
NeuroKit2

Release 0.0.39

Official Documentation

Jun 28, 2020

CONTENTS

1	Introduction	3
1.1	Quick Example	3
1.2	Installation	4
1.3	Contributing	4
1.4	Documentation	4
1.5	Citation	6
1.6	Physiological Data Preprocessing	6
1.7	Physiological Data Analysis	13
1.8	Miscellaneous	15
1.9	Popularity	21
1.10	Notes	22
2	Authors	23
2.1	Core team	23
2.2	Contributors	23
3	Installation	25
3.1	1. Python	25
3.2	2. NeuroKit	26
4	Get Started	27
4.1	Get familiar with Python in 10 minutes	27
4.2	Where to start	35
5	Examples	37
5.1	Try the examples in your browser	37
5.2	1. Analysis Paradigm	37
5.3	2. Biosignal Processing	38
5.4	3. Heart rate and heart cycles	38
5.5	4. Electrodermal activity	39
5.6	5. Respiration rate and respiration cycles	39
5.7	6. Muscle activity	39
5.8	Simulate Artificial Physiological Signals	40
5.9	Customize your Processing Pipeline	45
5.10	Event-related Analysis	50
5.11	Interval-related Analysis	58
5.12	Analyze Electrodermal Activity (EDA)	62
5.13	Analyze Respiratory Rate Variability (RRV)	64
5.14	ECG-Derived Respiration (EDR) Analysis	69
5.15	Extract and Visualize Individual Heartbeats	71

5.16	How to create epochs	85
5.17	Complexity Analysis of Physiological Signals	89
5.18	Analyze Electrooculography EOG data (eye blinks, saccades, etc.)	94
5.19	Fit a function to a signal	97
6	Resources	103
6.1	Recording good quality signals	103
6.2	What software for physiological signal processing	104
6.3	Additional Resources	106
7	Functions	107
7.1	ECG	107
7.2	PPG	124
7.3	HRV	128
7.4	RSP	135
7.5	EDA	146
7.6	EMG	157
7.7	EEG	163
7.8	Signal Processing	165
7.9	Events	187
7.10	Data	190
7.11	Epochs	191
7.12	Statistics	194
7.13	Complexity	202
7.14	Miscellaneous	234
8	Benchmarks	239
8.1	Benchmarking of ECG Preprocessing Methods	239
8.2	References	250
9	Datasets	251
9.1	ECG (<i>1000 hz</i>)	251
9.2	ECG - pandas (<i>3000 hz</i>)	251
9.3	Event-related (<i>4 events</i>)	251
9.4	Resting state (<i>5 min</i>)	252
9.5	Resting state (<i>8 min</i>)	252
10	Contributing	255
10.1	Understanding NeuroKit	256
10.2	Contributing guide	258
10.3	Ideas for first contributions	265
	Python Module Index	267
	Index	269

Welcome to **NeuroKit**'s documentation. Here you can find information and learn about Python, NeuroKit, Physiological Signals and more.

You can navigate to the different sections using the left panel. We would recommend checking out the **guides** and **examples**, where you can find tutorials and hands-on walkthroughs.

INTRODUCTION

NeuroKit2



The Python Toolbox for Neurophysiological Signal Processing

This package is the continuation of [NeuroKit 1](#). It's a user-friendly package providing easy access to advanced biosignal processing routines. Researchers and clinicians without extensive knowledge of programming or biomedical signal processing can **analyze physiological data with only two lines of code**.

1.1 Quick Example

```
import neurokit2 as nk

# Download example data
data = nk.data("bio_eventrelated_100hz")

# Preprocess the data (filter, find peaks, etc.)
processed_data, info = nk.bio_process(ecg=data["ECG"], rsp=data["RSP"], eda=data["EDA
→"], sampling_rate=100)

# Compute relevant features
results = nk.bio_analyze(processed_data, sampling_rate=100)
```

And **boom** your analysis is done

1.2 Installation

To install NeuroKit2, run this command in your terminal:

```
pip install neurokit2
```

If you're not sure how/what to do, be sure to read our [installation guide](#).

1.3 Contributing

NeuroKit2 is a collaborative project with a community of contributors with all levels of development expertise. Thus, if you have some ideas for **improvement**, **new features**, or just want to **learn Python** and do something useful at the same time, do not hesitate and check out the following guides:

- [Understanding NeuroKit](#)
- [Contributing guide](#)
- [Ideas for first contributions](#)

1.4 Documentation

Click on the links above and check out our tutorials:

