average window size
- `ar_interpolation` (str): Interpolation method used for NNI interpolation (hard-coded to ‘`cubic`’)
- `ar_resampling_frequency` (int): Resampling frequency used for NNI interpolation [Hz] (hard-coded to 4Hz as recommended by the [HRV Guidelines](#))
- `ar_order` (int): Autoregressive model order

See also:

The `biosppy.utils.ReturnTuple` Object

Application Notes

It is not necessary to provide input data for `signal`, `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`signal`, `nni` **or** `rpeaks`). The input data will be prioritized in the following order, in case multiple inputs are provided:

1. `signal`, 2. `nni`, 3. `rpeaks`.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Incorrect frequency band specifications will be automatically corrected, if possible. For instance the following frequency bands contain overlapping frequency band limits which would cause issues when computing the frequency parameters:

```
fbands = {'vlf': (0.0, 0.25), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}
```

Here, the upper band of the VLF band is greater than the lower band of the LF band. In this case, the overlapping frequency band limits will be switched:

```
fbands = {'vlf': (0.0, 0.2), 'lf': (0.25, 0.3), 'hf': (0.3, 0.4)}
```

Warning: Corrections of frequency bands trigger warnings which are displayed in the Python console. It is recommended to watch out for these warnings and to correct the frequency bands given that the corrected bands might not be optimal.

This issue is shown in the following PSD plot where the corrected frequency bands above were used and there is no frequency band covering the range between 0.2Hz and 0.25Hz:

Use the `kwargs_welch` dictionary to pass function specific parameters for the `welch_psd()` method. The following keys are supported:

- `nfft` (int, optional): Number of points computed for the FFT result (default: 2^{**12})
- `detrend` (bool, optional): If True, detrend NNI series by subtracting the mean NNI (default: True)
- `window` (scipy.window function, optional): Window function used for PSD estimation (default: 'hamming')

Use the `lomb_psd` dictionary to pass function specific parameters for the `lombg_psd()` method. The following keys are supported:

- `nfft` (int, optional): Number of points computed for the Lomb-Scargle result (default: 2^{**8})
- `ma_order` (int, optional): Order of the moving average filter (default: None; no filter applied)

Use the `ar_psd` dictionary to pass function specific parameters for the `ar_psd()` method. The following keys are supported:

- `nfft` (int, optional): Number of points computed for the FFT result (default: 2^{**12})
- `order` (int, optional): Autoregressive model order (default: 16)

Important: The following input data is equally set for all the 3 methods using the input parameters of this function without using the `kwargs` dictionaries.

Defining these parameters/this specific input data individually in the `kwargs` dictionaries will have no effect:

- `nn` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]

- `show` (bool, optional): If True, show PSD plot figure (default: True)
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `show_param` (bool, optional): If true, list all computed PSD parameters next to the plot (default: True)
- `legend` (bool, optional): If true, add a legend with frequency bands to the plot (default: True)

Any key or parameter in the kwargs dictionaries that is not listed above will have no effect on the functions.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()` function.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following example codes demonstrate how to use the `frequency_domain()` function.

You can choose either the ECG signal, the NNI series or the R-peaks as input data for the PSD estimation and parameter computation:

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNI series
nni = tools.nn_intervals(t[rpeaks])

# OPTION 1: Compute PSDs using the ECG Signal
signal_results = fd.frequency_domain(signal=filtered_signal)

# OPTION 2: Compute PSDs using the R-peak series
rpeaks_results = fd.frequency_domain(rpeaks=t[rpeaks])

# OPTION 3: Compute PSDs using the
nni_results = fd.frequency_domain(nni=nni)
```

The output of all three options above will be the same.

Note: If an ECG signal is provided, the signal will be filtered and the R-peaks will be extracted using the `biosppy.signals.ecg.ecg()` function. Finally, the NNI series for the PSD estimation will be computed from the extracted

R-peak series.

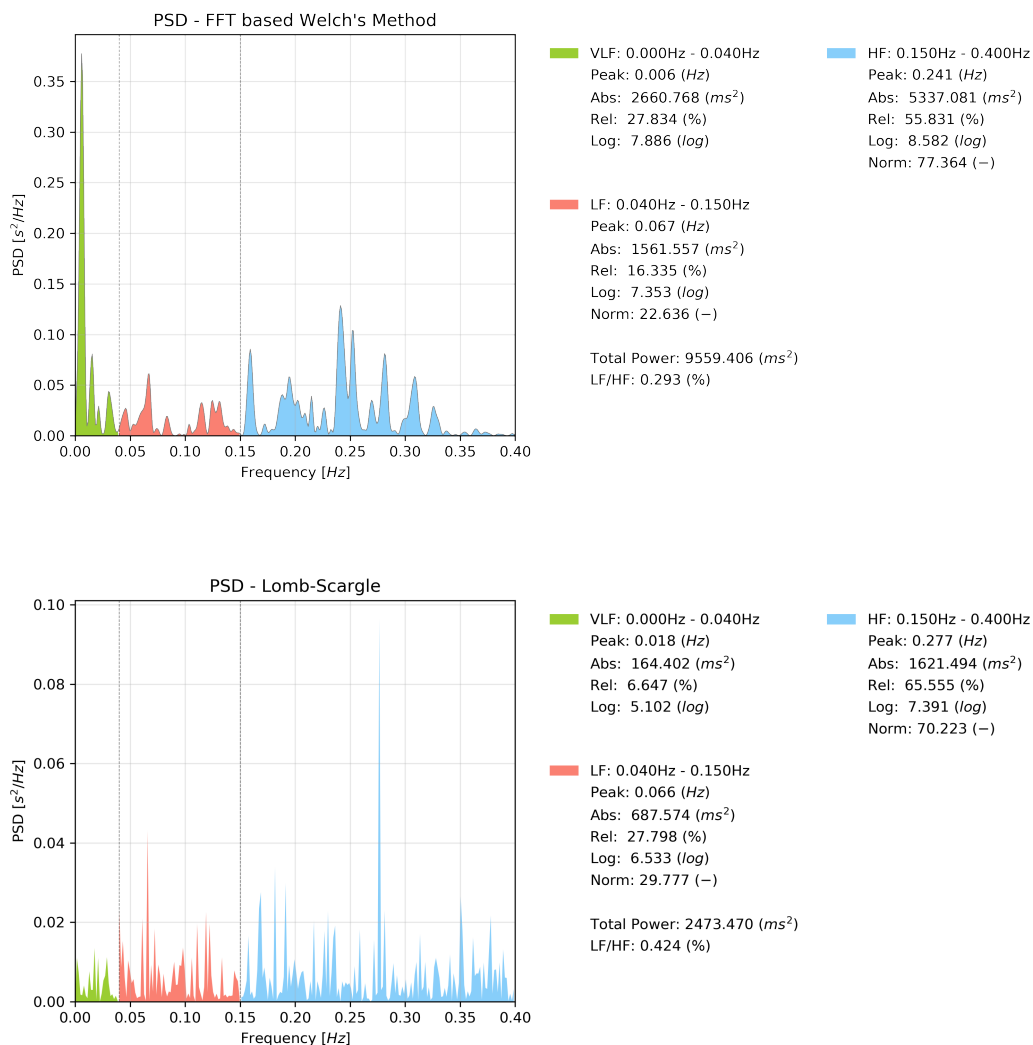
See also:

`biosppy.signals.ecg.ecg()`

You can now access the frequency parameters of each method using the following commands:

```
# Access peak frequencies from each method (works the same for 'rpeaks_results' and
# 'nni_results')
print(signal_results['fft_peak'])
print(signal_results['lomb_peak'])
print(signal_results['ar_peak'])
```

The plots generated using the example above should look like the following plots:



If you want to specify custom frequency bands, define the limits in a Python dictionary as shown in the following example below:

Note: The frequency bands are equally defined for all three PSD estimation methods when using the

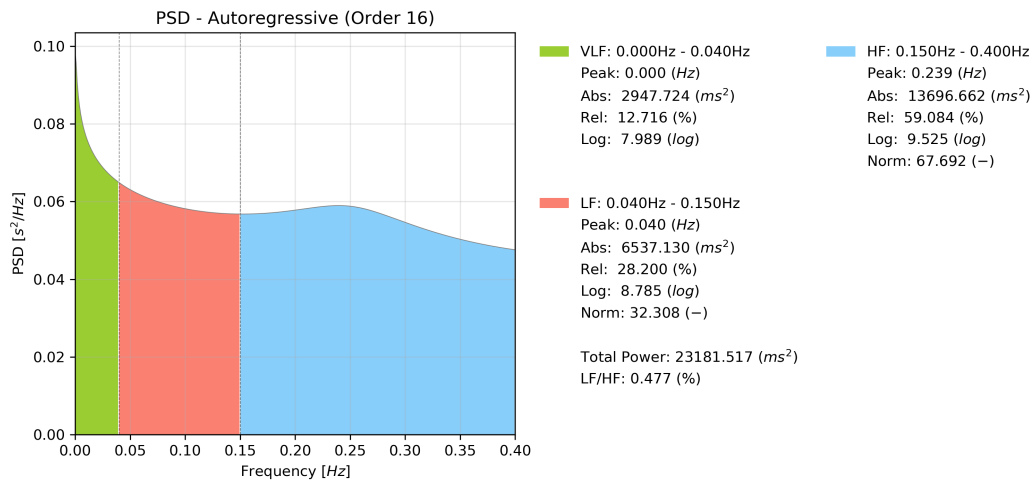


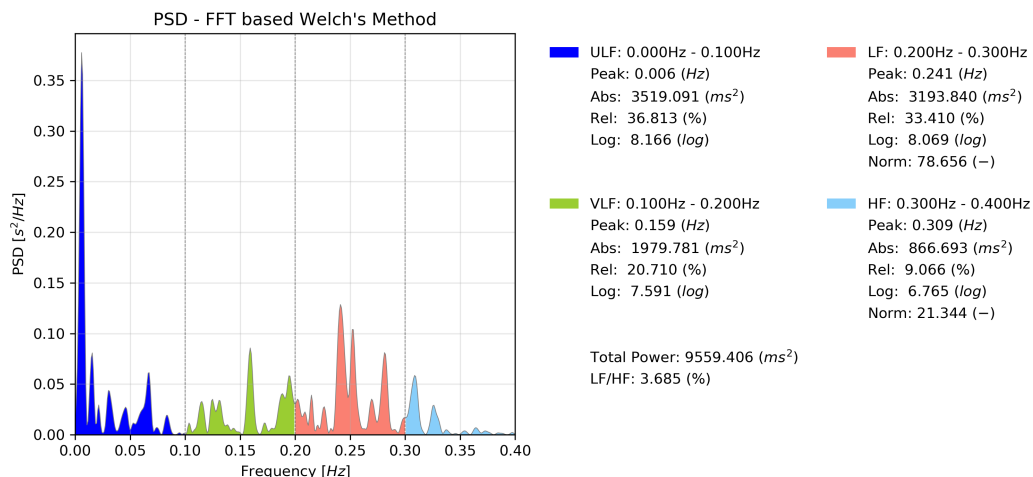
Fig. 28: Welch, Lomb-Scargle and Autoregressive PSDs with default parameters using only the `frequency_domain()` function.

`frequency_domain()` function. Use the individual method functions instead, in case you want to define method-specific frequency bands.

```
# Define custom frequency bands and add the ULF band
fbands = {'ulf': (0.0, 0.1), 'vlf': (0.1, 0.2), 'lf': (0.2, 0.3), 'hf': (0.3, 0.4)}

# Compute the PSD with custom frequency bands
result = fd.frequency_domain(nni, fbands=fbands)
```

The plots generated using the example above should look like the following plots:



By default, the figure will contain the PSD plot on the left and the computed parameter results on the right side of the figure. Set the `show_param` to `False` if only the PSD is needed in the figure.

Using the `frequency_domain()` function does not restrict you in specifying input parameters for the individual PSD methods. Define the compatible input parameters in Python dictionaries and pass them to the `kwargs` input

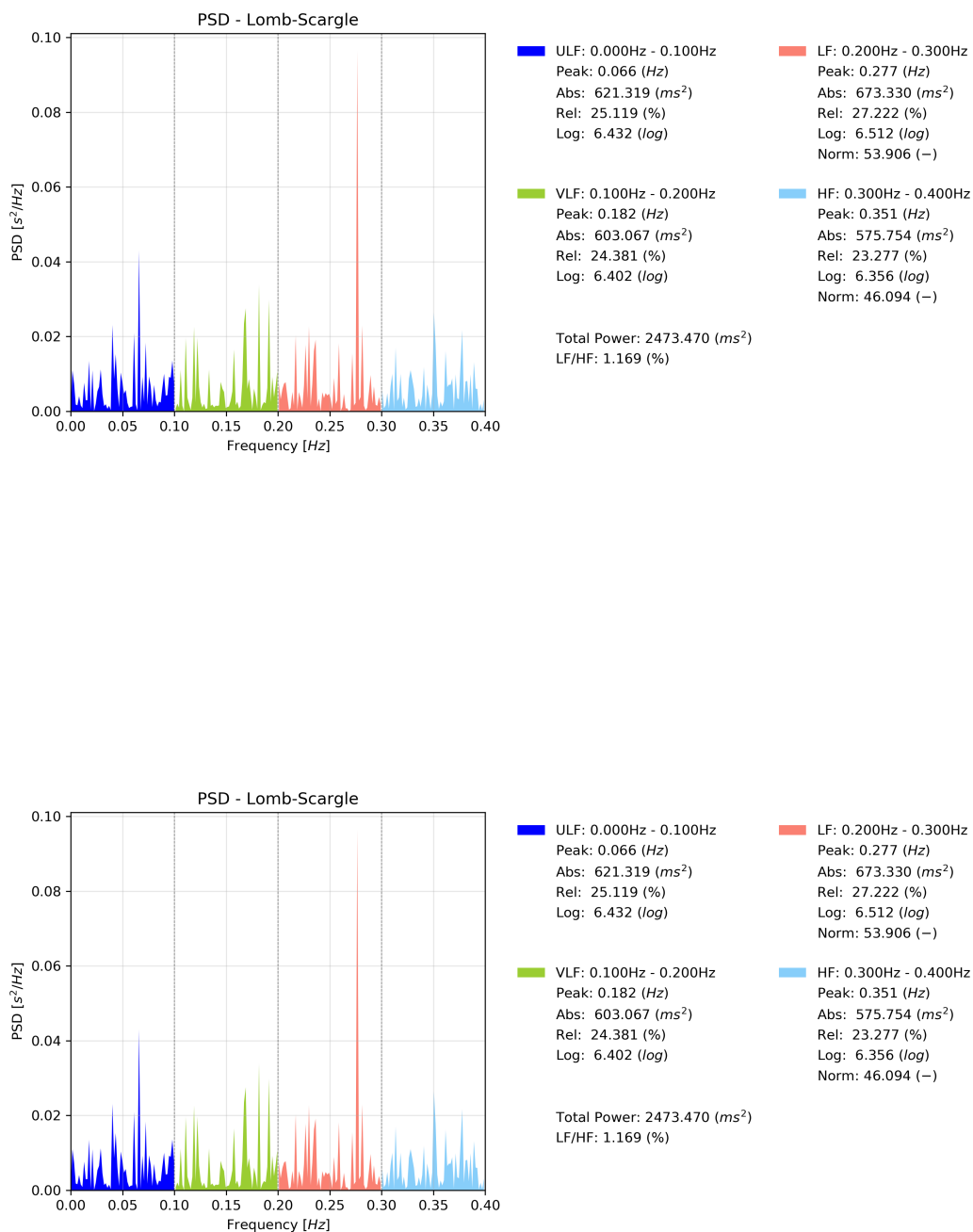


Fig. 29: Welch, Lomb-Scargle and Autoregressive PSDs with custom frequency bands using only the “frequency_domain ()” function.

dictionaries of this function (see this functions **Application Notes** for a list of compatible parameters):

```
# Import packages
import biosppy
import pyhrv.frequency_domain as fd

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Define input parameters for the 'welch_psd()' function
kwargs_welch = {'nfft': 2**8, 'detrend': False, 'window': 'hann'}

# Define input parameters for the 'lomb_psd()' function
kwargs_lomb = {'nfft': 2**8, 'ma_order': 5}

# Define input parameters for the 'ar_psd()' function
kwargs_ar = {'nfft': 2**8, 'order': 30}

# Compute PSDs using the ECG Signal
signal_results = fd.frequency_domain(signal=filtered_signal, show=True,
kwargs_welch=kwargs_lomb, kwargs_lomb=kwargs_lomb, kwargs_ar=kwargs_ar)
```

pyHRV is robust against invalid parameter keys. For example, if an invalid input parameter such as 'threshold' is provided in any of the frequency domain kwargs dictionaries, this parameter will be ignored and a warning message will

be issued.