1.4.1 General

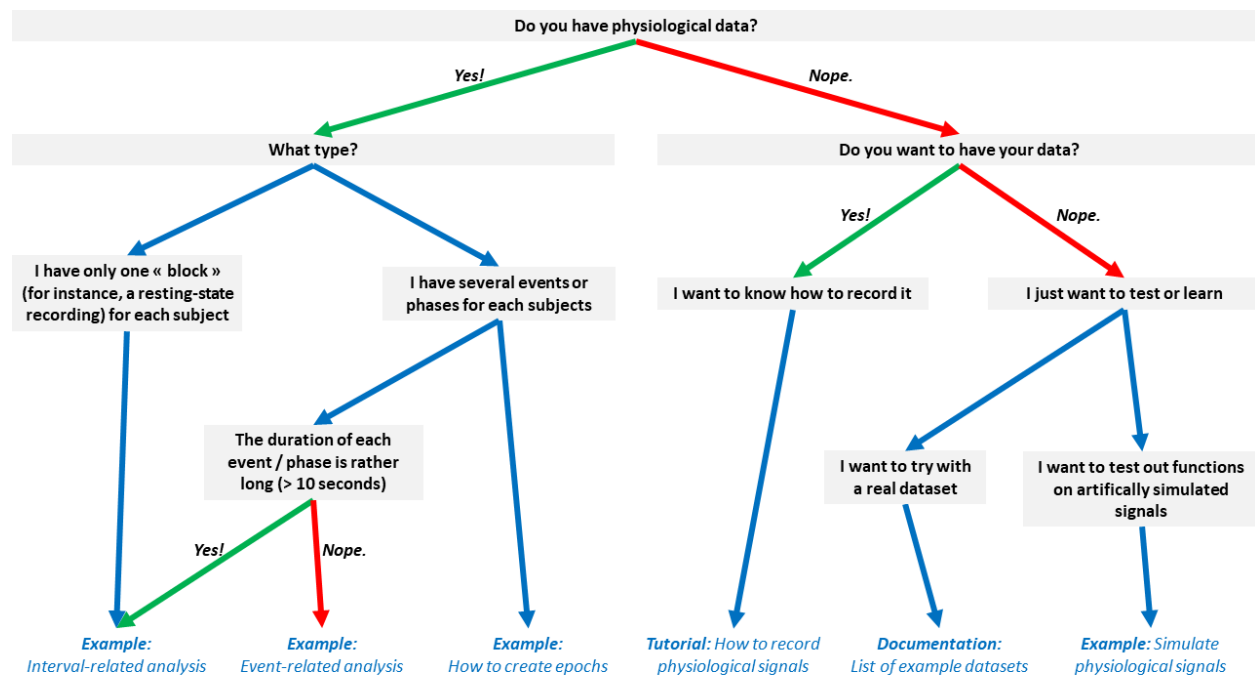
- [Get familiar with Python in 10 minutes](#)
- [Recording good quality signals](#)
- [What software for physiological signal processing](#)
- [Install Python and NeuroKit](#)
- [Included datasets](#)
- [Additional Resources](#)

1.4.2 Examples

- Simulate Artificial Physiological Signals
- Customize your Processing Pipeline
- Event-related Analysis
- Interval-related Analysis
- Analyze Electrodermal Activity (EDA)
- Analyze Respiratory Rate Variability (RRV)
- Extract and Visualize Individual Heartbeats
- Locate P, Q, S and T waves in ECG
- Complexity Analysis of Physiological Signals
- Analyze Electrooculography EOG data
- Fit a function to a signal

You can try out these examples directly in your browser.

Don't know which tutorial is suited for your case? Follow this flowchart:



1.5 Citation

```
nk.cite()
```

You can cite NeuroKit2 as follows:

```
- Makowski, D., Pham, T., Lau, Z. J., Brammer, J. C., Lesspinasse, F., Pham, H.,  
  Schölzel, C., & S H Chen, A. (2020). NeuroKit2: A Python Toolbox for  
  ↳Neurophysiological  
    Signal Processing. Retrieved March 28, 2020, from https://github.com/  
  ↳neuropsychology/NeuroKit
```

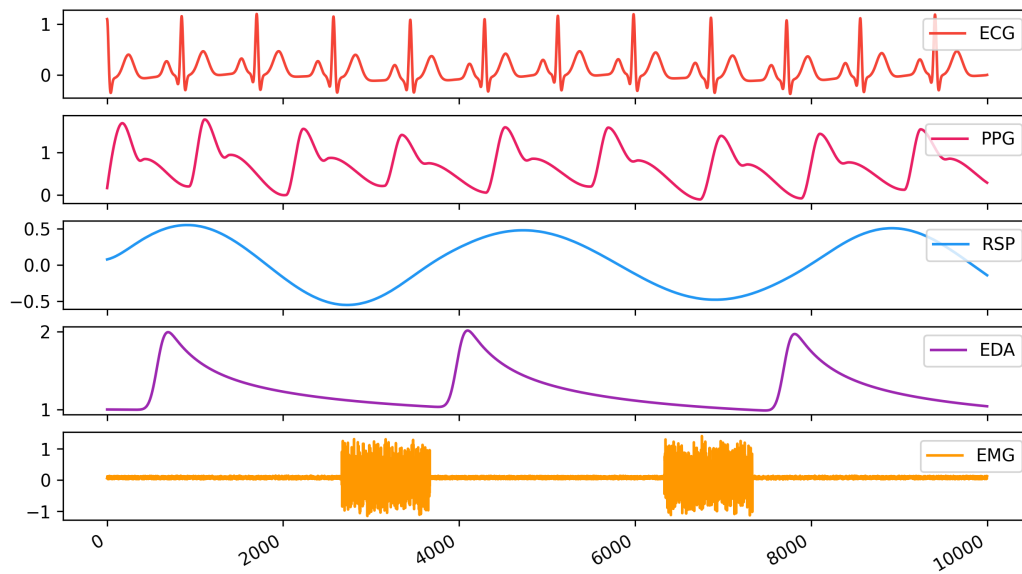
Full bibtex reference:

```
@misc{neurokit2,  
  doi = {10.5281/ZENODO.3597887},  
  url = {https://github.com/neuropsychology/NeuroKit},  
  author = {Makowski, Dominique and Pham, Tam and Lau, Zen J. and Brammer, Jan C. and  
  ↳Lespinasse, Fran↳c↳ois and Pham, Hung and Schölzel, Christopher and S H Chen,  
  ↳Annabel},  
  title = {NeuroKit2: A Python Toolbox for Neurophysiological Signal Processing},  
  publisher = {Zenodo},  
  year = {2020},  
}
```

1.6 Physiological Data Preprocessing

1.6.1 Simulate physiological signals

```
import numpy as np  
import pandas as pd  
import neurokit2 as nk  
  
# Generate synthetic signals  
ecg = nk.ecg_simulate(duration=10, heart_rate=70)  
ppg = nk.ppg_simulate(duration=10, heart_rate=70)  
rsp = nk.rsp_simulate(duration=10, respiratory_rate=15)  
eda = nk.eda_simulate(duration=10, scr_number=3)  
emg = nk.emg_simulate(duration=10, burst_number=2)  
  
# Visualise biosignals  
data = pd.DataFrame({"ECG": ecg,  
                     "PPG": ppg,  
                     "RSP": rsp,  
                     "EDA": eda,  
                     "EMG": emg})  
nk.signal_plot(data, subplots=True)
```

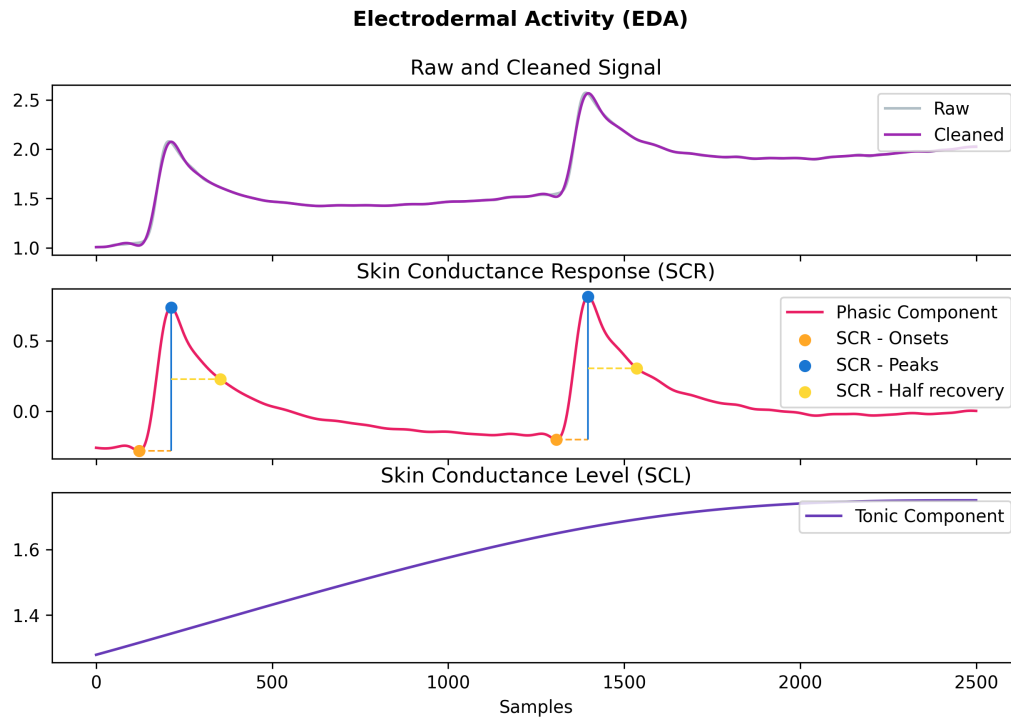


1.6.2 Electrodermal Activity (EDA/GSR)

```
# Generate 10 seconds of EDA signal (recorded at 250 samples / second) with 2 SCR_
↳peaks
eda = nk.eda_simulate(duration=10, sampling_rate=250, scr_number=2, drift=0.01)

# Process it
signals, info = nk.eda_process(eda, sampling_rate=250)

# Visualise the processing
nk.eda_plot(signals, sampling_rate=250)
```

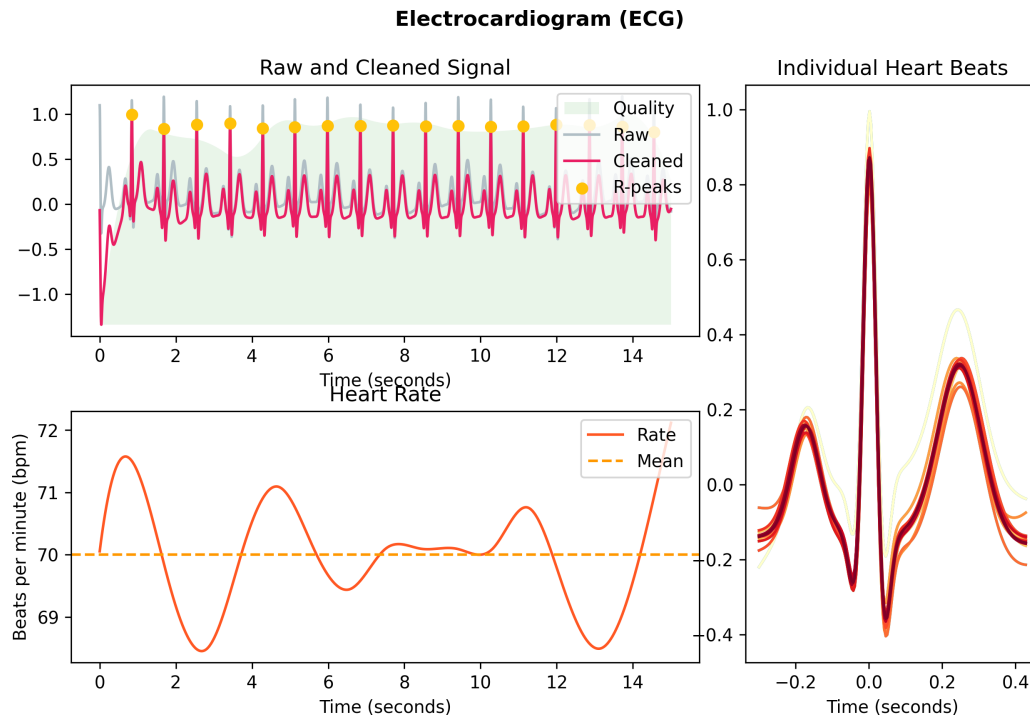


1.6.3 Cardiac activity (ECG)

```
# Generate 15 seconds of ECG signal (recorded at 250 samples / second)
ecg = nk.ecg_simulate(duration=15, sampling_rate=250, heart_rate=70)

# Process it
signals, info = nk.ecg_process(ecg, sampling_rate=250)

# Visualise the processing
nk.ecg_plot(signals, sampling_rate=250)
```