```
# Define custom input parameters using the kwargs dictionaries
kwargs_welch = {
    'nfft': 2**8,          # Valid key, will be used
    'threshold': 2**8      # Invalid key for the Welch's method domain, will be ignored
}

# Compute HRV parameters
fd.frequency_domain(nni=nni, kwargs_welch=kwargs_welch)
```

This will trigger the following warning message.

Warning: *Unknown kwargs for 'welch_psd()': threshold. These kwargs have no effect.*

6.4.7 Frequency Parameters

The following parameters and their computation formulas are computed from each of the PSD estimation methods computed using the `welch_psd()`, `lomb_psd()`, and `ar_psd()` functions presented above.

Note: The returned BioSPPy ReturnTuple object contains all frequency band parameters in parameter specific tuples of length 4 when using the ULF frequency band or of length 3 when NOT using the ULF frequency band. The structures of those tuples are shown in this example below:

Using ULF, VLF, LF and HF frequency bands:


```
fft_results['fft_peak'] = (ulf_peak, vlf_peak, lf_peak, hf_peak)
```

Using VLF, LF and HF frequency bands:

```
fft_results['fft_peak'] = (vlf_peak, lf_peak, hf_peak)
```

Absolute Powers

The absolute powers [ms²] are individually computed for each frequency band as the sum of the power over the frequency band.

$$P_{abs} = \Delta f \sum_{f=f_{min}}^{f_{max}} S(f)$$

with:

- P_{abs} : Absolute power
- Δf : Frequency resolution
- f_{min} : Lower limit of the frequency band
- f_{max} : Upper limit of the frequency band
- $S(f)$: PSD function in dependence of the frequency f

The absolute powers are stored in the ReturnTuple object and can be accessed with one of the following keys depending on the PSD method being used:

- `fft_abs` as a result of the `welch_psd()` function
- `lomb_abs` as a result of the `lomb_psd()` function
- `ar_abs` as a result of the `ar_psd()` function

Note: In case you are using the `pyhrv.hrv()` or the `pyhrv.frequency_domain.frequency_domain()` functions, you can use all the three keys listed above as all methods are computed using these functions.

Total Power

The total power [ms²] of the PSD is computed as the sum of the absolute powers of all frequency bands:

$$P_{Total} = P_{ULF} + P_{VLF} + P_{LF} + P_{HF}$$

with:

- P_{Total} : Total power
- P_{ULF} : Absolute power of the ULF frequency band (= 0 if ULF is not specified)
- P_{VLF} : Absolute power of the VLF frequency band
- P_{LF} : Absolute power of the LF frequency band
- P_{HF} : Absolute power of the HF frequency band

The total power is stored in the ReturnTuple object and can be accessed with one of the following keys depending on the PSD method being used:

- `fft_total` as a result of the `welch_psd()` function

- `lomb_total` as a result of the `lomb_psd()` function
- `ar_total` as a result of the `ar_psd()` function

Note: In case you are using the `pyhrv.hrv()` or the `pyhrv.frequency_domain.frequency_domain()` functions, you can use all the three keys listed above as all methods are computed using these functions.

Relative Power

The relative powers $[a]$ are computed as the ratio between the absolute power of a frequency band and the total power:

$$P_{rel,z} = \frac{P_{abs,z}}{P_{Total}} * 100$$

with:

- $P_{rel,z}$: Relative power of the frequency band z
- $P_{abs,z}$: Absolute power of the frequency band z
- z : Frequency band (ULF, VLF, LF or HF)
- P_{Total} : Total power over all frequency bands

The relative powers are stored in the `ReturnTuple` object and can be accessed with one of the following key depending on the PSD method being used:

- `fft_rel` as a result of the `welch_psd()` function
- `lomb_rel` as a result of the `lomb_psd()` function
- `ar_rel` as a result of the `ar_psd()` function

Note: In case you are using the `pyhrv.hrv()` or the `pyhrv.frequency_domain.frequency_domain()` functions, you can use all the three keys listed above as all methods are computed using these functions.

Logarithmic Powers

The logarithmic powers $[log(ms^2)]$ are computed as follows total power:

$$P_{log,z} = log(P_{abs,z})$$

with:

- $P_{log,z}$: Logarithmic power of the frequency band z
- $P_{abs,z}$: Absolute power of the frequency band z
- z : Frequency band (ULF, VLF, LF or HF)

The logarithmic powers are stored in the `ReturnTuple` object and can be accessed with one of the following key depending on the PSD method being used:

- `fft_log` as result of the `welch_psd()` function
- `lomb_log` as result of the `lomb_psd()` function
- `ar_log` as result of the `ar_psd()` function

Note: In case you are using the `pyhrv.hrv()` or the `pyhrv.frequency_domain.frequency_domain()` functions, you can use all the three keys listed above as all methods are computed using these functions.

Normalized Powers

The normalized powers [-] are computed for and based on the LF and HF frequency parameters only according to the following formulas:

$$P_{norm,LF} = \frac{P_{abs,LF}}{P_{abs,LF} + P_{abs,HF}} * 100$$

$$P_{norm,HF} = \frac{P_{abs,HF}}{P_{abs,LF} + P_{abs,HF}} * 100$$

with:

- $P_{norm,LF}$: Normalized power of the LF band
- $P_{abs,LF}$: Absolute power of the LF band
- $P_{norm,HF}$: Normalized power of the HF band
- $P_{abs,HF}$: Absolute power of the HF band

The normalized powers are stored in the ReturnTuple object and can be accessed with one of the following key depending on the PSD method being used:

- `fft_norm` as result of the `welch_psd()` function
- `lomb_norm` as result of the `lomb_psd()` function
- `ar_norm` as result of the `ar_psd()` function

Note: Independently of the specified frequency band (with or without the VLF band) the results of this parameter are always returned in a 2-element tuple. The first element is the normalized power of the LF band with the second being the normalized power of the HF band.

```
fft_results['fft_norm'] = (lf_norm, hf_norm)
```

Note: In case you are using the `pyhrv.hrv()` or the `pyhrv.frequency_domain.frequency_domain()` functions, you can use all the three keys listed above as all methods are computed using these functions.

LF/HF Ratio

The LF/HF ratio is computed based on the absolute powers of the LF and HF bands:

$$\frac{LF}{HF} = \frac{P_{abs,LF}}{P_{abs,HF}}$$

with:

- $P_{abs,LF}$: Absolute power of the LF band
- $P_{abs,HF}$: Absolute power of the HF band

The LF/HF ratio is stored in the ReturnTuple object and can be accessed with one of the following keys depending on the PSD method being used:

- `fft_ratio` (float) as result of the `welch_psd()` function
- `lomb_ratio` (float) as result of the `lomb_psd()` function
- `ar_ratio` (float) as result of the `ar_psd()` function

Note: Other than most of the other HRV frequency domain parameters, this parameter is always returned as a single float value rather than in a multi-dimensional tuple or array.

```
fft_results['fft_ratio'] = float(lf_hf_ratio)
```

Note: In case you are using the `pyhrv.hrv()` or the `pyhrv.frequency_domain.frequency_domain()` functions, you can use all the three keys listed above as all methods are computed using these functions.

6.5 Nonlinear Module

The `nonlinear.py` module contains functions to compute nonlinear HRV parameters and methods.

See also:

[pyHRV Nonlinear Module source code](#)

Module Contents

- *Nonlinear Module*
 - *Poincaré: `poincare()`*
 - *Sample Entropy: `sample_entropy()`*
 - *Detrended Fluctuation Analysis: `dfa()`*
 - *Domain Level Function: `nonlinear()`*

See also:

Useful links:

- [Sample NNI Series \(docs\)](#)
- [Sample NNI Series on GitHub](#)
- [series_1.npy](#) (file used in the examples below)

6.5.1 Poincaré: `poincare()`

```
pyhrv.nonlinear.poincare(nni=None, rpeaks=None, show=True, figsize=None, ellipse=True, vectors=True, legend=True, marker='o')
```

Function Description

Creates the Poincaré plot from a series of NN intervals or R-peak locations and derives the Poincaré related parameters SD1, SD2, SD2/SD1 ratio, and area of the Poincaré ellipse.

An example of the Poincaré scatter plot generated by this function can be seen [here](#):

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

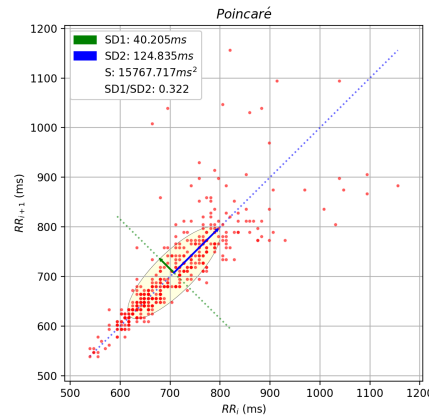


Fig. 30: Poincaré scatter plot.

- `show` (bool, optional): If True, show Poincaré plot (default: True)
- `figsize` (array, optional): Matplotlib figure size; Format: `figsize=(width, height)` (default: None: (6, 6))
- `ellipse` (bool, optional): If True, shows fitted ellipse in plot (default: True)
- `vectors` (bool, optional): If True, shows SD1 and SD2 vectors in plot (default: True)
- `legend` (bool, optional): If True, adds legend to the Poincaré plot (default: True)
- `marker` (str, optional): NNI marker in plot (must be compatible with the matplotlib markers (default: 'o'))

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `poincare_plot` (matplotlib figure object): Poincaré plot figure
- `sd1` (float): Standard deviation (SD1) of the major axis
- `sd2` (float): Standard deviation (SD2) of the minor axis
- `sd_ratio` (float): Ratio between SD1 and SD2 (SD2/SD1)
- `ellipse_area` (float): Area S of the fitted ellipse

See also:

The `biosppy.utils.ReturnTuple` Object

Computation

The SD1 parameter is the standard deviation of the data series along the minor axis and is computed using the SDSD parameter of the Time Domain.

$$SD1 = \sqrt{\frac{1}{2}SDSD^2}$$

with:

- *SD1*: Standard deviation along the minor axis
- *SDSD*: Standard deviation of successive differences (Time Domain parameter)

The SD2 parameter is the standard deviation of the data series along the major axis and is computed using the SDSD and the SDNN parameters of the Time Domain.

$$SD2 = \sqrt{2SDNN^2 - \frac{1}{2}SDSD^2}$$

with:

- *SD2*: Standard deviation along the major axis
- *SDNN*: Standard deviation of the NNI series
- *SDSD*: Standard deviation of successive differences (Time Domain parameter)

The SD ratio is computed as

$$SD_{ratio} = \frac{SD2}{SD1}$$

The area of the ellipse fitted into the Poincaré scatter plot is computed as

$$S = \pi \cdot SD1 \cdot SD2$$

See also:

- *SDNN*: `sdnn()`
- *SDSD*: `sdsd()`

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Use the `ellipse` and the `vectors` input parameters to hide the fitted ellipse and the SD1 and SD2 vectors from the Poincaré scatter plot.

Examples & Tutorials

The following example code demonstrates how to use this function and how to access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the parameters:

```
# Import packages
import pyhrv
import pyhrv.nonlinear as nl

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute Poincaré using NNI series
results = nl.poincare(nni)

# Print SD1
print(results['sd1'])
```

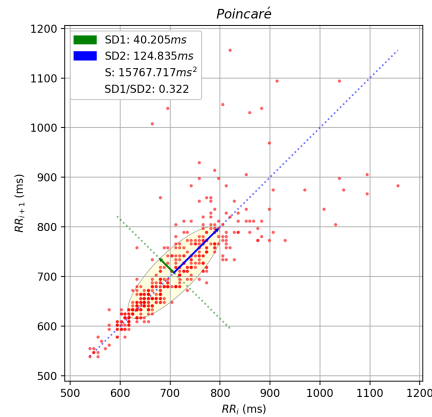


Fig. 31: Poincaré scatter plot with default configurations.

The codes above create a plot similar to the plot below:

Alternatively, you can use R-peak series (rpeaks):

```
# Import packages
import biosppy
import pyhrv.nonlinear as nl

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, _, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute Poincaré using R-peak series
results = pyhrv.nonlinear.poincare(rpeaks=t[rpeaks])
```

Use the ellipse, the vectors and the legend to show only the Poincaré scatter plot.

```
# Show the scatter plot without the fitted ellipse, the SD1 & SD2 vectors and the legend
results = nl.poincare(nni, ellipse=False, vectors=False, legend=False)
```

The generated plot is similar to the one below:

6.5.2 Sample Entropy: `sample_entropy()`

`pyhrv.nonlinear.sampen` (*nni=None, rpeaks=None, dim=2, tolerance=None*)

Function Description

Computes the sample entropy (sampen) of the NNI series for a given Entropy Embedding Dimension and vector distance tolerance.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].

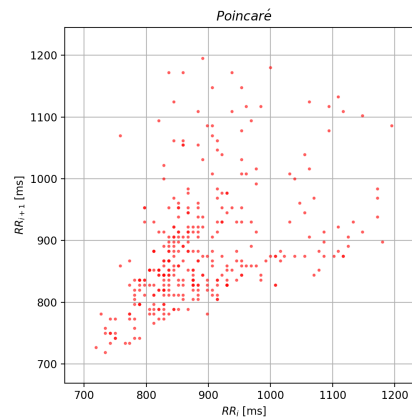


Fig. 32: Barebone Poincaré scatter plot.

- `dim` (int, optional): Entropy embedding dimension (default: 2)
- `tolerance` (int, float, optional): Tolerance distance for which the two vectors can be considered equal (default: `std(NNI)`)

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `sample_entropy` (float): Sample entropy of the NNI series.

See also:

[The `biosppy.utils.ReturnTuple` Object](#)

Raises `TypeError`: If `tolerance` is not a numeric value

Computation

This parameter is computed using the `nolds.sampen()` function.

See also:

[Nolds Documentation - Sampen](#)

The default embedding dimension and tolerance values have been set to suitable HRV values.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the parameters:


```
# Import packages
import pyhrv
import pyhrv.nonlinear as nl

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute Sample Entropy using NNI series
results = nl.sampen(nni)

# Print Sample Entropy
print(results['sampen'])
```

Alternatively, you can use R-peak series (rpeaks):

```
# Import packages
import biosppy
import pyhrv.nonlinear as nl

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, _, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute Sample Entropy using R-peak series
results = nl.sampen(rpeaks=rpeaks)
```

6.5.3 Detrended Fluctuation Analysis: dfa()

`pyhrv.nonlinear.dfa(nni=None, rpeaks=None, short=None, long=None, show=True, figsize=None, legend=False)`

Function Description

Conducts Detrended Fluctuation Analysis (DFA) for short and long-term fluctuations of a NNI series.

An example plot of the DFA is shown below:

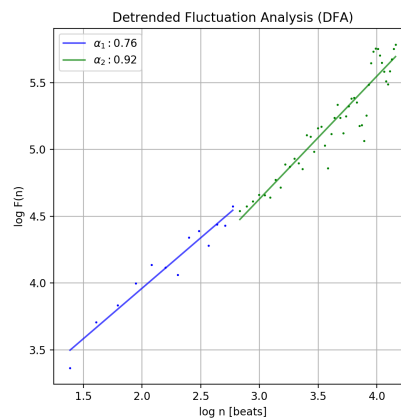


Fig. 33: Detrended Fluctuation Analysis plot.

Input Parameters

- `nni` (array): NN intervals in [ms] or [s].
- `rpeaks` (array): R-peak times in [ms] or [s].
- `short` (array, optional): Interval limits of the short-term fluctuations (default: None: [4, 16])
- `long` (array, optional): Interval limits of the long-term fluctuations (default: None: [17, 64])
- `show` (bool, optional): If True, shows DFA plot (default: True)
- `figsize` (array, optional): 2-element array with the `matplotlib` figure size `figsize`. Format: `figsize=(width, height)` (default: will be set to (6, 6) if input is None).
- `legend` (bool, optional): If True, adds legend with `alpha1` and `alpha2` values to the DFA plot (default: True)

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `dfa_short` (float): Alpha value of the short-term fluctuations (`alpha1`)
- `dfa_long` (float): Alpha value of the long-term fluctuations (`alpha2`)

See also:

The `biosppy.utils.ReturnTuple` Object

Raises `TypeError`: If `tolerance` is not a numeric value

Computation

This parameter is computed using the `nolds.dfa()` function.

See also:

[Nolds Documentation - DFA](#)

The default short- and long-term fluctuation intervals have been set to HRV suitable intervals.

Application Notes

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

The DFA cannot be computed if the number of NNI samples is lower than the specified short- and/or long-term fluctuation intervals. In this case, an empty plot with the information “*Insufficient number of NNI samples for DFA*” will be returned:

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()` function.

See also:

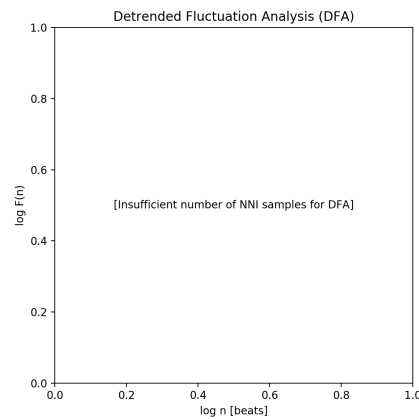


Fig. 34: Resulting plot if there are not enough NNI samples for the DFA.

- *What may help when matplotlib blocks your code from being executed*
- *More information about the matplotlib Interactive Mode*
- *More information about matplotlib Backends*

Examples & Tutorials

The following example code demonstrates how to use this function and how access the results stored in the returned `biosppy.utils.ReturnTuple` object.

You can use NNI series (`nni`) to compute the parameters:

```
# Import packages
import pyhrv
import pyhrv.nonlinear as nl

# Load sample data
nni = pyhrv.utils.load_sample_nni()

# Compute DFA using NNI series
results = nl.dfa(nni)

# Print DFA alpha values
print(results['dfa_short'])
print(results['dfa_long'])
```

Alternatively, you can use R-peak series (`rpeaks`):

```
# Import packages
import biosppy
import pyhrv.time_domain as td

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, _, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]
```

(continues on next page)

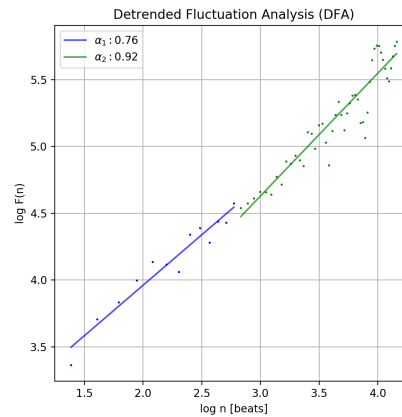


Fig. 35: Detrended Fluctuation Analysis plot.

(continued from previous page)

```
# Compute DFA using R-peak series
results = nl.dfa(rpeaks=t[rpeaks])
```

6.5.4 Domain Level Function: `nonlinear()`