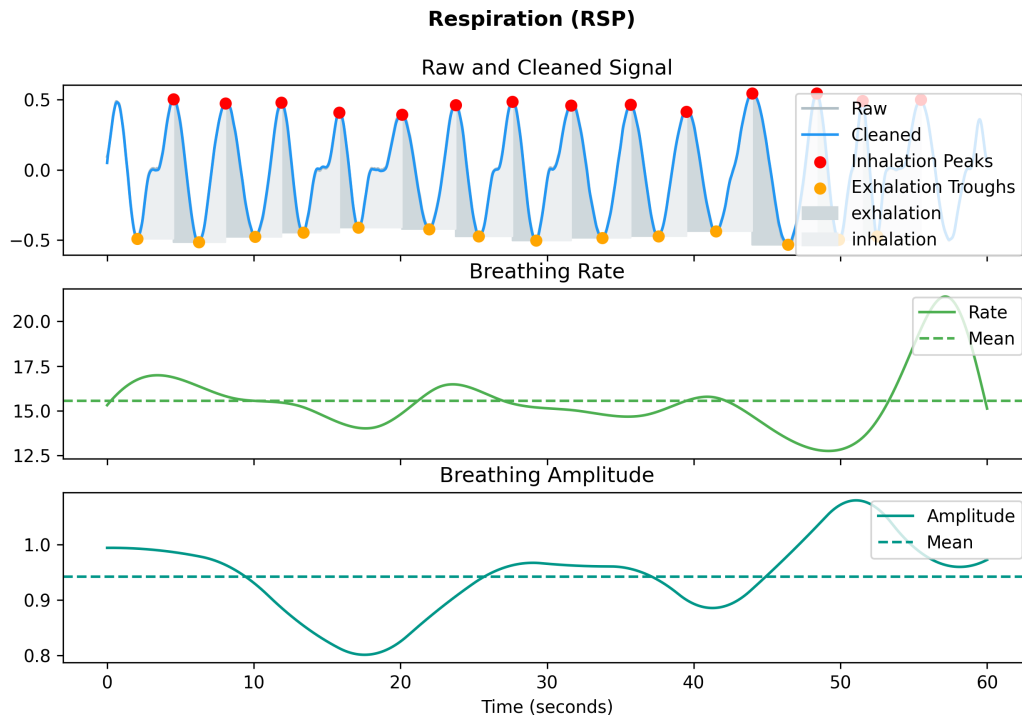



1.6.4 Respiration (RSP)

```
# Generate one minute of respiratory (RSP) signal (recorded at 250 samples / second)
rsp = nk.rsp_simulate(duration=60, sampling_rate=250, respiratory_rate=15)

# Process it
signals, info = nk.rsp_process(rsp, sampling_rate=250)

# Visualise the processing
nk.rsp_plot(signals, sampling_rate=250)
```

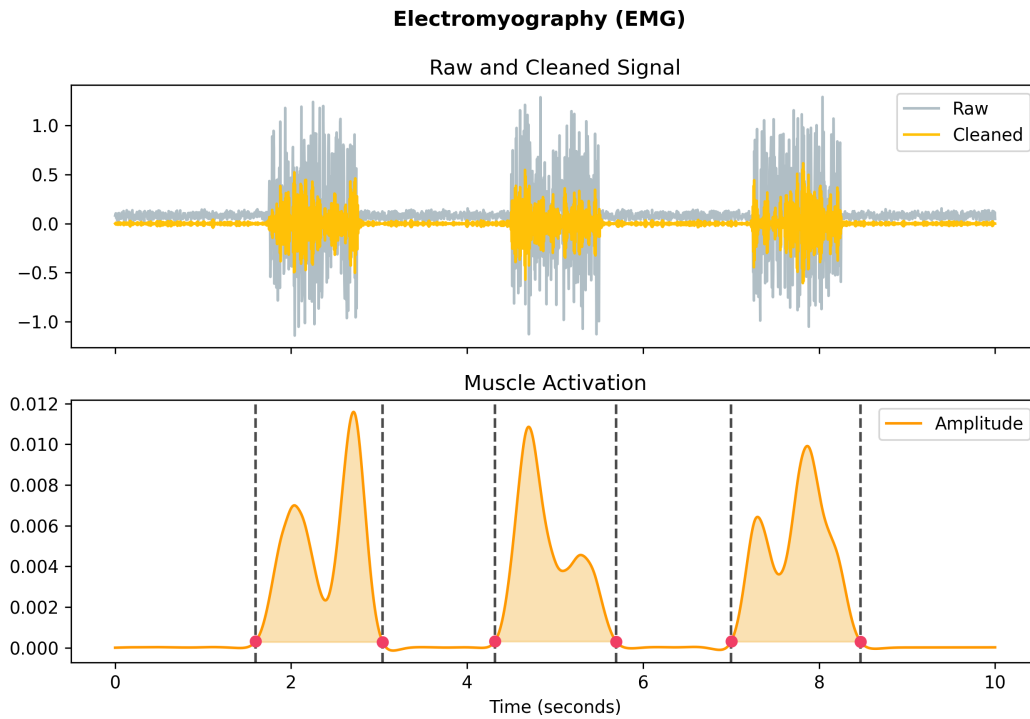


1.6.5 Electromyography (EMG)

```
# Generate 10 seconds of EMG signal (recorded at 250 samples / second)
emg = nk.emg_simulate(duration=10, sampling_rate=250, burst_number=3)

# Process it
signal, info = nk.emg_process(emg, sampling_rate=250)

# Visualise the processing
nk.emg_plot(signals, sampling_rate=250)
```