```
pyhrv.nonlinear.frequency_domain(nn=None, rpeaks=None, signal=None, sam-
    pling_rate=1000., show=False, kwargs_poincare={},
    kwargs_sampen={}, kwargs_dfa{})
```

Function Description

Computes all the nonlinear HRV parameters of the Nonlinear module and returns them in a single ReturnTuple object.

See also:

The individual parameter functions of this module for more detailed information about the computed parameters:

- *Poincaré*: `poincare()`
- *Sample Entropy*: `sample_entropy()`
- *Detrended Fluctuation Analysis*: `dfa()`

Input Parameters

- `signal` (array): ECG signal
- `nnt` (array): NN intervals in [ms] or [s]
- `rpeaks` (array): R-peak times in [ms] or [s]
- `fbands` (dict, optional): Dictionary with frequency band specifications (default: None)
- `show` (bool, optional): If true, show all PSD plots.
- `kwargs_poincare` (dict, optional): Dictionary containing the kwargs for the ‘poincare’ function
- `kwargs_sampen` (dict, optional): Dictionary containing the kwargs for the ‘sample_entropy’ function
- `kwargs_dfa` (dict, optional): Dictionary containing the kwargs for the ‘dfa’ function

Important: This function computes the nonlinear parameters using either the `signal`, `nni`, or `rpeaks` data. Provide only one type of data, as not all three types need to be passed as input argument

Returns (ReturnTuple Object) The results of this function are returned in a `biosppy.utils.ReturnTuple` object. This function returns the frequency parameters computed with all three PSD estimation methods. You can access all the parameters using the following keys (X = one of the methods 'fft', 'ar', 'lomb'):

- `poincare_plot` (matplotlib figure object): Poincaré plot figure
- `sd1` (float): Standard deviation (SD1) of the major axis
- `sd2` (float): Standard deviation (SD2) of the minor axis
- `sd_ratio` (float): Ratio between SD1 and SD2 (SD2/SD1)
- `ellipse_area` (float): Area S of the fitted ellipse
- `sample_entropy` (float): Sample entropy of the NNI series
- `dfa_short` (float): Alpha value of the short-term fluctuations (alpha1)
- `dfa_long` (float): Alpha value of the long-term fluctuations (alpha2)

See also:

The `biosppy.utils.ReturnTuple` Object

Application Notes

It is not necessary to provide input data for `signal`, `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`signal`, `nni` **or** `rpeaks`). The input data will be prioritized in the following order, in case multiple inputs are provided:

1. `signal`, 2. `nni`, 3. `rpeaks`.

`nni` or `rpeaks` data provided in seconds [s] will automatically be converted to `nni` data in milliseconds [ms].

See also:

Section *NN Format*: `nn_format()` for more information.

Use the `kwargs_poincare` dictionary to pass function specific parameters for the `poincare()` function. The following keys are supported:

- `ellipse` (bool, optional): If True, shows fitted ellipse in plot (default: True)
- `vectors` (bool, optional): If True, shows SD1 and SD2 vectors in plot (default: True)
- `legend` (bool, optional): If True, adds legend to the Poincaré plot (default: True)
- `marker` (str, optional): NNI marker in plot (must be compatible with the matplotlib markers (default: 'o'))

Use the `kwargs_sampen` dictionary to pass function specific parameters for the `sample_entropy()` function. The following keys are supported:

- `dim` (int, optional): Entropy embedding dimension (default: 2)
- `tolerance` (int, float, optional): Tolerance distance for which the two vectors can be considered equal (default: `std(NNI)`)

Use the `kwargs_dfa` dictionary to pass function specific parameters for the `dfa()` function. The following keys are supported:

- `short` (array, optional): Interval limits of the short-term fluctuations (default: None: [4, 16])
- `long` (array, optional): Interval limits of the long-term fluctuations (default: None: [17, 64])

- `legend` (bool, optional): If True, adds legend to the Poincaré plot (default: True)

Important: The following input data is equally set for all the 3 methods using the input parameters of this function without using the kwargs dictionaries.

Defining these parameters/this specific input data individually in the kwargs dictionaries will have no effect:

- `show` (bool, optional): If True, show Poincaré plot (default: True)
- `figsize` (array, optional): Matplotlib figure size; Format: `figsize=(width, height)` (default: None: (6, 6))

Any key or parameter in the kwargs dictionaries that is not listed above will have no effect on the functions.

Important: This function generates `matplotlib` plot figures which, depending on the backend you are using, can interrupt your code from being executed whenever plot figures are shown. Switching the backend and turning on the `matplotlib` interactive mode can solve this behavior.

In case it does not - or if switching the backend is not possible - close all the plot figures to proceed with the execution of the rest your code after the `plt.show()` function.

See also:

- [What may help when matplotlib blocks your code from being executed](#)
 - [More information about the matplotlib Interactive Mode](#)
 - [More information about matplotlib Backends](#)
-

Examples & Tutorials

The following example codes demonstrate how to use the `nonlinear()` function.

You can choose either the ECG signal, the NNI series or the R-peaks as input data for the PSD estimation and parameter computation:

```
# Import packages
import biosppy
import pyhrv.nonlinear as nl
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNI series
nni = tools.nn_intervals(t[rpeaks])

# OPTION 1: Compute using the ECG Signal
signal_results = nl.nonlinear(signal=filtered_signal)

# OPTION 2: Compute using the R-peak series
rpeaks_results = nl.nonlinear(rpeaks=t[rpeaks])

# OPTION 3: Compute using the
nni_results = nl.nonlinear(nni=nni)
```

The use of this function generates plots that are similar to the plots below:

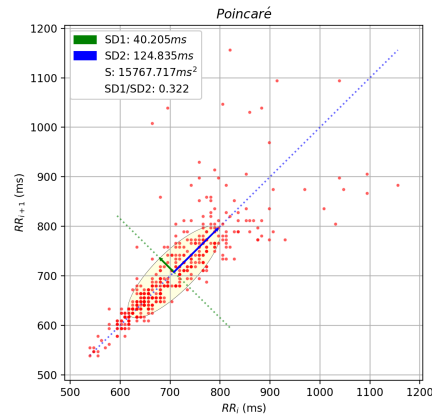


Fig. 36: Sample Poincaré scatter plot.

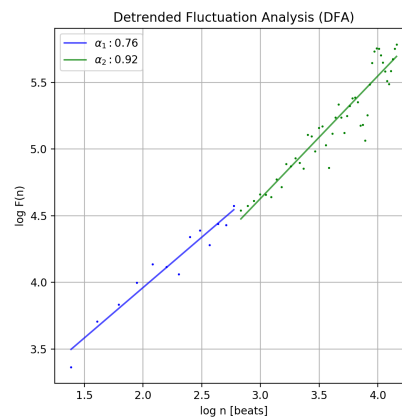


Fig. 37: Sample Detrended Fluctuation Analysis plot.

Using the `nonlinear()` function does not limit you in specifying input parameters for the individual nonlinear functions. Define the compatible input parameters in Python dictionaries and pass them to the `kwargs` input dictionaries of this function (see this functions **Application Notes** for a list of compatible parameters):

```
# Import packages
import biosppy
import pyhrv.nonlinear as nl

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Define input parameters for the 'poincare()' function
kwargs_poincare = {'ellipse': True, 'vectors': True, 'legend': True, 'markers': 'o'}

# Define input parameters for the 'sample_entropy()' function
kwargs_sampen = {'dim': 2, 'tolerance': 0.2}
```

(continues on next page)

(continued from previous page)

```
# Define input parameters for the 'dfa()' function
kwargs_dfa = {'short': [4, 16] , 'long': [17, 64]}

# Compute PSDs using the ECG Signal
signal_results = fd.frequency_domain(signal=signal, show=True,
    kwargs_poincare=kwargs_poincare, kwargs_sampen=kwargs_sampen, kwargs_dfa=kwargs_
    ↪dfa)
```

pyHRV is robust against invalid parameter keys. For example, if an invalid input parameter such as *nfft* is provided with the *kwargs_poincare* dictionary, this parameter will be ignored and a warning message will be issued.

```
# Define custom input parameters using the kwargs dictionaries
kwargs_poincare = {
    'ellipse': True,      # Valid key, will be used
    'nfft': 2**8          # Invalid key for the time domain, will be ignored
}

# Compute HRV parameters
nl.nonlinear(nni=nni, kwargs_poincare=kwargs_poincare)
```

This will trigger the following warning message.

Warning: *Unknown kwargs for 'poincare()': nfft. These kwargs have no effect.*

6.6 Tools Module

The *Tools Module* contains general purpose functions and key functionalities (e.g. computation of NNI series) for the entire HRV package, among other useful features for HRV analysis workflow (e.g. HRV export/import).

See also:

[pyHRV Tools Module source code](#)

Module Contents

- *Tools Module*
 - *NN Intervals:* `nn_intervals()`
 - *NN Interval Differences:* `nn_diff()`
 - *Heart Rate:* `heart_rate()`
 - *Plot ECG:* `plot_ecg()`
 - *Tachogram:* `tachogram()`
 - *Heart Rate Heatplot:* `hr_heatplot()`
 - *HRV Reports:* `hrv_report()`
 - *HRV Export:* `hrv_export()`
 - *HRV Import:* `hrv_import()`
 - *Radar Chart:* `radar_chart()`

6.6.1 NN Intervals: `nn_intervals()`

`pyhrv.tools.nn_intervals(rpeaks=None)`

Function Description

Computes the series of NN intervals [ms] from a series of successive R-peak locations.

Input Parameters

- `rpeaks` (array): R-peak times in [ms] or [s].

Returns

- `nni` (array): Series of NN intervals in [ms].

Computation

The NN interval series is computed from the R-peak series as follows:

$$NNI_j = R_{j+1} - R_j$$

for $0 \leq j \leq (n - 1)$

with:

- NNI_j : NN interval j
- R_j : Current R-peak j
- R_{j+1} : Successive R-peak j+1
- n : Number of R-peaks

Application Notes

The `nni` series will be returned in [ms] format, even if the `rpeaks` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

Example

The following example code demonstrates how to use this function:

```
# Import packages
import biosppy
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNI parameters
nni = tools.nn_intervals(t[rpeaks])
```

6.6.2 NN Interval Differences: `nn_diff()`

`pyhrv.tools.nn_diff(nn=None)`

Function Description

Computes the series of NN interval differences [ms] from a series of successive NN intervals.

Input Parameters

- `nni` (array): NNI series in [ms] or [s].

Returns

- `nn_diff_` (array): Series of NN interval differences in [ms].

Computation

The NN interval series is computed from the R-peak series as follows:

$$\Delta NNI_j = NNI_{j+1} - NNI_j$$

for $0 \leq j \leq (n - 1)$

with:

- ΔNNI_j : NN interval j
- NNI_j : Current NNI j
- NNI_{j+1} : Successive NNI j+1
- n : Number of NNI

Application Notes

The `nn_diff_` series will be returned in [ms] format, even if the `nni` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

Example

The following example code demonstrates how to use this function:

```
# Import packages
import biosppy
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

# Compute NNI parameters
nni = tools.nn_intervals(t[rpeaks])

# Compute NNI differences
delta_nni = tools.nn_diff(nni)
```

6.6.3 Heart Rate: `heart_rate()`

`pyhrv.tools.heart_rate(nni=None, rpeaks=None)`

Function Description

Computes a series of Heart Rate values in [bpm] from a series of NN intervals or R-peaks in [ms] or [s] or the HR from a single NNI.

Input Parameters

- `nni` (int, float, array): NN interval series in [ms] or [s]
- `rpeaks` (array): R-peak locations in [ms] or [s]

Returns

- `hr` (array): Series of NN intervals in [ms].

Computation

The Heart Rate series is computed as follows:

$$HR_j = \frac{60000}{NNI_j}$$

for $0 \leq j \leq n$

with:

- HR_j : Heart rate j (in [bpm])
- NNI_j : NN interval j (in [ms])
- n : Number of NN intervals

Application Notes

The input `nni` series will be converted to [ms], even if the `rpeaks` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

Example

The following example code demonstrates how to use this function:

```
# Import packages
import numpy as np
import pyhrv.tools as tools
import pyhrv.utils as utils

# Load sample data
nn = pyhrv.utils.load_sample_nni()

# Compute Heart Rate series
hr = tools.heart_rate(nn)
```

It is also possible to compute the HR from a single NNI:

```
# Compute Heart Rate from a single NNI
hr = tools.heart_rate(800)
# here: hr = 75 [bpm]
```

Attention: In this case, the input NNI must be provided in [ms] as the [s] to [ms] conversion is only conducted for series of NN Intervals.

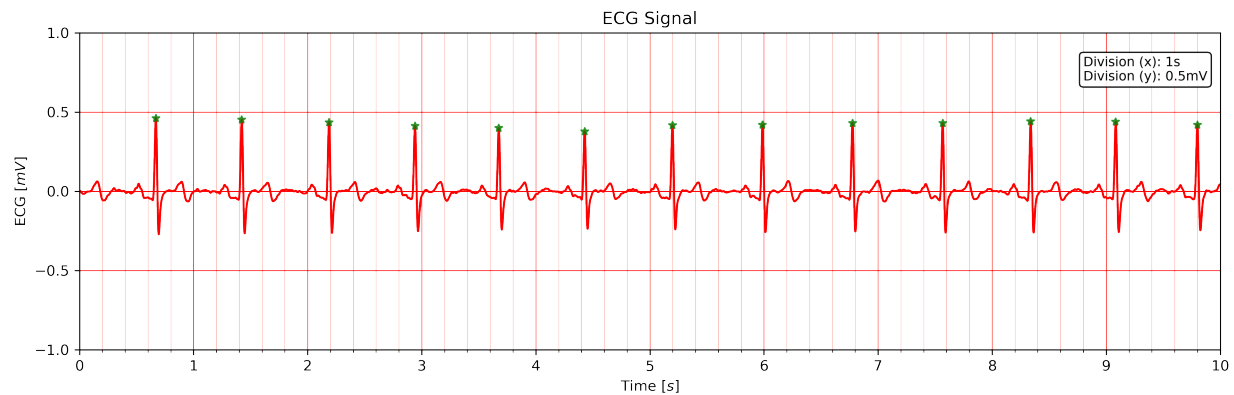
6.6.4 Plot ECG: `plot_ecg()`

```
pyhrv.tools.plot_ecg (signal=None, t=None, samplin_rate=1000., interval=None, rpeaks=True, fig-  
size=None, title=None, show=True)
```

Function Description

Plots ECG signal on a medical grade ECG paper-like figure layout.

An example of an ECG plot generated by this function can be seen here:



The x-Division does automatically adapt to the visualized interval (e.g., 10s interval -> 1s, 20s interval -> 2s, ...).

Input Parameters

- `signal` (array): ECG signal (filtered or unfiltered)
- `t` (array, optional): Time vector for the ECG signal (default: None)
- `sampling_rate` (int, float, optional): Sampling rate of the acquired signal in [Hz] (default: 1000Hz)
- `interval` (array, optional): Visualization interval of the ECG signal plot (default: [0s, 10s])
- `rpeaks` (bool, optional): If True, marks R-peaks in ECG signal (default: True)
- `figsize` (array, optional): Matplotlib figure size (width, height) (default: None: (12, 4))
- `title` (str, optional): Plot figure title (default: None)
- `show` (bool, optional): If True, shows the ECG plot figure (default: True)

Returns

- `fig_ecg` (matplotlib figure object): Matplotlib figure of the ECG plot

Application Notes

The input `nni` series will be converted to [ms], even if `nni` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

This functions marks, by default, the detected R-peaks. Use the `rpeaks` input parameter to turn on (`rpeaks=True`) or to turn of (`rpeaks=False`) the visualization of these markers.

Important: This parameter will have no effect if the number of R-peaks within the visualization interval is greater than 50. In this case, for reasons of plot clarity, no R-peak markers will be added to the plot.

The time axis scaling will change depending on the duration of the visualized interval:

- t in [s] if visualized duration ≤ 60 s
- t in [mm:ss] (minutes:seconds) if $60\text{s} < \text{visualized duration} \leq 1\text{h}$
- t in [hh:mm:ss] (hours:minutes:seconds) if visualized duration $> 1\text{h}$

Example

```
# Import
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Plot ECG
tools.plot_ecg(signal)
```

The plot of this example should look like the following plot:

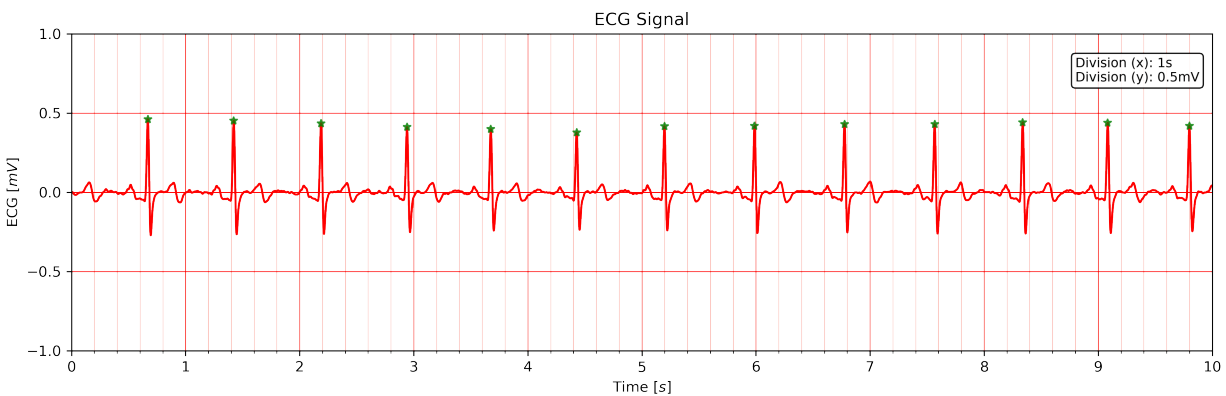


Fig. 38: Default visualization interval of the `plot_ecg()` function.

Use the `interval` input parameter to change the visualization interval using a 2-element array (`[lower_interval_limit, upper_interval_limit]`; default: 0s to 10s). Additionally, use the `rpeaks` parameter to toggle the R-peak markers.

The following code sets the visualization interval from 0s to 20s and hides the R-peak markers:

```
# Plot ECG
tools.plot_ecg(signal, interval=[0, 20], rpeaks=False)
```

The plot of this example should look like the following plot:

Use the `title` input parameter to add titles to the ECG plot:

```
# Plot ECG
tools.plot_ecg(signal, interval=[0, 20], rpeaks=False, title='This is a Title')
```

6.6.5 Tachogram: `tachogram()`

`pyhrv.tools.tachogram` (`signal=None`, `nn=None`, `rpeaks=None`, `sampling_rate=1000.`, `hr=True`, `interval=None`, `title=None`, `figsize=None`, `show=True`)

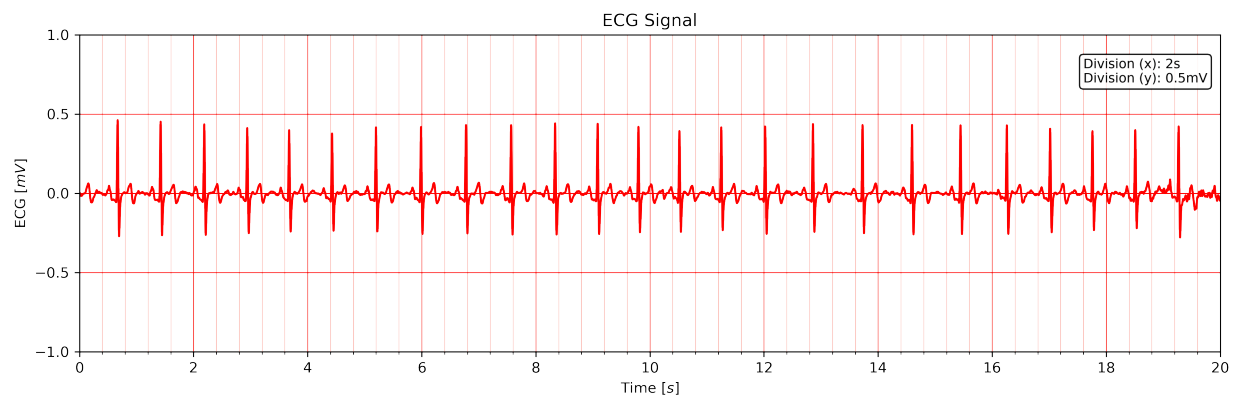


Fig. 39: Visualizing the first 20 seconds of the ECG signal without R-peak markers.

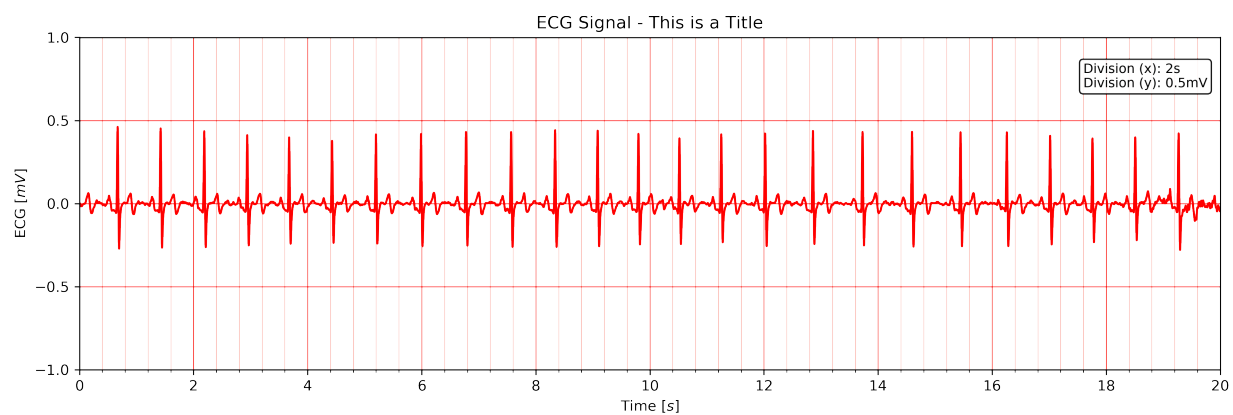
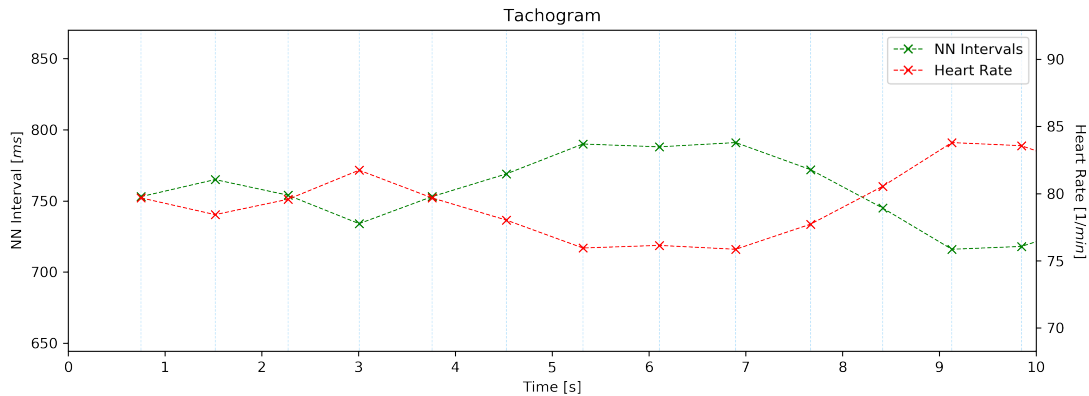


Fig. 40: ECG plot with custom title.

Function Description

Plots Tachogram (NNI & HR) of an ECG signal, NNI or R-peak series.

An example of a Tachogram plot generated by this function can be seen here:



Input Parameters

- `signal` (array): ECG signal (filtered or unfiltered)
- `nni` (array): NN interval series in [ms] or [s]
- `rpeaks` (array): R-peak locations in [ms] or [s] - `t` (array, optional): Time vector for the ECG signal (default: None)
- `sampling_rate` (int, optional): Sampling rate in [hz] of the ECG signal (default: 1000Hz)
- `hr` (bool, optional): If True, plot HR series in [bpm] on second axis (default: True)
- `interval` (array, optional): Visualization interval of the Tachogram plot (default: None: [0s, 10s])
- `title` (str, optional): Optional plot figure title (default: None)
- `figsize` (array, optional): Matplotlib figure size (width, height) (default: None: (12, 4))
- `show` (bool, optional): If True, shows the ECG plot figure (default: True)

Returns

- `fig` (matplotlib figure object): Matplotlib figure of the Tachogram plot.

Application Notes

The input `nni` series will be converted to [ms], even if the `rpeaks` or `nni` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

Example

The following example demonstrates how to load an ECG signal.

```
# Import
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]
```

(continues on next page)

(continued from previous page)

```
# Plot Tachogram
tools.tachogram(signal)
```

Alternatively, use R-peak data to plot the histogram...

```
# Import
import biosppy
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]
# Plot Tachogram
tools.tachogram(rpeaks=t[rpeaks])
```

... or using directly the NNI series...

```
# Compute NNI intervals from the R-peaks
nni = tools.nn_intervals(t[rpeaks])

# Plot Tachogram
tools.tachogram(nni=nni)
```

The plots generated by the examples above should look like the plot below:

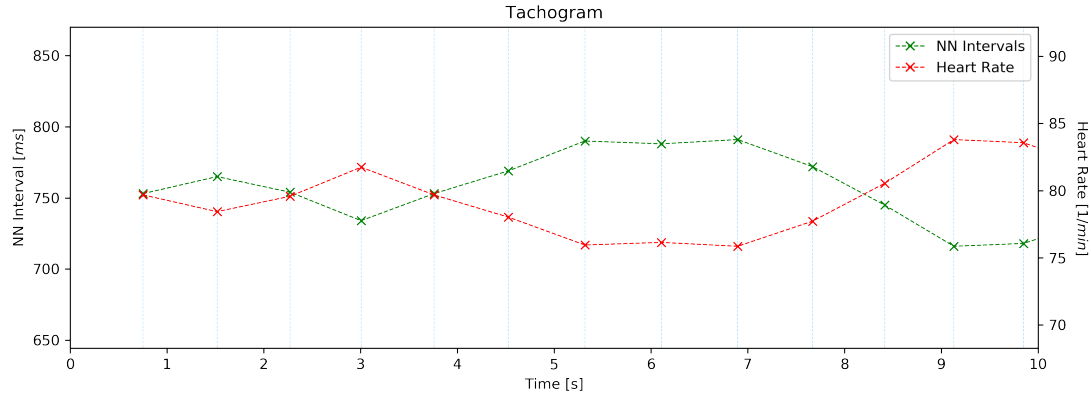


Fig. 41: Tachogram with default visualization interval.

Use the interval input parameter to change the visualization interval (default: 0s to 10s; here: 0s to 20s):

```
# Plot ECG
tools.tachogram(signal=signal, interval=[0, 20])
```

The plot of this example should look like the following plot:

Note: Interval limits which are out of bounce will automatically be corrected.

Example:

- lower limit < 0 -> lower limit = 0

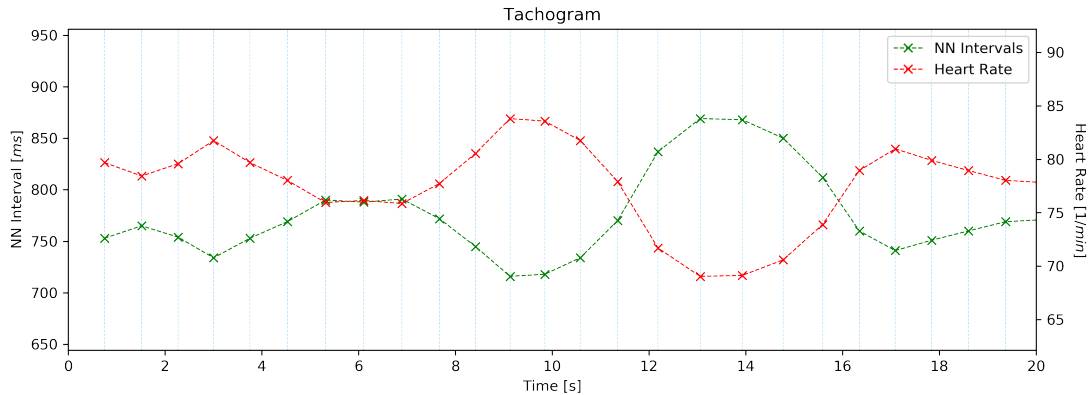


Fig. 42: Tachogram with custom visualization interval.

- upper limit > maximum ECG signal duration -> upper limit = maximum ECG signal duration

Set the `hr` parameter to `False` in case only the NNI Tachogram is needed:

```
# Plot ECG
tools.tachogram(signal=signal, interval=[0, 20], hr=False)
```

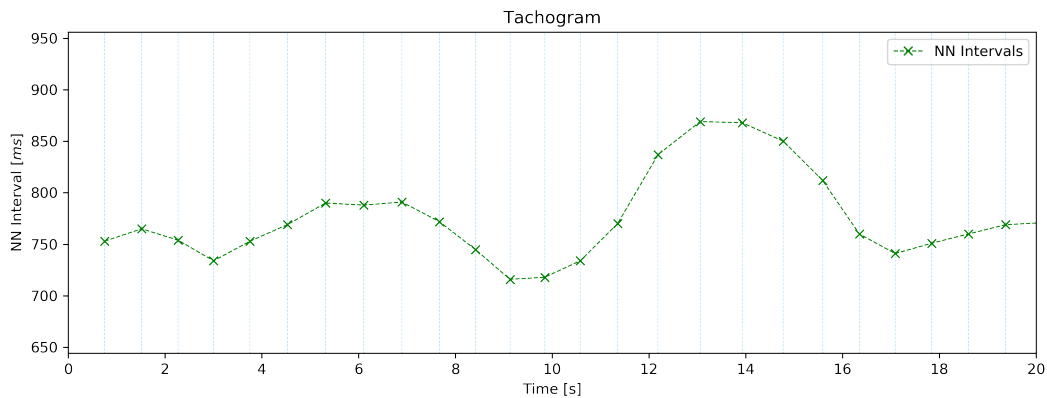


Fig. 43: Tachogram of the NNI series only.

The time axis scaling will change depending on the duration of the visualized interval:

- `t` in [s] if visualized duration <= 60s
- `t` in [mm:ss] (minutes:seconds) if 60s < visualized duration <= 1h
- `t` in [hh:mm:ss] (hours:minutes:seconds) if visualized duration > 1h

6.6.6 Heart Rate Heatplot: `hr_heatplot()`

`pyhrv.tools.hr_heatplot` (`signal=None`, `nn=None`, `rpeaks=None`, `sampling_rate=1000.`, `hr=True`, `interval=None`, `title=None`, `figsize=None`, `show=True`)

Function Description

Graphical visualization & classification of HR performance based on normal HR ranges by age and gender.

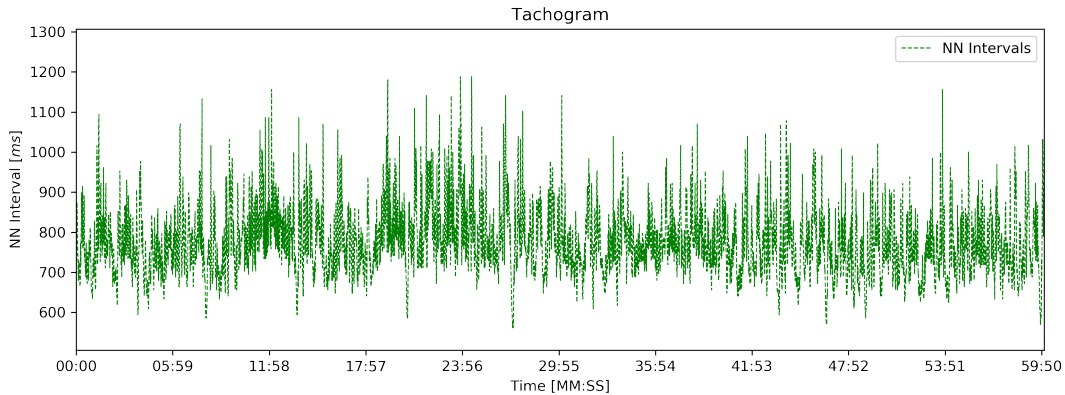
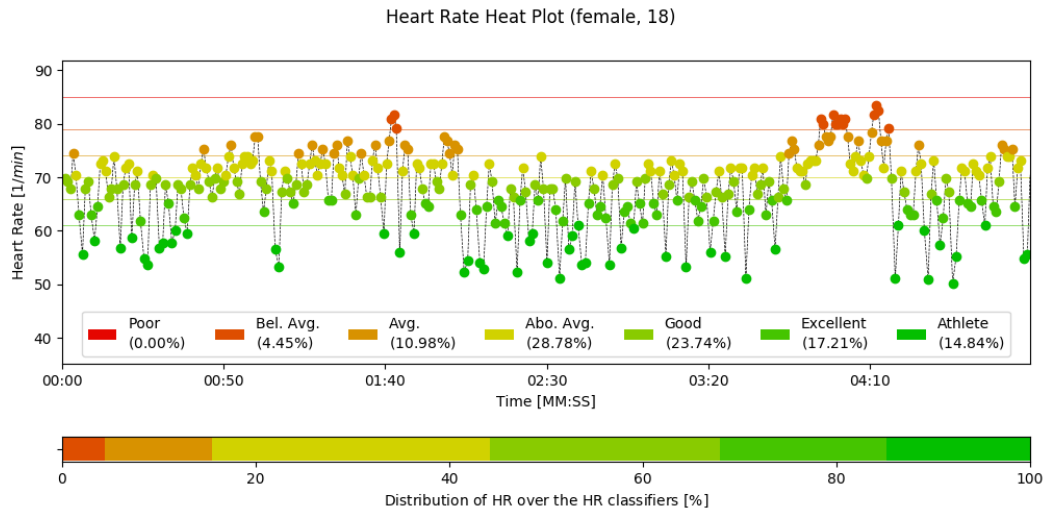


Fig. 44: Tachogram of an ~1h NNI series.

An example of a Heart Rate Heatplot generated by this function can be seen here:



Input Parameters

- `nni` (array): NN interval series in [ms] or [s]
- `rpeaks` (array): R-peak locations in [ms] or [s] - `t` (array, optional): Time vector for the ECG signal (default: None)
- `signal` (array): ECG signal (filtered or unfiltered)
- `sampling_rate` (int, optional): Sampling rate in [hz] of the ECG signal (default: 1000Hz)
- `age` (int, float): Age of the subject (default: 18)
- `gender` (str): Gender of the subject ('m', 'male', 'f', 'female'; default: 'male')
- `interval` (list, optional): Sets visualization interval of the signal (default: [0, 10])
- `figsize` (array, optional): Matplotlib figure size (weight, height) (default: (12, 4))
- `show` (bool, optional): If True, shows plot figure (default: True)

Returns

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following keys below (on the left) to index the results:

- `hr_heatplot` (matplotlib figure object): Matplotlib figure of the Heart Rate Heatplot.

Application Notes

The input `nni` series will be converted to [ms], even if the `rpeaks` or `nni` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

Interval limits which are out of bounce will automatically be corrected:

- lower limit < 0 -> lower limit = 0
- upper limit > maximum ECG signal duration -> upper limit = maximum ECG signal duration

The time axis scaling will change depending on the duration of the visualized interval:

- t in [s] if visualized duration <= 60s
- t in [mm:ss] (minutes:seconds) if 60s < visualized duration <= 1h
- t in [hh:mm:ss] (hours:minutes:seconds) if visualized duration > 1h

Example

The following example demonstrates how to load an ECG signal.

```
# Import
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Plot Heart Rate Heatplot using an ECG signal
tools.hr_heatplot(signal=signal)
```

Alternatively, use R-peak or NNI data to plot the HR Heatplot...