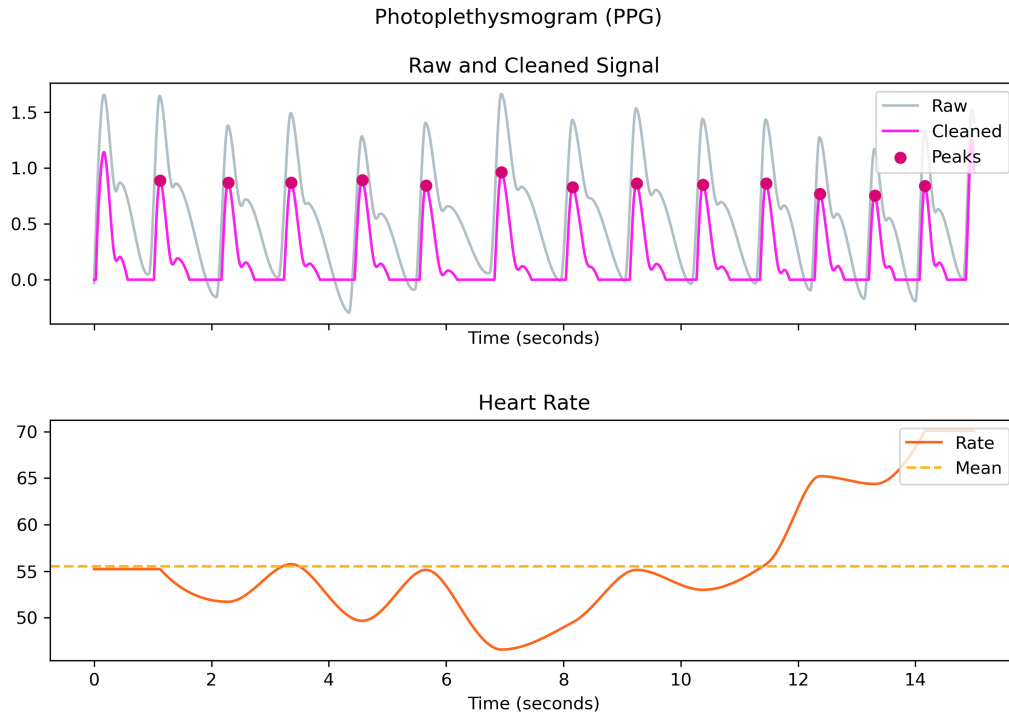


1.6.6 Photoplethysmography (PPG/BVP)

```
# Generate 15 seconds of PPG signal (recorded at 250 samples / second)
ppg = nk.ppg_simulate(duration=15, sampling_rate=250, heart_rate=70)

# Process it
signals, info = nk.ppg_process(ppg, sampling_rate=250)

# Visualize the processing
nk.ppg_plot(signals, sampling_rate=250)
```

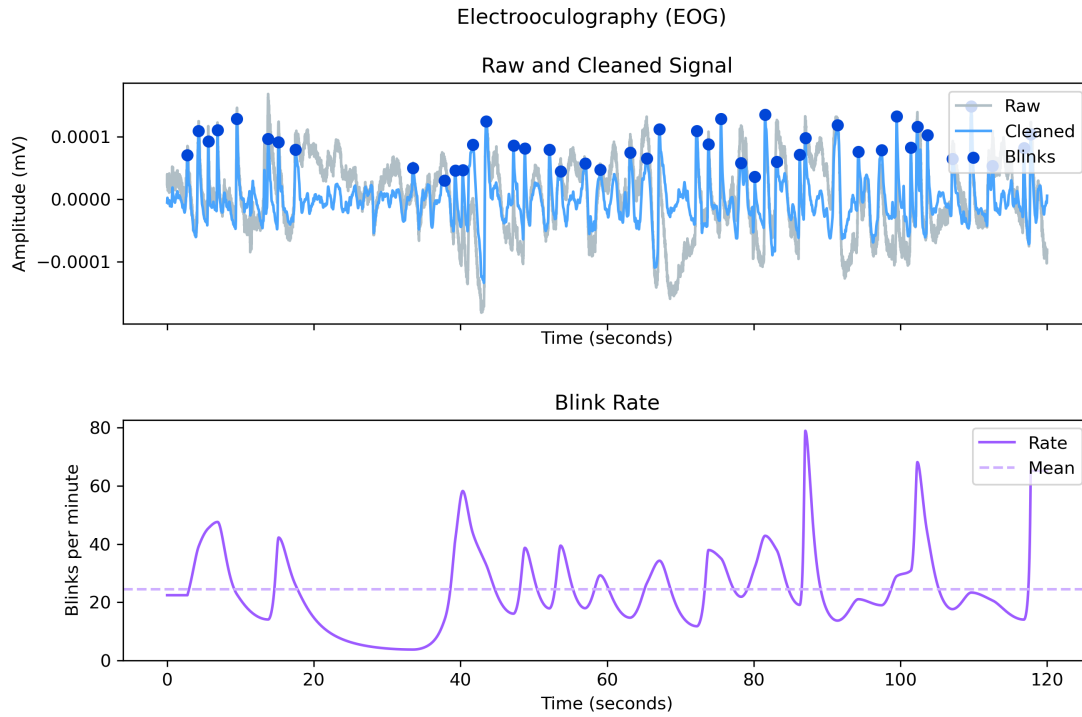


1.6.7 Electrooculography (EOG)

```
# Import EOG data
eog_signal = nk.data("eog_100hz")

# Process it
signals, info = nk.eog_process(eog_signal, sampling_rate=100)

# Plot
plot = nk.eog_plot(signals, sampling_rate=100)
```