```
# Import
import biosppy
import pyhrv.tools as tools

# Load sample ECG signal
signal = np.loadtxt('./files/SampleECG.txt')[:, -1]

# Get R-peaks series using biosppy
t, filtered_signal, rpeaks = biosppy.signals.ecg.ecg(signal)[:3]

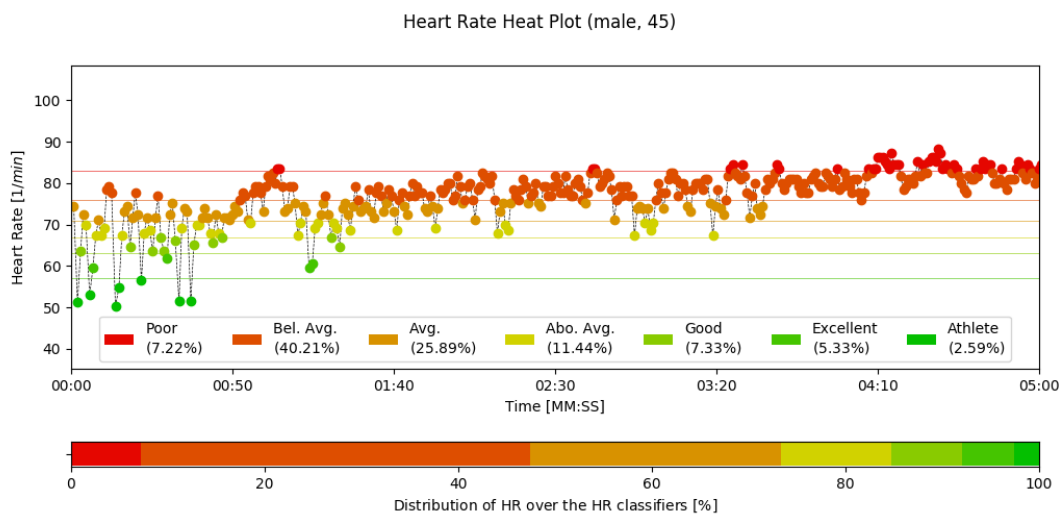
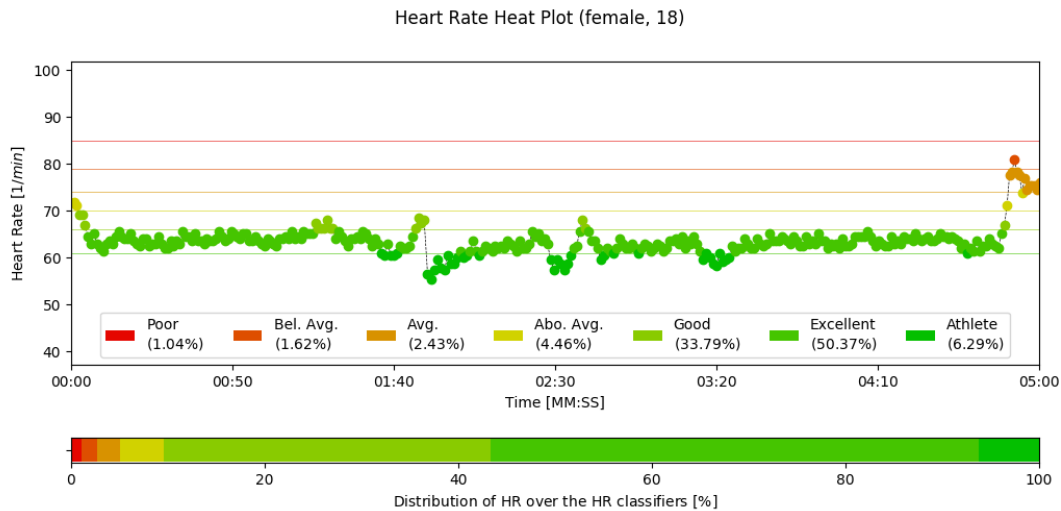
tools.hr_heatplot(rpeaks=t[rpeaks])
```

... or using directly the NNI series...

```
# Compute NNI intervals from the R-peaks
nni = tools.nn_intervals(t[rpeaks])

# Plot HR Heatplot using the NNIs
tools.hr_heatplot(signal=signal)
```

The following plots are example results of this function:



6.6.7 HRV Reports: `hrv_report()`

```
pyhrv.tools.hrv_report (results=None, path=None, rfile=None, nn=None, info={}, file_format='txt',  
                        delimiter=';', hide=False, plots=False)
```

Function Description

Generates HRV report (in .txt or .csv format) of the provided HRV results. You can find a sample report generated with this function [here](#).

Input Parameters

- `results` (dict, ReturnTuple object): Computed HRV parameter results
- `path` (str): Absolute path of the output directory
- `rfile` (str): Output file name
- `nni` (array, optional): NN interval series in [ms] or [s]
- `info` (dict, optional): Dictionary with HRV metadata
- `file_format` (str, optional): Output file format, select 'txt' or 'csv' (default: 'txt')
- `delimiter` (str, optional): Delimiter separating the columns in the report (default: ';')
- `hide` (bool, optional): Hide parameters in report that have not been computed
- `plots` (bool, optional): If True, save plot figures in .png format

Note: The `info` dictionary can contain the following metadata:

- key: `file` - Name of the signal acquisition file
- key: `device` - ECG acquisition device
- key: `identifier` - ECG acquisition device identifier (e.g. MAC address)
- key: `fs` - Sampling rate used during ECG acquisition
- key: `resolution` - Resolution used during acquisition

Any other key will be ignored.

Important: It is recommended to use absolute file paths when using the `path` parameter to ensure the correct functionality of this function.

Raises

- `TypeError`: If no HRV results are provided
- `TypeError`: If no file or directory path is provided
- `TypeError`: If the specified selected file format is not supported
- `IOError`: If the selected output file or directory does not exist

Application Notes

This function uses the weak `_check_fname()` function found in this module to prevent the (accidental) overwriting of existing HRV reports. If a file with the file name `rfile` does exist in the specified `path`, then the file name will be incremented.

For instance, if a report file with the name *SampleReport.txt* exists, this file will not be overwritten, instead, the file name of the new report will be incremented to *SampleReport_1.txt*.

If the file with the file name *SampleReport_1.txt* exists, the file name of the new report will be incremented to *SampleReport_2.txt*, and so on...

Important: The maximum supported number of file name increments is limited to 999 files, i.e., using the example above, the implemented file protection mechanisms will go up to *SampleReport_999.txt*.

If no file name is provided, an automatic file name with a time stamp will be generated for the generated report (*hrv_report_YYYY-MM-DD_hh-mm-ss.txt* or *hrv_report_YYYY-MM-DD_hh-mm-ss.txt*).

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv
import numpy as np

# Load Sample NNI series (~5min)
nni = pyhrv.utils.load_sample_nni()

# Compute HRV results
results = pyhrv.hrv(nn=nni)

# Create HRV Report
pyhrv.tools.hrv_report(results, rfile='SampleReport', path='/my/favorite/path/')
```

This generates a report looking like the one below:

See also:

- [Sample report in .txt format](#)
- [Sample report in .csv format](#)

6.6.8 HRV Export: `hrv_export()`

`pyhrv.tools.hrv_export` (*results=None, path=None, efile=None, comment=None, plots=False*)

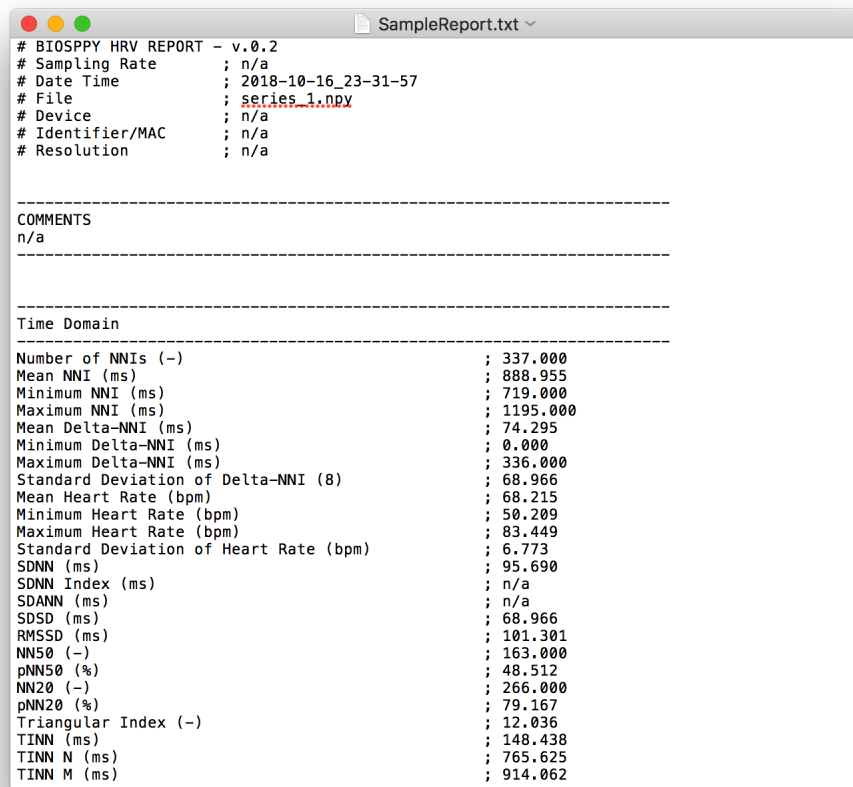
Function Description

Exports HRV results into a JSON file. You can find a sample export generated with this function [here](#).

Input Parameters

- `results` (dict, ReturnTuple object): Computed HRV parameter results
- `path` (str): Absolute path of the output directory
- `efile` (str): Output file name
- `comment` (str, optional): Optional comment
- `plots` (bool, optional): If True, save figures of the results in .png format

Important: It is recommended to use absolute file paths when using the `path` parameter to ensure the correct operation of this function.



```

# BIOSPPY HRV REPORT - v.0.2
# Sampling Rate      ; n/a
# Date Time         ; 2018-10-16_23-31-57
# File              ; series_1.npy
# Device            ; n/a
# Identifier/MAC     ; n/a
# Resolution         ; n/a

-----

COMMENTS
n/a

-----

Time Domain
-----
Number of NNIs (-)                ; 337.000
Mean NNI (ms)                     ; 888.955
Minimum NNI (ms)                   ; 719.000
Maximum NNI (ms)                   ; 1195.000
Mean Delta-NNI (ms)                ; 74.295
Minimum Delta-NNI (ms)             ; 0.000
Maximum Delta-NNI (ms)             ; 336.000
Standard Deviation of Delta-NNI (8) ; 68.966
Mean Heart Rate (bpm)              ; 68.215
Minimum Heart Rate (bpm)           ; 50.209
Maximum Heart Rate (bpm)           ; 83.449
Standard Deviation of Heart Rate (bpm) ; 6.773
SDNN (ms)                          ; 95.690
SDNN Index (ms)                    ; n/a
SDANN (ms)                         ; n/a
SDSD (ms)                          ; 68.966
RMSSD (ms)                         ; 101.301
NN50 (-)                           ; 163.000
pNN50 (%)                           ; 48.512
NN20 (-)                           ; 266.000
pNN20 (%)                           ; 79.167
Triangular Index (-)               ; 12.036
TINN (ms)                          ; 148.438
TINN N (ms)                        ; 765.625
TINN M (ms)                        ; 914.062

```

Returns

- `efile` (str): Absolute path of the output report file (may vary from the input data)

Raises

- `TypeError`: If no HRV results are provided
- `TypeError`: If no file or directory path is provided
- `TypeError`: If specified selected file format is not supported
- `IOError`: If the selected output file or directory does not exist

Application Notes

This function uses the weak `_check_fname()` function found in this module to prevent the (accidental) overwriting of existing HRV exports. If a file with the file name `efile` exists in the specified `path`, then the file name will be incremented.

For instance, if an export file with the name *SampleExport.json* exists, this file will not be overwritten, instead, the file name of the new export file will be incremented to *SampleExport_1.json*.

If the file with the file name *SampleExport_1.json* exists, the file name of the new export will be incremented to *SampleExport_2.json*, and so on.

Important: The maximum supported number of file name increments is limited to 999 files, i.e., using the example above, the implemented file protection mechanisms will go up to *SampleExport_999.json*.

If no file name is provided, an automatic file name with a time stamp will be generated for the generated report (*hrv_export_YYYY-MM-DD_hh-mm-ss.json*).

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv
import numpy as np
import pyhrv.tools as tools

# Load Sample NNI series (~5min)
nni = np.load('series_1.npy')

# Compute HRV results
results = pyhrv.hrv(nn=nni)

# Export HRV results
tools.hrv_export(results, efile='SampleExport', path='/my/favorite/path/')
```

See also:

- [Sample HRV export](#)

6.6.9 HRV Import: `hrv_import()`

`pyhrv.tools.hrv_import` (*hrv_file=None*)

Function Description

Imports HRV results stored in JSON files generated with the `'hrv_export()'`.

See also:

- *HRV Export*: `hrv_export()` function
- *Sample HRV export*

Input Parameters

- `hrv_file` (str, file handler): File handler or absolute string path of the HRV JSON file

Returns

- `output` (ReturnTuple object): All HRV parameters stored in a `biosppy.utils.ReturnTuple` object

Raises

- `TypeError`: If no file path or handler is provided

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv.tools as tools

# Import HRV results
hrv_results = tools.hrv_import('/path/to/my/HRVResults.json')
```

See also:

HRV keys file for a full list of HRV parameters and their respective keys.

6.6.10 Radar Chart: `radar_chart()`

```
pyhrv.tools.radar_chart(nni=None, rpeaks=None, comparison_nni=None, comparison_rpeaks=None, parameters=None, reference_label='Reference', comparison_label='Comparison', show=True, legend=True)
```

Function Description

Plots a radar chart of HRV parameters to visualize the evolution the parameters computed from a NNI series (e.g. extracted from an ECG recording while doing sports) compared to a reference/baseline NNI series (e.g. extracted from an ECG recording while at rest).

The radarchart normalizes the values of the reference NNI series with the values extracted from the baseline NNI this series being used as the 100% reference values.

Example:

- Reference NNI series: SDNN = 100ms → 100%
- Comparison NNI series: SDNN = 150ms → 150%

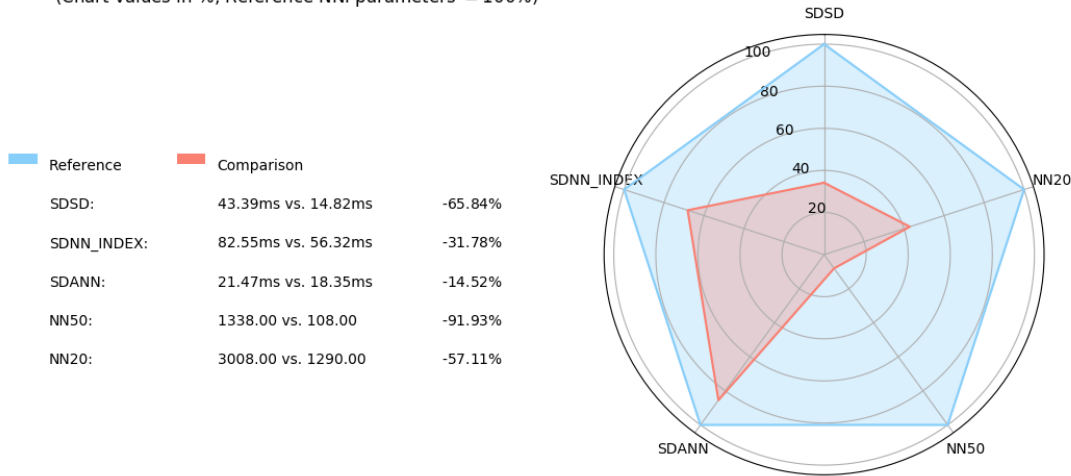
The radar chart is not limited by the number of HRV parameters to be included in the chart; it dynamically adjusts itself to the number of compared parameters.

An example of a Radar Chart plot generated by this function can be seen here:

Input Parameters

- `nni` (array): NN interval series in [ms] or [s] (default: None)
- `rpeaks` (array): R-peak locations in [ms] or [s] (default: None)

HRV Parameter Radar Chart
Reference NNI Series (lightskyblue) vs. Comparison NNI Series (salmon)
(Chart values in %, Reference NNI parameters $\pm 100\%$)



- `comparison_nni` (array): Comparison NNI series in [ms] or [s] (default: None)
- `comparison_rpeaks` (array): Comparison R-peak series in [ms] or [s] (default: None)
- `parameters` (list): List of pyHRV parameters (see `hrv_keys.json` file for a full list of available parameters)
- `reference_label` (str, optional): Plot label of the reference input data (e.g. 'ECG while at rest'; default: 'Reference')
- `comparison_label` (str, optional): Plot label of the comparison input data (e.g. 'ECG while running'; default: 'Comparison')
- `show` (bool, optional): If True, shows plot figure (default: True).
- `legend` (bool, optional): If true, add a legend with the computed results to the plot (default: True)

Returns (ReturnTuple Object)

The results of this function are returned in a `biosppy.utils.ReturnTuple` object. Use the following key below (on the left) to index the results:

- `reference_results` (dict): HRV parameters computed from the reference input data
- `comparison_results` (dict): HRV parameters computed from the comparison input data
- `radar_plot` (dict): Resulting radar chart plot figure

Raises

- `TypeError`: If an error occurred during the computation of a parameter
- `TypeError`: If no input data is provided for the baseline/reference NNI or R-Peak series
- `TypeError`: If no input data is provided for the comparison NNI or R-Peak series
- `TypeError`: If no selection of pyHRV parameters is provided
- `ValueError`: If less than 2 pyHRV parameters were provided

Application Notes

The input `nni` series will be converted to [ms], even if the `rpeaks` or `nni` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

It is not necessary to provide input data for `nni` **and** `rpeaks`. The parameter(s) of this function will be computed with any of the input data provided (`nni` **or** `rpeaks`). `nni` will be prioritized in case both are provided. This is both valid for the reference as for the comparison input series.

Example

The following example shows how to compute the radar chart from two NNI series (here one NNI series is split in half to generate 2 series):

```
# Import
import pyhrv.utils as utils
import pyhrv.tools as tools

# Load Sample Data
nni = utils.load_sample_nni()
reference_nni = nni[:300]
comparison_nni = nni[300:]

# Specify the HRV parameters to be computed
params = ['nni_mean', 'sdnn', 'rmssd', 'sdsd', 'nn50', 'nn20', 'sdl', 'fft_peak']

# Plot the Radar Chart
radar_chart(nni=ref_nni, comparison_nni=comparison_nni, parameters=params)
```

This generates the following radar chart:

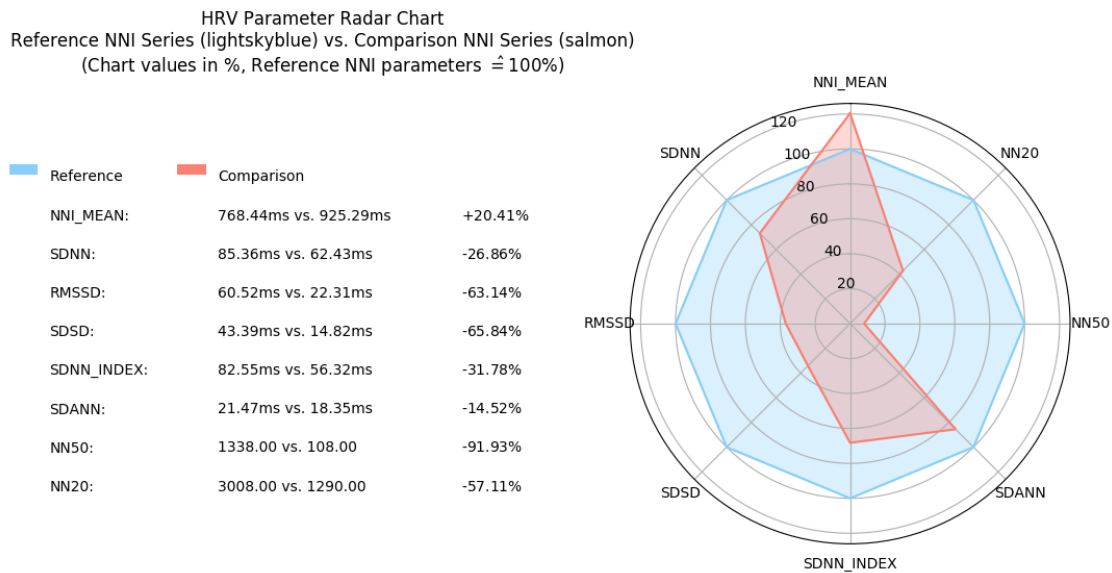


Fig. 45: Sample Radar Chart plot with 8 parameters.

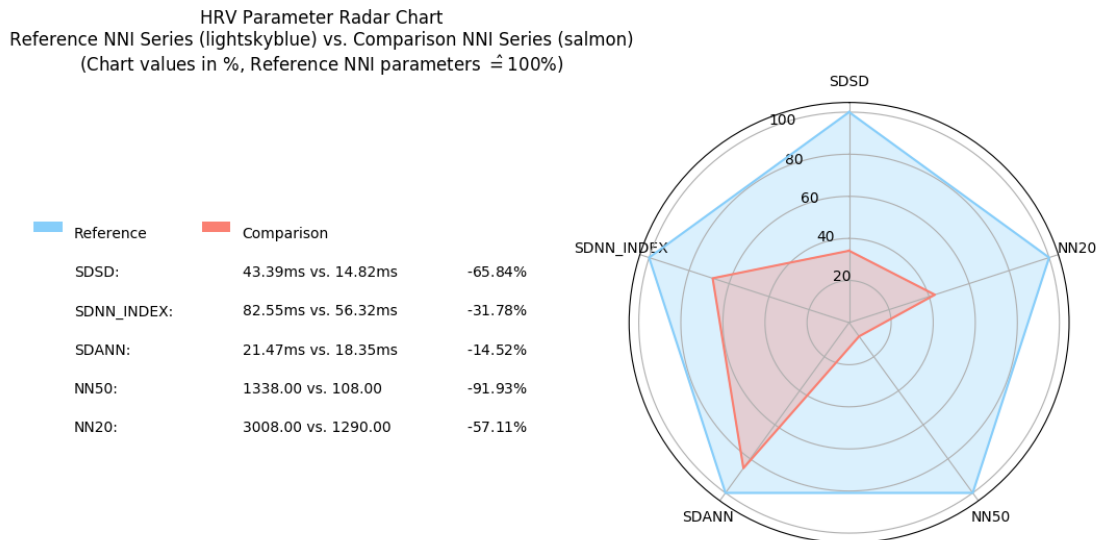
The `radar_chart()` function is not limited to a specific number of HRV parameters, as the Radar Chart will automatically be adjusted to the number of provided HRV parameters.

For instance, in the previous example, the input parameter list consisted of 8 HRV parameters. In the following example, the input parameter list consists of 5 parameters only:

```
# Specify the HRV parameters to be computed
params = ['nni_mean', 'sdnn', 'rmssd', 'sdsd', 'nn50', 'nn20', 'sd1', 'fft_peak']

# Plot the Radar Chart
radar_chart(nni=ref_nni, comparison_nni=comparison_nni, parameters=params)
```

... which generates the following Radar Chart:



6.7 Utils Module

The *Utils Module* contains general purpose functions (utilities) to support the features of the pyHRV toolbox incl. loading NNI sample data, check input data of HRV functions, segment arrays, check for duplicate file names to avoid accidental overwriting of files, and other utilities. These functions do not compute any HRV parameters nor provide any parameter-specific features (e.g. comparison plots).

See also:

[pyHRV Utils Module source code](#)

Module Contents

- *Utils Module*
 - Loading NNI Sample Data: `load_sample_nni()`
 - Loading the pyHRV Keys: `load_hrv_keys_json()`
 - Check Input: `check_input()`
 - NN Format: `nn_format()`

- Check Interval: `check_interval()`
- Segmentation: `segmentation()`
- Join Tuples: `join_tuples()`
- Standard Deviation: `std()`
- Time Vector: `time_vector()`

6.7.1 Loading NNI Sample Data: `load_sample_nni()`

```
pyhrv.utils.load_sample_nni(series='short')
```

Function Description

Returns a short-term (5min) or long-term (60min) series of sample NNI found in the `pyhrv/files/` directory.

These sample series were extracted from the [MIT-BIH NSRDB Database from physionet.org](#) and can be found in the `samples` folder.

Input Parameters

- `series` (str): If 'long', returns a 60min NNI series, if 'short' returns a 5min NNI series (default: 'short').

Returns

- `nni` (array): Series of NN intervals in [ms].

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv

# Load short sample series (5min)
nni = pyhrv.utils.load_sample_nni()

# Load long sample series (60min)
nni = pyhrv.utils.load_sample_nni(series='long')
```

6.7.2 Loading the pyHRV Keys: `load_hrv_keys_json()`

```
pyhrv.utils.load_hrv_keys_json()
```

Function Description

Loads the content of the 'hrv_keys.json' file found in the 'pyhrv/files/' directory.

Note: The `hrv_keys.json` file contains all the keys and additional information related to the computed HRV parameters. These keys are used throughout pyHRV when exporting and importing data.

For example, the `pyhrv.time_domain.sdn()` function returns a *ReturnTuple* object from the BioSPPy package in which the result of the function can be accessed using the `sdnn` key.

Input Parameters

- none

Returns

- `hrv_keys` (dict): Content of the `pyhrv/files/hrv_keys.json` file in a dictionary

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv

# Load content of the hrv_keys.json file
hrv_keys = pyhrv.utils.load_hrv_keys_json()
```

6.7.3 Check Input: `check_input()`

`pyhrv.utils.check_input` (*nn=None, rpeaks=None*)

Function Description

Checks if input series of NN intervals or R-peaks are provided and, if yes, returns a NN interval series in [ms] format.

Input Parameters

- `nni` (array): NN interval series in [ms] or [s] (default: None)
- `rpeaks` (array): R-peak locations in [ms] or [s] (default: None)

Returns

- `nni` (array): NN interval series in [s] (default: None)

Raises

- `TypeError`: If no R-peak data or NN intervals provided

Application Notes

This function is mainly used by the parameter computation functions of the `time_domain.py`, the `frequency_domain.py`, and the `nonlinear.py` modules.

The `nni` series will be returned in [ms] format, even if the input `rpeaks` or `nni` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

6.7.4 NN Format: `nn_format()`

`pyhrv.utils.nn_format` (*nni=None*)

Function Description

Checks the format of the NNI series and converts data in [s] to [ms] format. Additionally, it ensures that the data will be returned in the NumPy array format.

Input Parameters

- `nni` (array): NNI series [ms] or [s].

Returns

- `nni` (array): NNI series [ms] and NumPy array format.

Computation

The automatic [s] to [ms] conversion occurs on a threshold based identification whether the data is in [s] or [ms] format: if the maximum value of the input array is < 10 , then the data is assumed to be in [s] format.

This conversion process is based on the following two assumptions:

- Any interval data in [s] format ranges between 0.2s ($\hat{=}$ 300bpm) and 1.5s ($\hat{=}$ 40bpm). Any interval greater 1.5s is highly unlikely to occur, and even if it does, it does still not reach the specified maximum interval limit of 10s ($\hat{=}$ 6bpm)
- The provided NNI series has been filtered from NNI outliers caused by signal artifacts (e.g. ECG signal loss)

Note: It is important to filter the NNI series from the intervals caused by signal artifacts first, otherwise the returned series will be influenced by these NNI and distort all HRV parameter results.

Application Notes

The `nni` series will be returned in [ms] format, even if the `rpeaks` are provided in [s] format.

See also:

NN Format: `nn_format()` for more information about the [s] to [ms] conversion.

Example

The following example code demonstrates how to use this function:

Note: This functions is intended to be used by the parameter functions of pyHRV, an external use might not be appropriate.

```
# Import packages
import biosppy
import pyhrv
from opensignalsreader import OpenSignalsReader

# Load sample ECG signal stored in an OpenSignals file
signal = OpenSignalsReader('./samples/SampleECG.npy').signal('ECG')

# Get R-peak locations
rpeaks = biosppy.signals.ecg.ecg(signal)[2]

# Compute NNI parameters
nni = pyhrv.utils.nn_intervals(rpeaks)

# Confirm [ms] format
nni_in_ms = pyhrv.utils.nn_format(nni)
```

6.7.5 Check Interval: `check_interval()`

`pyhrv.utils.check_interval(interval=None, limits=None, default=None)`

Function Description

General purpose function that checks and verifies correctness of interval limits within optionally defined valid interval specifications and/or default values if no interval is specified.

This function can be used to set visualization intervals, check overlapping frequency bands, or for other similar purposes, and is intended to automatically catch possible error sources due to invalid intervals boundaries.

Input Parameters

- `interval` (array): Input interval [min, max] (default: None)
- `limits` (array): Minimum and maximum allowed interval limits (default: None)
- `default` (array): Specified default interval (e.g. if `interval` is None) (default: None)

Returns

- `interval` (array): Interval with correct(ed)/valid interval limits.

Raises

- `TypeError` If no input data is specified.
- `ValueError` If the input interval[s] have equal lower and upper limits.

Computation

The input data is provided as `interval = [int(lower_limit), int(upper_limit)]`. Depending on the limits, the following conditions should be met:

- If `lower_limit > upper_limit`: the interval limits will be switched to `interval = [upper_limit, lower_limit]`
- If `lower_limit == upper_limit`: raises `ValueError`

If minimum and maximum intervals are specified, i.e. `limit = [int(minimum), int(maximum)]`, the following additional actions may occur:

- If `lower_limit < minimum`: the lower limit will be set to the minimum allowed limit `lower_limit = minimum`
- If `upper_limit > maximum`: the upper limit will be set to the maximum allowed limit `upper_limit = maximum`

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv

# Check valid interval limits; returns interval without modifications
interval = [0, 10]
res = pyhrv.utils.check_interval(interval)

# Check invalid interval limits; returns corrected interval limits
interval = [10, 0]
res = pyhrv.utils.check_interval(interval)
# here: res = [0, 10]
```

You can specify valid minimum and maximum values for the interval limits. If an interval with limits outside the valid region are provided, the limits will be set to the specified valid minimum and maximum values:

```
# Specify minimum and maximum valid values (here: [2, 8]); interval is out of valid_
↪ interval
interval = [0, 10]
limits = [2, 8]
```

(continues on next page)

(continued from previous page)

```
res = pyhrv.utils.check_interval(interval, limits)
# here: res = [2, 8]
```

You can specify default values for this function. These can be used if no interval is specified by the user and default values should apply (e.g. when integrating this function in custom functions with dynamic intervals).

```
# Don't specify intervals or limits, but set a default values (here: [0, 10])
res = pyhrv.utils.check_interval(interval=None, limits=None, default=[0, 10])
```

6.7.6 Segmentation: segmentation()

`pyhrv.utils.segmentation` (*nn=None, rpeaks=None, overlap=False, duration=300*)

Function Description

Segmentation of NNI series into individual segments of specified duration (e.g. splitting R-peak locations into 5 minute segments for computation of the SDNN index).

Note: The segmentation of the NNI series can only be conducted if the sum of the NNI series (i.e. the maximum duration) is greater than the specified segment duration (`segment`).

See also:

Application Notes below for more information.

Input Parameters

- `nni` (array): NN interval series in [ms] or [s]
- `full` (bool, optional): If True, returns last segment, even if the last segment is significantly shorter than the specified duration (default: True)
- `duration` (int, optional): Segment duration in [s] (default: 300s)
- `warn` (bool, optional): If True, raise a warning message if a segmentation could not be conducted (duration > NNI series duration)

Returns

- `segments` (array of arrays): Array with the segmented NNI series.
- `control` (bool): If True, segmentation was possible.

See also:

Application Notes below for more information about the returned segmentation results.

Raises

- `TypeError`: If `nni` input data is not specified

Application Notes

The function returns the results in an array of arrays if a segmentation of the signal was possible. This requires the sum of the provided NNI series (i.e. the maximum duration) to be greater than the specified segment duration (`segment`). In this case, a segmentation can be conducted and the segments with the respective NNIs will be returned along with the control variable set to `True`.

If a segmentation cannot be conducted, i.e. the maximum duration of the NNI series is shorter than the specified segment duration, the input unmodified NNI series will be returned along with the control variable set to `False`.

You can use the control variable to test whether the segmentation could be conducted successfully or not.

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv

# Load Sample NNI series (~5min)
nni = pyhrv.utils.load_sample_nni()

# Segment NNI series with a segment duration of [60s]
segments, control = pyhrv.utils.segmentation(nn=nni, duration=60)
```

This will return 5 segments and the control variable will be True. Use the code below to see the exact results:

```
# Print control variable
print("Segmentation?", control)

# Print segments
for i, segment in enumerate(segments):
    print("Segment %i" % i)
    print(segment)
```

6.7.7 Join Tuples: `join_tuples()`

`pyhrv.utils.join_tuples(*args)`

Function Description

Joins multiple `biosppy.utils.ReturnTuple` objects into one `biosppy.utils.ReturnTuple` object.

See also:

The `biosppy.utils.ReturnTuple` Object

Input Parameters

- `*args` (`biosppy.utils.ReturnTuple`): Multiple `biosppy.utils.ReturnTuple` objects (can also be stored in an array)

Returns

- `output` (`ReturnTuple` object): `biosppy.utils.ReturnTuple` object with the content of all input tuples/objects merged together.

Raises

- `TypeError`: If no input data is provided
- `TypeError`: If input data contains non-`biosppy.utils.ReturnTuple` objects

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv

# Join multiple ReturnTuple objects
tuples = pyhrv.utils.join_tuples(return_tuple1, return_tuple2, return_tuple3)
```

6.7.8 Standard Deviation: std()

`pyhrv.utils.std(array=None, dof=1)`

Function Description

Computes the standard deviation of a data series.

Input Parameters

- `array (array)`: Data series
- `dof (int, optional)`: Degrees of freedom (default: 1)

Returns

- `result (float)`: Standard deviation of the input data series

Raises

- `TypeError`: If no input array is provided

Computation

The standard deviation is computed according to the following formula:

$$SD = \sqrt{\frac{1}{n - dof} \sum_{i=1}^n (NNI_j - \overline{NNI})^2}$$

with:

- SD : Standard Deviation
- n : Number of NN Intervals
- dof : Degrees of Freedom
- NNI_j : NN Interval j
- \overline{NNI} : Mean NN Interval

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv

# Sample array
data = [600, 650, 800, 550, 900, 1000, 750]

# Compute standard deviation
sd = pyhrv.utils.std(data)
# sd = 163.2993161855452
```

6.7.9 Time Vector: time_vector()

`pyhrv.utils.time_vector(signal=None, sampling_rate=1000.)`

Function Description

Computes time vector based on the sampling rate of the provided input signal.

Input Parameters

- `signal` (array): ECG signal (or any other sensor signal)
- `sampling_rate` (int, float, optional): Sampling rate of the input signal in [Hz] (default: 1000Hz)

Returns

- `time_vector` (array): Time vector for the input signal sampled at the input `sampling_rate`

Raises

- `TypeError`: If no input array is provided

Example

The following example code demonstrates how to use this function:

```
# Import packages
import pyhrv
from opensignalsreader import OpenSignalsReader

# Load sample ECG signal stored in an OpenSignals file
acq = OpenSignalsReader('./samples/SampleECG.npy')
signal = acq.signal('ECG')
sampling_rate = acq.sampling_rate

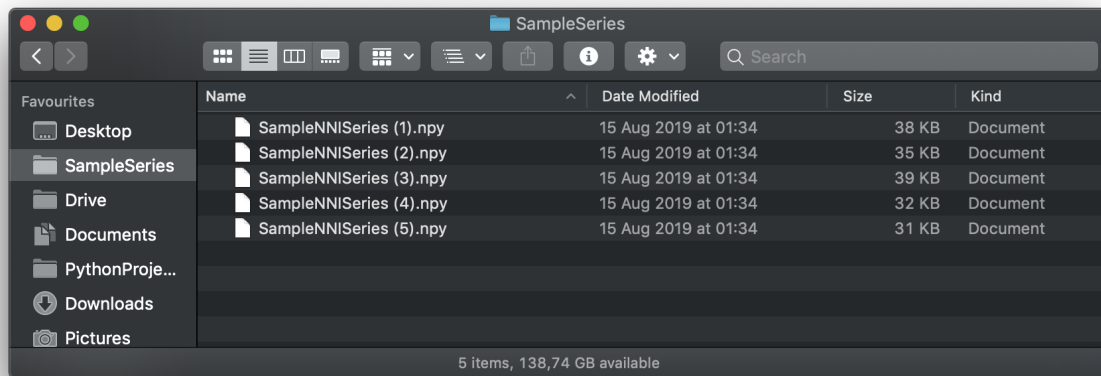
# Compute time vector
t = pyhrv.utils.time_vector(signal, sampling_rate)
```

7.1 Bulk Processing of Multiple NNI Series with pyHRV

This tutorial aims to guide you through all the steps needed to bulk process multiple NNI series using a simple Python script. This can be helpful in one of the following examples (there exist of course many other cases for which this might be useful:

- You need to analyse datasets from multiple participants joining your experiment
- Over time, multiple new participants join your experiment and you need to compute their datasets
- The analysis process requires you to test different input settings, such as for example adjusting the selected frequency bands or filter orders, which requires you to recompute the HRV parameters with each change being made to the input
- Any other reason that you might think of needing bulk processing...

For instance, I have a couple of files with NNI series stored in the `.npy` [numpy array format](#) which I would like to process using the same input settings.



7.1.1 Step 1: Loading the Necessary Packages

We will require the following 3 packages in this script:

- `glob` to loop through all the files containing the NNI series in a folder
- `numpy` to load the NNI series
- `pyhrv` to compute the HRV results

```
import glob
import pyhrv
import numpy as np
```

7.1.2 Step 2: Defining the Input Parameters for the HRV Functions

pyHRV's functions already come with default values for input parameters. However, the use of custom input values is often interesting in order to adjust the functions to the experimental conditions.

In this step, we will prepare a series of dictionaries containing the input parameters for the computation of Time Domain, Frequency Domain and Nonlinear parameters. These dictionaries will afterwards be used with the `pyhrv.hrv()` (see *The HRV Function: `hrv()`*).