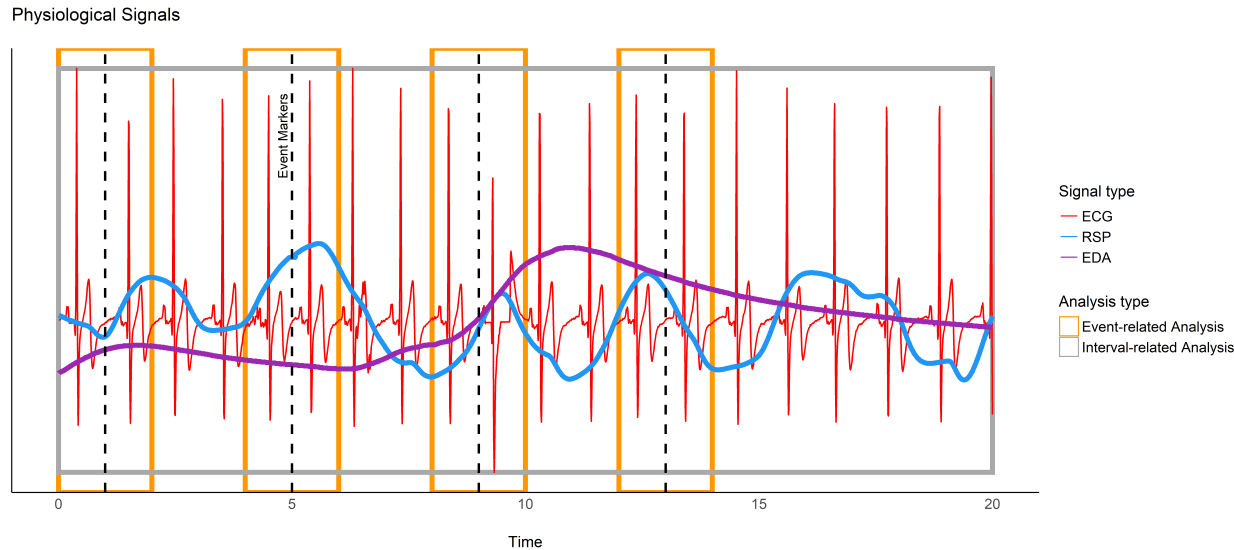


1.6.8 Electrogastrography (EGG)

Consider [helping us develop it!](#)

1.7 Physiological Data Analysis

The analysis of physiological data usually comes in two types, **event-related** or **interval-related**.



Event-related Features	Interval-related Features
ECG Rate Changes: Min, Mean, Max, Time of Min and Max, Trend (Linear, Quadratic, R2)	ECG Rate Characteristics: Mean, Amplitude
RSP Rate Changes: Min, Mean, Max, Time of Min and Max	Heart Rate Variability (HRV) metrics
RSP Amplitude Measures: Min, Mean, Max	Respiratory Rate Variability (RRV) metrics
ECG and RSP Phase Measures: Type (Inspiration/Expiration, Systole/Diastole), Completion	Respiratory Sinus Arrhythmia (RSA) metrics
EDA Phasic Max Peak Amplitude	Number of SCR Peaks
Number of SCRs and first SCR characteristics (Peak Amplitude, Risettime, Recovery time)	Mean of SCR Peaks Amplitude

1.7.1 Event-related

This type of analysis refers to physiological changes immediately occurring in response to an event. For instance, physiological changes following the presentation of a stimulus (e.g., an emotional stimulus) indicated by the dotted lines in the figure above. In this situation the analysis is epoch-based. An epoch is a short chunk of the physiological signal (usually < 10 seconds), that is locked to a specific stimulus and hence the physiological signals of interest are time-segmented accordingly. This is represented by the orange boxes in the figure above. In this case, using `bio_analyze()` will compute features like rate changes, peak characteristics and phase characteristics.

- [Event-related example](#)

1.7.2 Interval-related

This type of analysis refers to the physiological characteristics and features that occur over longer periods of time (from a few seconds to days of activity). Typical use cases are either periods of resting-state, in which the activity is recorded for several minutes while the participant is at rest, or during different conditions in which there is no specific time-locked event (e.g., watching movies, listening to music, engaging in physical activity, etc.). For instance, this type of analysis is used when people want to compare the physiological activity under different intensities of physical exercise, different types of movies, or different intensities of stress. To compare event-related and interval-related analysis, we can refer to the example figure above. For example, a participant might be watching a 20s-long short film where particular stimuli of interest in the movie appears at certain time points (marked by the dotted lines). While event-related analysis pertains to the segments of signals within the orange boxes (to understand the physiological changes pertaining to the appearance of stimuli), interval-related analysis can be applied on the entire 20s duration to investigate how physiology fluctuates in general. In this case, using `bio_analyze()` will compute features such as rate characteristics (in particular, variability metrics) and peak characteristics.