First, we prepare the inputs for the Time Domain functions:

```
# Time Domain Settings
settings_time = {
    'threshold': 50,          # Computation of NNXX/pNNXX with 50 ms threshold ->
    # NN50 & pNN50
    'plot': True,            # If True, plots NNI histogram
    'binsize': 7.8125        # Binsize of the NNI histogram
}
```

For the Frequency Domain parameters, we will prepare individual input dictionaries for each of the available functions, *Welch's Method: `welch_psd()`*, *Lomb-Scargle Periodogram: `lomb_psd()`* and *Autoregressive Method: `ar_psd()`*.

```
# Frequency Domain Settings
settings_welch = {
    'nfft': 2 ** 12,          # Number of points computed for the FFT result
    'detrend': True,         # If True, detrend NNI series by subtracting the mean
    ↪ NNI
    'window': 'hanning'      # Window function used for PSD estimation
}

settings_lomb = {
    'nfft': 2**8,            # Number of points computed for the Lomb PSD
    'ma_size': 5             # Moving average window size
}

settings_ar = {
    'nfft': 2**12,           # Number of points computed for the AR PSD
    'order': 32              # AR order
}
```

At last, we will set the input parameters for the Nonlinear Parameters.

```
# Nonlinear Parameter Settings
settings_nonlinear = {
    'short': [4, 16],        # Interval limits of the short term fluctuations
    'long': [17, 64],       # Interval limits of the long term fluctuations
    'dim': 2,                # Sample entropy embedding dimension
    'tolerance': None        # Tolerance distance for which the vectors to be
    ↪ considered equal (None sets default values)
}
```

Tip: Storing the settings in multiple dictionaries within a Python script might become unhandy if you would like to share your settings or simply keep those out of a Python script for more versatility.

In those cases, store the settings in a JSON file which can be easily read with Python's native **JSON Package**.

7.1.3 Step 3: Looping Through All the Available Files

In this step, we will create a loop which to go through all the available files and load the NNI series from each file which we then use to compute the HRV parameters.

For this, we will first define the path where the files are stored. Afterwards, we will loop through all the files using the glob package.

```
# Path where the NNI series are stored
nni_data_path = './SampleSeries/'

# Go through all files in the folder (here: files that end with .npy only)
for nni_file in glob.glob(nni_data_path + '*.npy'):

    # Load the NNI series of the current file
    nni = np.load(nni_file)
```

7.1.4 Step 4: Computing the HRV parameters

At last, we will pass the previously defined input parameters to the `pyhrv.hrv()` function and compute the HRV parameters.

```
# Path where the NNI series are stored
nni_data_path = './SampleSeries/'

# Go through all files in the folder (here: that end with .npy)
for nni_file in glob.glob(nni_data_path + '*.npy'):

    # Load the NNI series of the current file
    nni = np.load(nni_file)

    # Compute the pyHRV parameters
    results = pyhrv.hrv(nni=nni,
                        kwargs_time=settings_time,
                        kwargs_welch=settings_welch,
                        kwargs_ar=settings_ar,
                        kwargs_lomb=settings_lomb,
                        kwargs_nonlinear=settings_nonlinear)
```

That's it! Now adjust the script to your needs so that the computed results can be used as you need those for your project.

7.1.5 Tl;dr - The Entire Script

The code sections we have generated over the course of this tutorial are summarized in the following Python script:

```
# Import necessary packages
import glob
import pyhrv
import numpy as np

# Define HRV input parameters
# Time Domain Settings
settings_time = {
    'threshold': 50,          # Computation of NNXX/pNNXX with 50 ms threshold ->
    'NN50 & pNN50',          # If True, plots NNI histogram
    'plot': True,            # Binsize of the NNI histogram
    'binsize': 7.8125
}

# Frequency Domain Settings
settings_welch = {
    'nfft': 2 ** 12,         # Number of points computed for the FFT result
    'detrend': True,         # If True, detrend NNI series by subtracting the mean
    'NNI',                   # Window function used for PSD estimation
    'window': 'hanning'
}

settings_lomb = {
    'nfft': 2**8,            # Number of points computed for the Lomb PSD
    'ma_size': 5             # Moving average window size
}
```

(continues on next page)

(continued from previous page)

```
settings_ar = {
    'nfft': 2**12,          # Number of points computed for the AR PSD
    'order': 32             # AR order
}

# Nonlinear Parameter Settings
settings_nonlinear = {
    'short': [4, 16],       # Interval limits of the short term fluctuations
    'long': [17, 64],      # Interval limits of the long term fluctuations
    'dim': 2,               # Sample entropy embedding dimension
    'tolerance': None       # Tolerance distance for which the vectors to be
    ↪ considered equal (None sets default values)
}

# Path where the NNI series are stored
nni_data_path = './SampleSeries/'

# Go through all files in the folder (here: that end with .npy)
for nni_file in glob.glob(nni_data_path + '*.npy'):

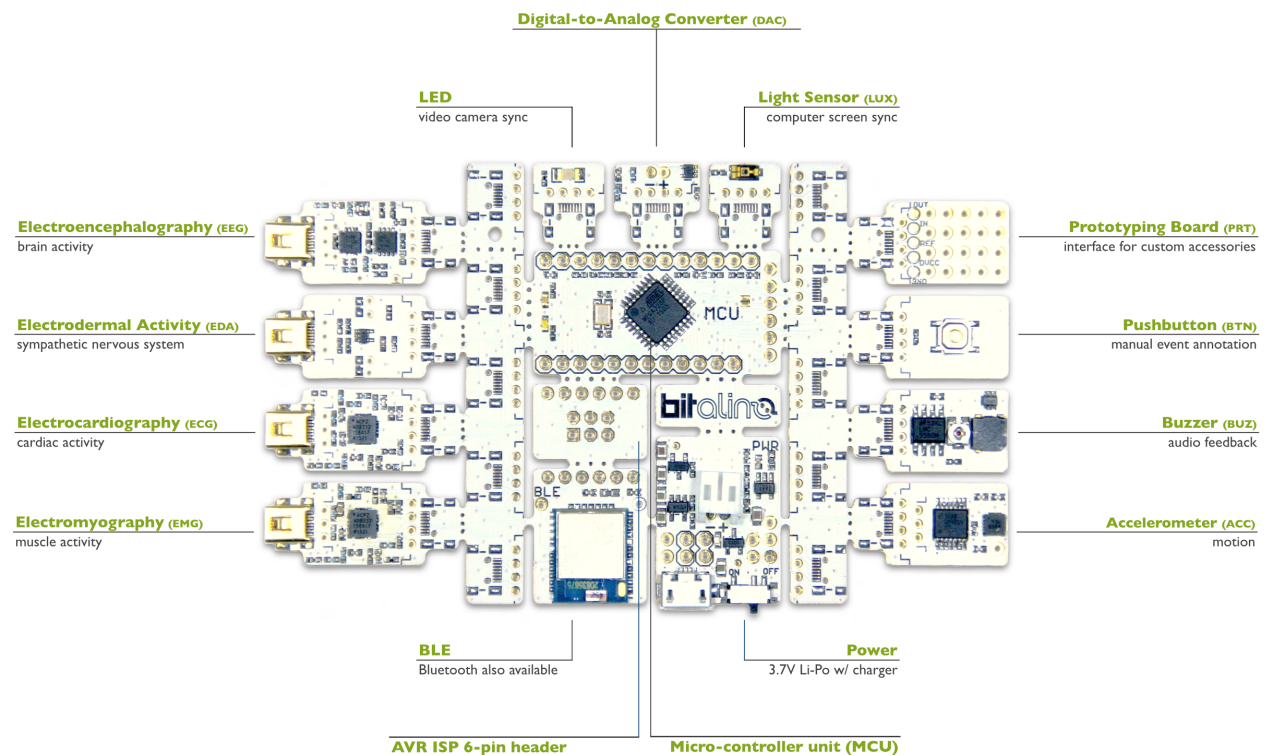
    # Load the NNI series of the current file
    nni = np.load(nni_file)

    # Compute the pyHRV parameters
    results = pyhrv.hrv(nni=nni,
                        kwargs_time=settings_time,
                        kwargs_welch=settings_welch,
                        kwargs_ar=settings_ar,
                        kwargs_lomb=settings_lomb,
                        kwargs_nonlinear=settings_nonlinear)
```

Note: Any feedback or ideas how to improve this tutorial? Feel free to share your ideas or questions with me via e-mail: pgomes92@gmail.com

7.2 ECG Acquisition & HRV Analysis with BITalino & pyHRV

This tutorial aims to guide you through all the steps from recording your own ECG signals up to computing all HRV parameters using pyHRV and saving them in your own, first HRV report.



The ECG signals will be acquired using a BITalino (r) evolution Board and the OpenSignals (r)evolution software. Additionally, this tutorial uses the BioSPPy toolkit to filter your ECG signal and to extract the R-peak locations. Finally, we'll use the pyHRV package to compute all available HRV parameters from your ECG signal(s) and generate your first HRV report.

Note: This tutorial demonstrates how to acquire ECG signals using the BITalino toolkit and the OpenSignals software. It is of course not limited to these tools. You can, of course, use any other ECG signal independent from the signal acquisition hardware and software.

Note: The images of OpenSignals, BITalino and electrode placement have been provided with the friendly support of PLUX wireless biosignals S.A..

7.2.1 Step 1: Getting Started

Acquiring ECG signals requires both hardware and software. With your BITalino device in your hands you have already mastered the first step of this tutorial - hardware: check!

Download & Install OpenSignals (r)evolution

The easiest way to manage your signal acquisitions using BITalino is by using the OpenSignals (r)evolution software. Download and install the OpenSignals (r)evolution software.

Note: OpenSignals (r)evolution uses Google Chrome as rendering engine for its graphical user interface. You must

install the [Google Chrome Browser](#) before installing OpenSignals.

Before heading directly to the signal acquisition, we'll install all the Python packages that we'll be using in this tutorial first.

Download & Install the Packages Used in this Tutorial

OpenSignals (r)evolution can store the raw ECG data in .TXT files which we will be importing into our Python script or project using the [opensignalsreader](#) package. You can install this package using the following command in your terminal (macOS and Linux) or command prompt (Windows).

```
pip install opensignalsreader
```

Next, we'll install the [BioSPPy toolkit](#) which we'll use to filter the acquired ECG signals and to extract the R-peak data. You can install this package using the following command in your your command prompt.

```
pip install biosppy
```

For the sake of completeness - in case you have not installed it yet - download and install pyHRV package by using the following command in your terminal or command prompt:

```
pip install pyhrv
```

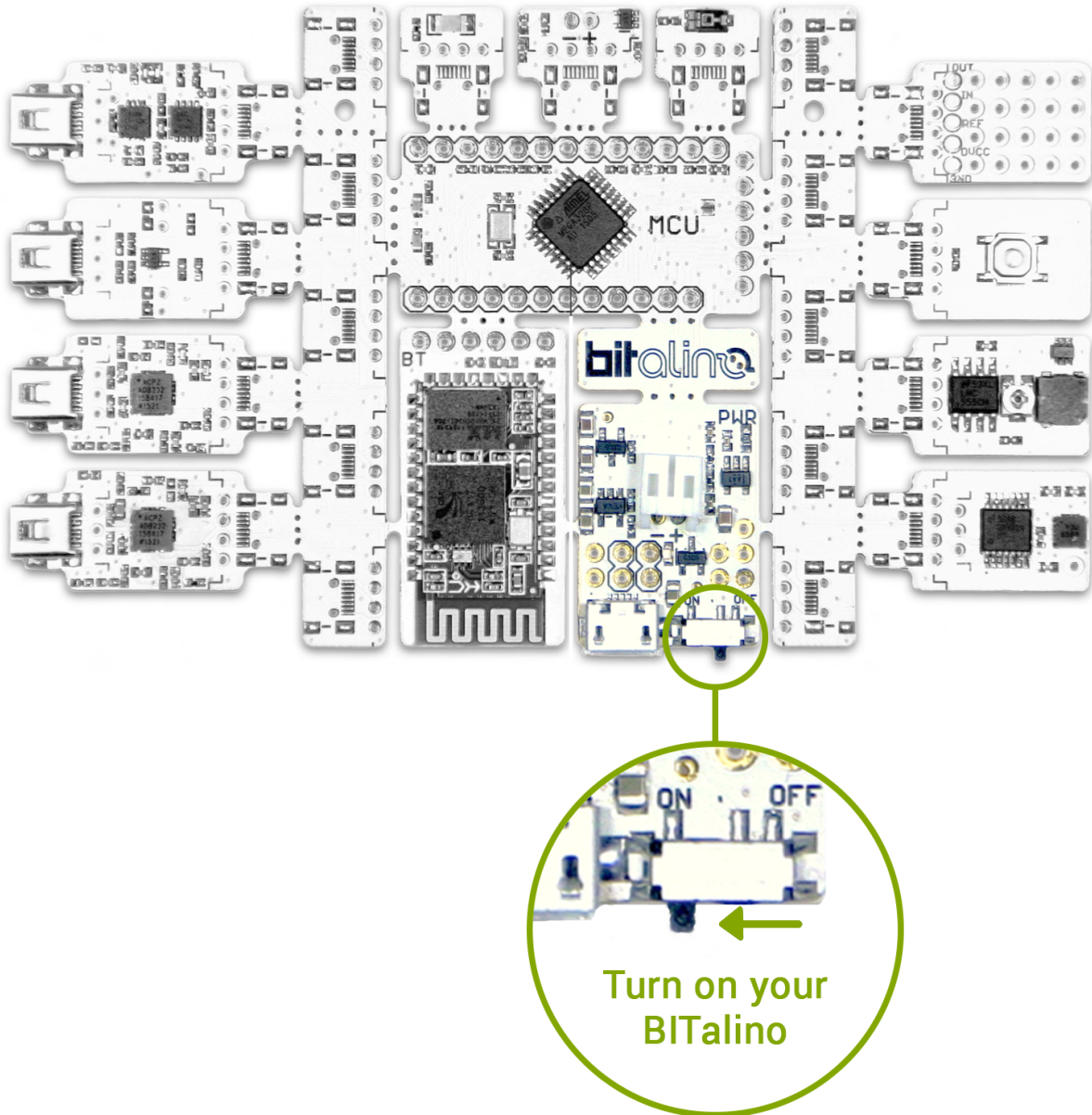
Note: In some cases, pyhrv cannot be installed due to outdated setuptools or (yet) officially supported Python versions. In this case, install the dependencies first before installing pyhrv:

```
pip install biosppy
pip install matplotlib
pip install numpy
pip install scipy
pip install nolds
pip install spectrum
pip install pyhrv
```

7.2.2 Step 2: Setting Up the Acquisition

BITalino transmits the acquired sensor signals via Bluetooth to your computer, where the signals can be recorded and visualized in real-time using the OpenSignals software.

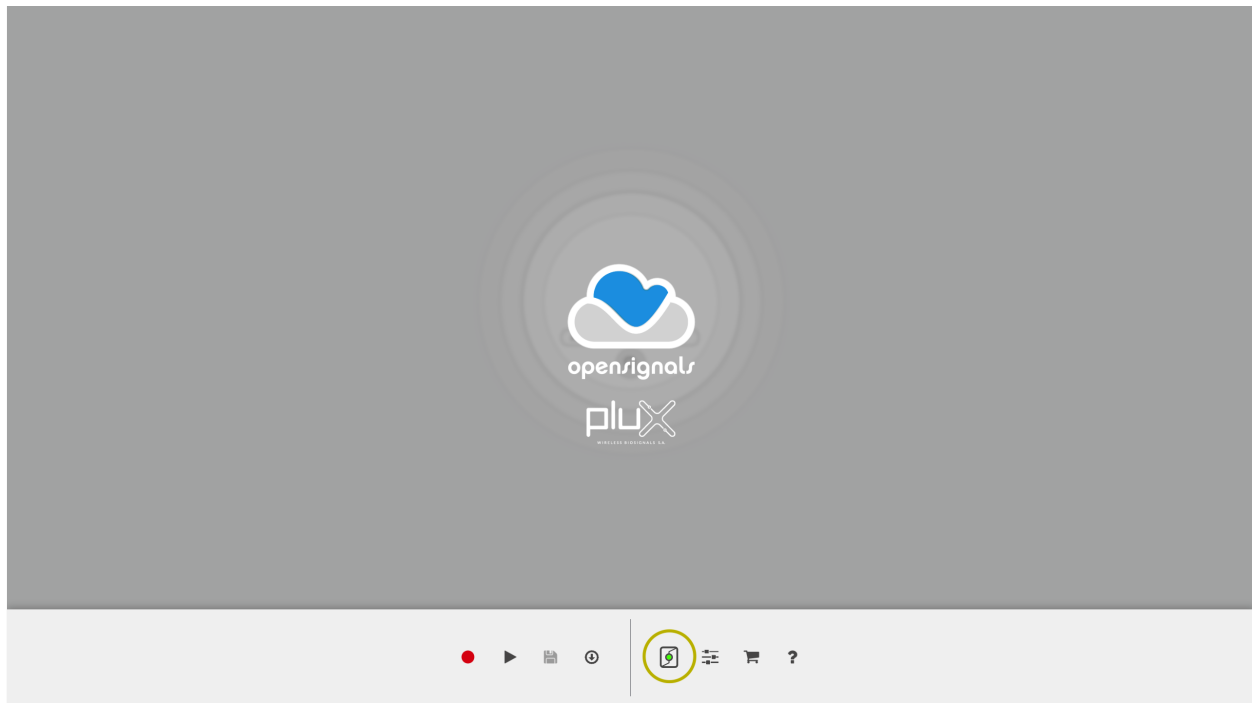
Turn on your BITalino by turning the switch of the power module into the position shown in the image below:



The Bluetooth connection between your BITalino device and your operating system must be established before using the OpenSignals software. Use your operating system's Bluetooth manager in order to establish the connection.

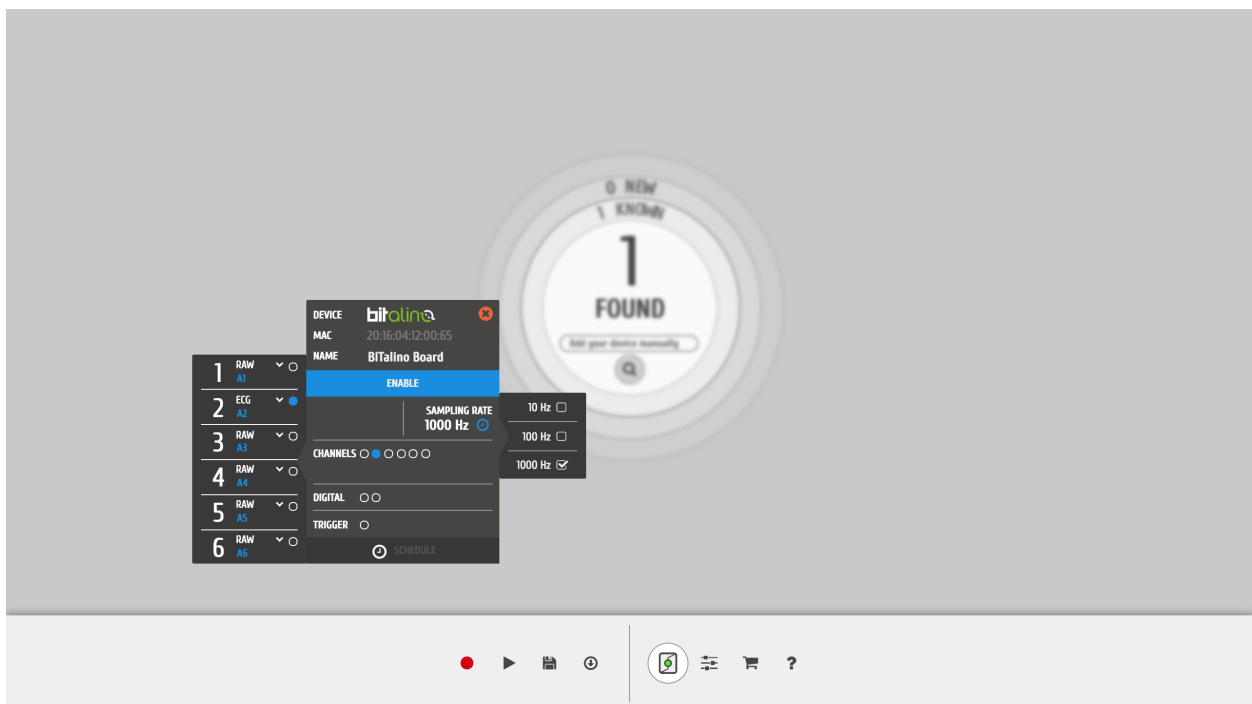
Note: You will be prompted to insert a pairing code to establish the connection, which for BITalino devices is *1234*.

After establishing the Bluetooth connection, open the OpenSignals software and click on the device manager icon highlighted in the screenshot below (green circle) where you should find your BITalino now.



Click on the BITalino panel, select *ECG* from the dropdown menu of channel 2 and click on the circle on the left (must turn blue) to activate this channel for acquisition. Finally, click on the *ENABLE* button (must turn blue) to select your BITalino as acquisition device.

Your device panel should now look like the device panel seen in the screenshot below (you can ignore the configuration of the remaining channels).



Note: Click on the magnifying glass icon in the center of the device manager to search for your device, if your

BITalino is not listed shown yet.

7.2.3 Step 3: Connecting the Electrodes and Sensor

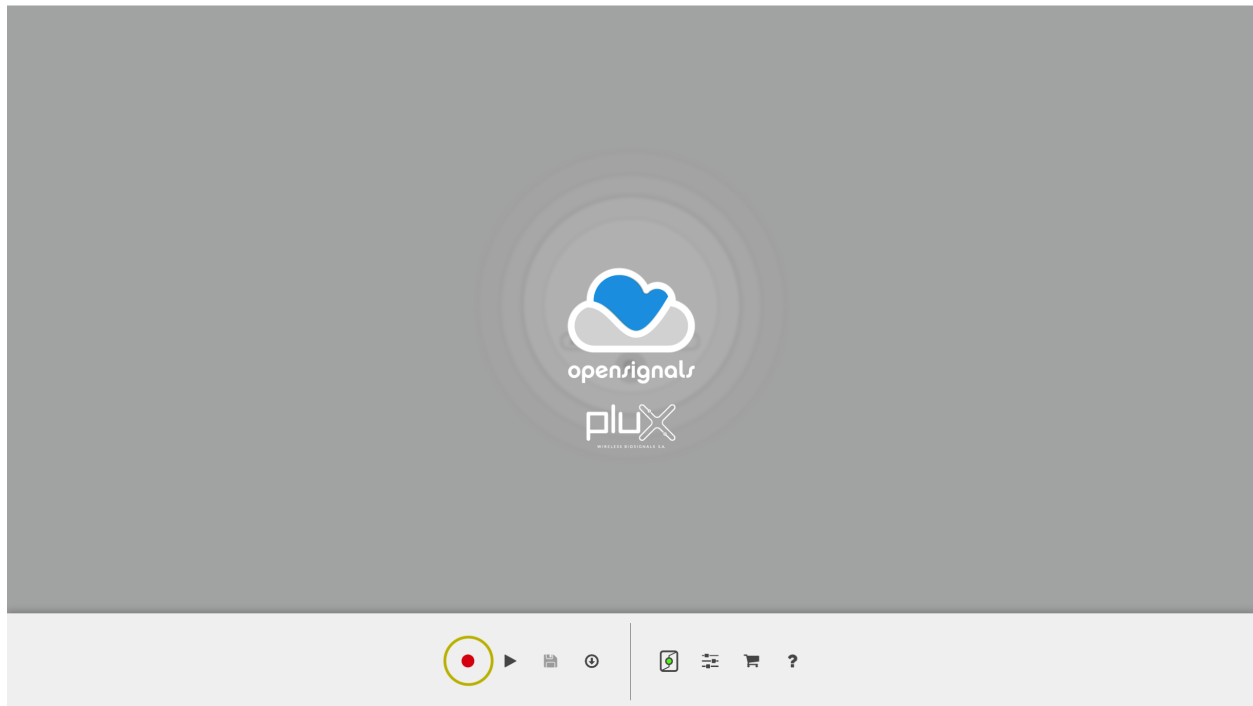
The BITalino ECG sensor is designed for single-lead ECG acquisitions according to the Einthoven leads. Visit the following forum thread of the BITalino forum to learn how place your electrodes:

<http://forum.bitalino.com/viewtopic.php?t=135>

Connect the 2 or 3 lead electrode cable to your electrodes and connect it with the ECG sensor of your BITalino board.

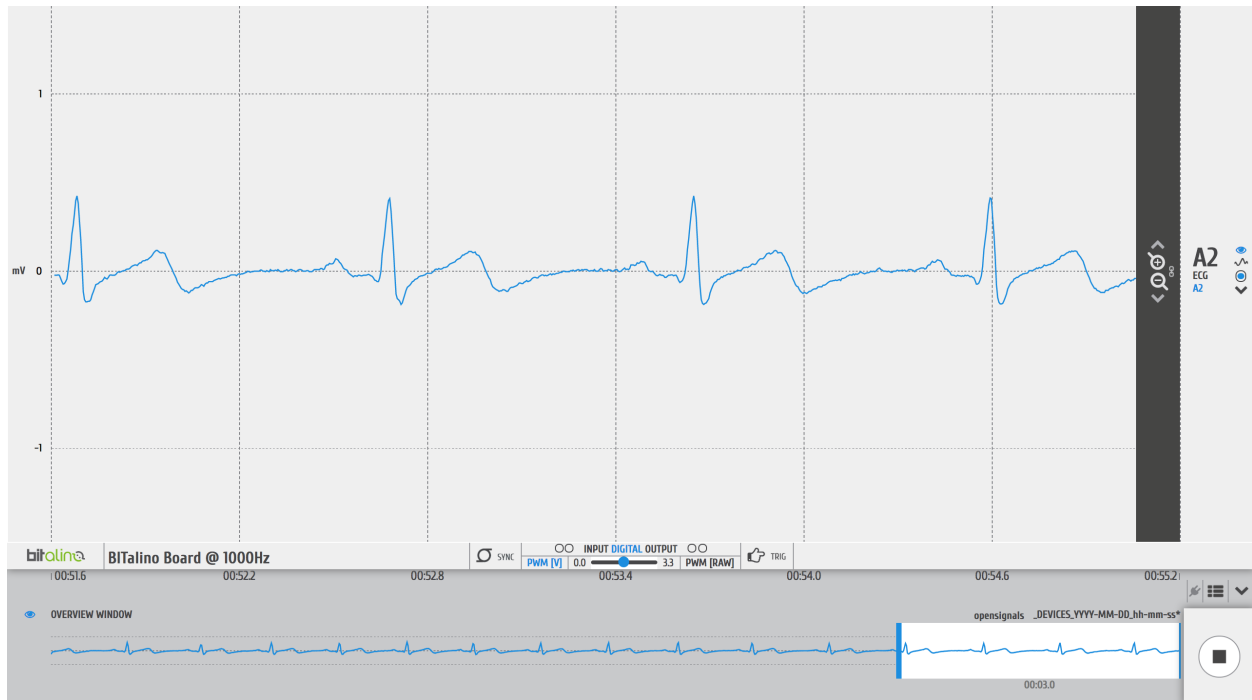
7.2.4 Step 4: Acquiring ECG Signals

After successfully configuring your BITalino in the OpenSignals software, we can now acquire ECG signals. For this, click on the record icon in the OpenSignals menu highlighted in the screenshot below (green circle).



Now, the connection between the software and your BITalino will be established, and the acquisition will start shortly after. The status LED of your BITalino will switch from fading into blinking when it enters the acquisition mode.

In the OpenSignals software, the ECG signal should be visualized as shown in the screenshot below:



7.2.5 Step 5: Loading OpenSignals Sensor Data

In this step, we will import the ECG signal acquired in the previous step using Python. If you haven't done yet, create and open a new Python script in the IDE of your preference, and use the following code to import the ECG signal. Note, that we will also already import all the necessary packages for the upcoming steps.

```
# Import packages
from pyhrv.hrv import hrv
from opensignalsreader import OpenSignalsReader
from biosppy.signals.ecg import ecg

# Specify the file path of your OpenSignals file (absolute file path is recommended)
fpath = '/path/to/SampleECG.txt'

# Load the acquisition file
acq = OpenSignalsReader(fpath)

# Get the ECG signal
signal = acq.signal('ECG')
```

That's it! Now that we have the acquired ECG signal stored in the `signal` variable, we can move on to the next step.

7.2.6 Step 6: Processing ECG Signal and Extracting R-Peaks

'BioSPPy is an open-source biosignal processing toolkit<<https://github.com/PIA-Group/BioSPPy>>' which we will be using to filter our ECG signal (e.g. removing motion artifacts), and to extract the R-peak locations which are needed for the HRV computation. We can do this using the `biosppy.signals.ecg.ecg()` function.

This function returns a series of datasets and parameters, however, we are only interested in the filtered ECG signal. Add the following line of code to the existing code of the previous step:

```
# Filter ECG signal
filtered_signal = ecg(signal)[1]
```

Note: You can also skip this step if you want to use any of the functions below as this step is already integrated in these functions:

- `pyhrv.hrv()`
- `pyhrv.time_domain.time_domain()`
- `pyhrv.frequency_domain.frequency_domain()`
- `pyhrv.nonlinear.nonlinear()`

In these cases, simply pass the ECG signal to the functions as follows:

```
pyhrv.hrv(signal=signal)
pyhrv.time_domain.time_domain(signal=signal)
pyhrv.frequency_domain.frequency_domain(signal=signal)
pyhrv.nonlinear.nonlinear(signal=signal)
```

For all the other functions, pass the R-peak locations or the NNI series to the functions.

7.2.7 Step 7: Compute HRV Parameters

In this final step, we will use the `pyhrv.hrv.hrv()` function to compute all the HRV parameters of this toolbox and have a short look on how to compute individual parameters or methods from the different domains.

To compute all available HRV parameters with the default parameters, add the following line to your code:

```
# Compute all HRV parameters with default input parameters
results = hrv(signal=filtered_signal)
```

Note: Set the `show` input parameter of the `pyhrv.hrv.hrv()` function to `True` if you want to display all the generated plots.

```
# Compute all HRV parameters with default input parameters and show all plot figures
results = pyhrv.hrv(signal=signal, show=True)
```

Important: You might have to close all generated plot figures to allow the execution of the upcoming code sections. Alternatively, turn on the interactive mode of the matplotlib package to prevent this issue.

See also:

https://matplotlib.org/faq/usage_faq.html#what-is-interactive-mode

You can now print the results and see all the computed parameters using:

```
print(results)
```

However, if you want list the parameters in a more reader-friendly format, it is better to loop through all the available keys and parameters and print them one at a time using:


```
# Print all the parameters keys and values individually
for key in results.keys():
    print(key, results[key])
```

That's it! We have successfully recorded an ECG signal, processed it and computed the HRV parameters with only a few lines of code.

7.2.8 Tl;dr - The Entire Script

The code sections we have generated over the course of this tutorial are summarized in the following Python script:

```
# Import packages
import pyhrv.tools as tools
from pyhrv.hrv import hr
from opensignalsreader import OpenSignalsReader
from biosppy.signals.ecg import ecg

# Specify the file path of your OpenSignals file (absolute file path is recommended)
fpath = '/path/to/SampleECG.txt'

# Load the acquisition file
acq = OpenSignalsReader(fpath)

# Get the ECG signal
signal = acq.signal('ECG')

# Filter ECG signal and extract the R-peak locations
filtered_signal = ecg(signal)[1]

# Compute all HRV parameters with default input parameters
results = hrv(signal=filtered_signal)

# Print all the parameters keys and values individually
for key in results.keys():
    print(key, results[key])

# Create HRV report in .TXT format
hrv_report(results, path='/my/favorite/path', rfile='MyFirstHRVReport')
```

Note: Any feedback or ideas how to improve this tutorial? Feel free to share your ideas or questions with me via e-mail: pgomes92@gmail.com

8.1 License

BSD 3-Clause License

Copyright (c) 2018, Pedro Gomes All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

8.2 Disclaimer

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Highlights

This Python package computes...

- ... fundamental HRV data series (NNI, NNI, HR)
- ... HRV Time Domain parameters
- ... HRV Frequency Domain parameters
- ... nonlinear HRV parameters

... and comes with variety of additional HRV tools, such as...

- ... ECG and Tachogram plotting features
- ... export and import of HRV results to .JSON files
- ... HRV report generation in .TXT, .CSV and .PDF formats
- ... and many other useful features to support your HRV research!

CHAPTER 10

Installation

This package can be installed using the *pip* tool:

```
$ pip install pyhrv
```

Disclaimer & Context

This program is distributed in the hope it will be useful and provided to you “as is”, but WITHOUT ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. This program is NOT intended for medical diagnosis. We expressly disclaim any liability whatsoever for any direct, indirect, consequential, incidental or special damages, including, without limitation, lost revenues, lost profits, losses resulting from business interruption or loss of data, regardless of the form of action or legal theory under which the liability may be asserted, even if advised of the possibility of such damages.

This package has initially (up to version 0.3) been developed within the scope of my master thesis “Development of an Open-Source Python Toolbox for Heart Rate Variability (HRV)” at the University of Applied Sciences Hamburg, Germany (Faculty Life Sciences, Department of Biomedical Engineering) and PLUX wireless biosignals, S.A., Lisbon, Portugal.

P

- `pyhrv.frequency_domain.ar_psd()` (*built-in function*), 62
- `pyhrv.frequency_domain.frequency_domain()` (*built-in function*), 81
- `pyhrv.frequency_domain.lomb_psd()` (*built-in function*), 58
- `pyhrv.frequency_domain.psd_comparison()` (*built-in function*), 68, 75
- `pyhrv.frequency_domain.welch_psd()` (*built-in function*), 52
- `pyhrv.hrv.hrv()` (*built-in function*), 14
- `pyhrv.nonlinear.dfa()` (*built-in function*), 97
- `pyhrv.nonlinear.frequency_domain()` (*built-in function*), 100
- `pyhrv.nonlinear.poincare()` (*built-in function*), 92
- `pyhrv.nonlinear.sampen()` (*built-in function*), 95
- `pyhrv.time_domain.geometrical_parameters()` (*built-in function*), 46
- `pyhrv.time_domain.hr_parameters()` (*built-in function*), 23
- `pyhrv.time_domain.nn20()` (*built-in function*), 38
- `pyhrv.time_domain.nn50()` (*built-in function*), 37
- `pyhrv.time_domain.nni_differences_parameters()` (*built-in function*), 22
- `pyhrv.time_domain.nni_parameters()` (*built-in function*), 21
- `pyhrv.time_domain.nnXX()` (*built-in function*), 35
- `pyhrv.time_domain.rmssd()` (*built-in function*), 32
- `pyhrv.time_domain.sdann()` (*built-in function*), 29
- `pyhrv.time_domain.sdnns()` (*built-in function*), 25
- `pyhrv.time_domain.sdnns_index()` (*built-in function*), 26
- `pyhrv.time_domain.sdsd()` (*built-in function*), 33
- `pyhrv.time_domain.time_domain()` (*built-in function*), 48
- `pyhrv.time_domain.tinn()` (*built-in function*), 40
- `pyhrv.time_domain.triangular_index()` (*built-in function*), 44
- `pyhrv.tools.heart_rate()` (*built-in function*), 106
- `pyhrv.tools.hr_heatplot()` (*built-in function*), 113
- `pyhrv.tools.hrv_export()` (*built-in function*), 118
- `pyhrv.tools.hrv_import()` (*built-in function*), 120
- `pyhrv.tools.hrv_report()` (*built-in function*), 117
- `pyhrv.tools.nn_diff()` (*built-in function*), 105
- `pyhrv.tools.nn_intervals()` (*built-in function*), 105
- `pyhrv.tools.plot_ecg()` (*built-in function*), 108
- `pyhrv.tools.radar_chart()` (*built-in function*), 121
- `pyhrv.tools.tachogram()` (*built-in function*), 109
- `pyhrv.utils.check_input()` (*built-in function*), 126
- `pyhrv.utils.check_interval()` (*built-in function*), 127
- `pyhrv.utils.join_tuples()` (*built-in function*), 130
- `pyhrv.utils.load_hrv_keys_json()` (*built-in function*), 125
- `pyhrv.utils.load_sample_nni()` (*built-in function*), 125
- `pyhrv.utils.nn_format()` (*built-in function*), 126

`pyhrv.utils.segmentation()` (*built-in function*), [129](#)
`pyhrv.utils.std()` (*built-in function*), [131](#)
`pyhrv.utils.time_vector()` (*built-in function*),
[131](#)