- Interval-related example

1.8 Miscellaneous

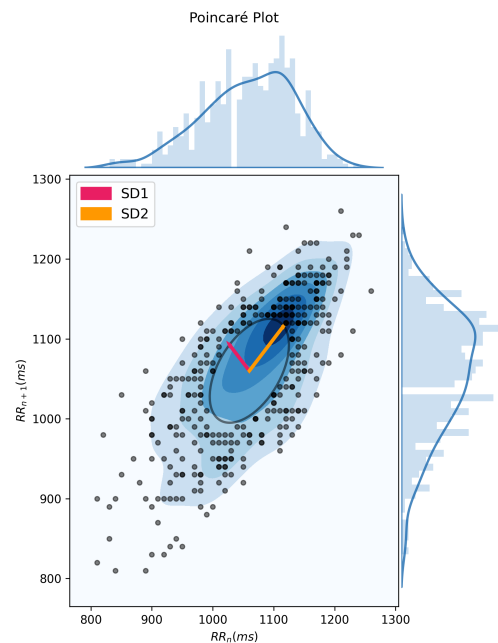
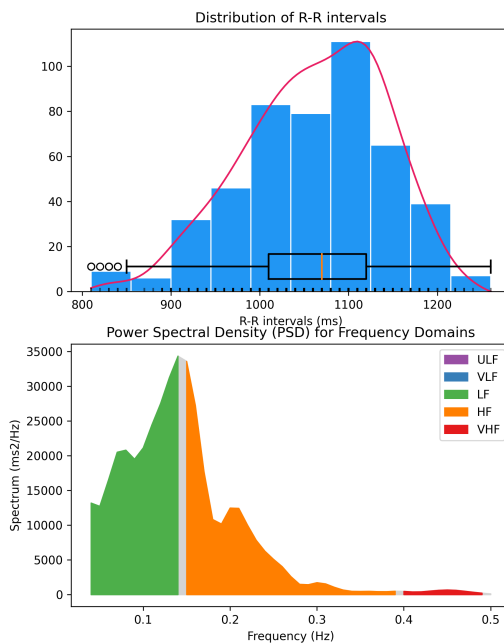
1.8.1 Heart Rate Variability (HRV)

- **Compute HRV indices**
 - **Time domain:** RMSSD, MeanNN, SDNN, SDSD, CVNN etc.
 - **Frequency domain:** Spectral power density in various frequency bands (Ultra low/ULF, Very low/VLF, Low/LF, High/HF, Very high/VHF), Ratio of LF to HF power, Normalized LF (LFn) and HF (HF_n), Log transformed HF (LnHF).
 - **Nonlinear domain:** Spread of RR intervals (SD1, SD2, ratio between SD2 to SD1), Cardiac Sympathetic Index (CSI), Cardial Vagal Index (CVI), Modified CSI, Sample Entropy (SampEn).

```
# Download data
data = nk.data("bio_resting_8min_100hz")

# Find peaks
peaks, info = nk.ecg_peaks(data["ECG"], sampling_rate=100)

# Compute HRV indices
nk.hrv(peaks, sampling_rate=100, show=True)
>>> HRV_RMSSD HRV_MeanNN HRV_SDNN ... HRV_CVI HRV_CSI_Modified HRV_SampEn
>>> 0 69.697983 696.395349 62.135891 ... 4.829101 592.095372 1.259931
```



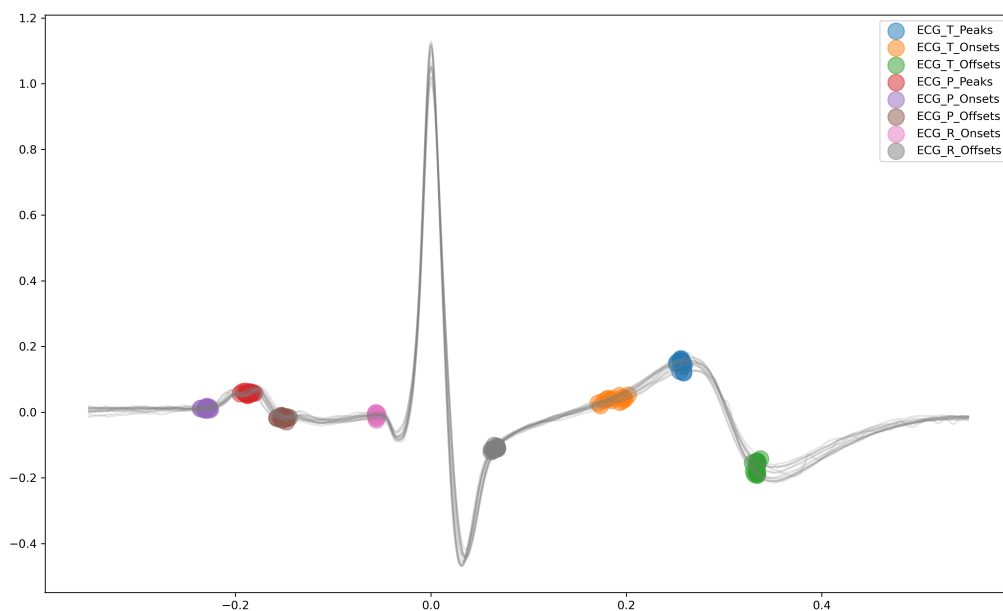
1.8.2 ECG Delineation

- Delineate the QRS complex of an electrocardiac signal (ECG) including P-peaks, T-peaks, as well as their onsets and offsets.

```
# Download data
ecg_signal = nk.data(dataset="ecg_3000hz")['ECG']

# Extract R-peaks locations
_, rpeaks = nk.ecg_peaks(ecg_signal, sampling_rate=3000)

# Delineate
signal, waves = nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="dwt",
→ show=True, show_type='all')
```



1.8.3 Signal Processing

- **Signal processing functionalities**
 - **Filtering:** Using different methods.
 - **Detrending:** Remove the baseline drift or trend.
 - **Distorting:** Add noise and artifacts.

```
# Generate original signal
original = nk.signal_simulate(duration=6, frequency=1)

# Distort the signal (add noise, linear trend, artifacts etc.)
distorted = nk.signal_distort(original,
                              noise_amplitude=0.1,
```

(continues on next page)

(continued from previous page)

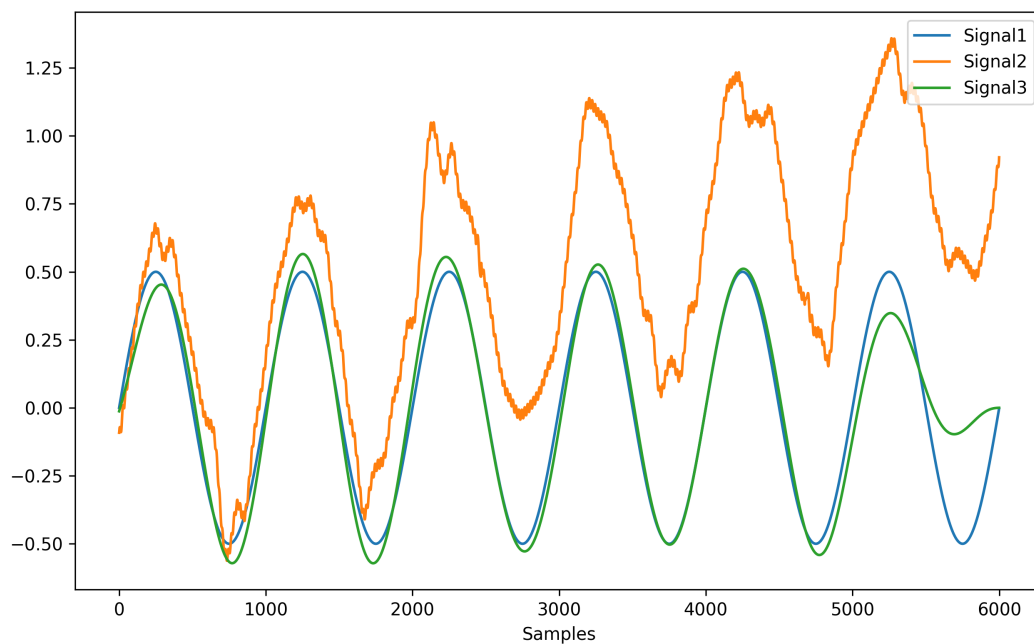
```

noise_frequency=[5, 10, 20],
powerline_amplitude=0.05,
artifacts_amplitude=0.3,
artifacts_number=3,
linear_drift=0.5)

# Clean (filter and detrend)
cleaned = nk.signal_detrend(distorted)
cleaned = nk.signal_filter(cleaned, lowcut=0.5, highcut=1.5)

# Compare the 3 signals
plot = nk.signal_plot([original, distorted, cleaned])

```



1.8.4 Complexity (Entropy, Fractal Dimensions, ...)

- Optimize complexity parameters (delay τ , dimension m , tolerance r)

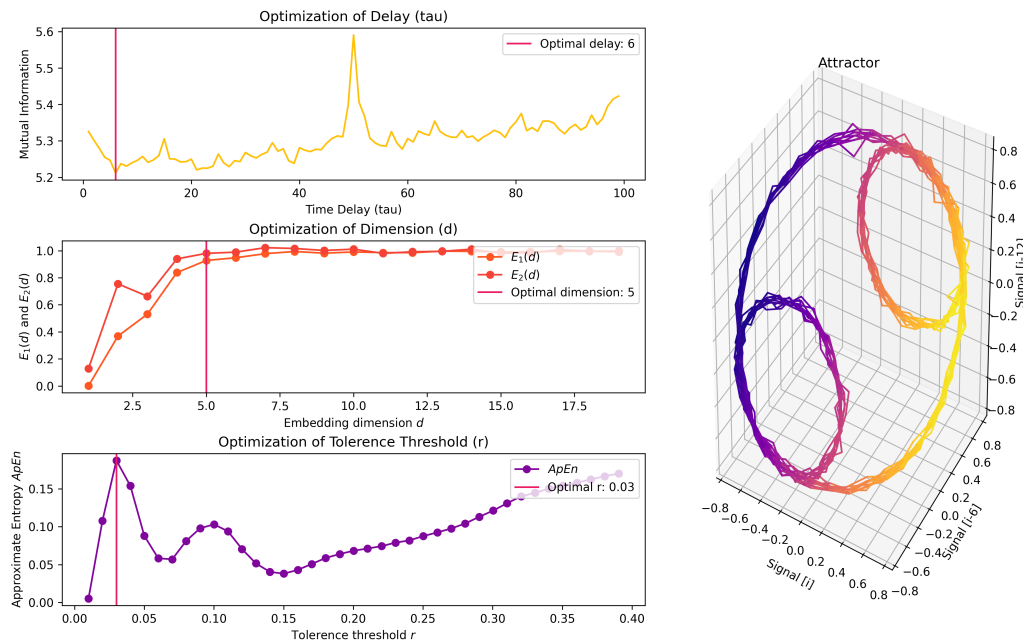
```

# Generate signal
signal = nk.signal_simulate(frequency=[1, 3], noise=0.01, sampling_rate=100)

# Find optimal time delay, embedding dimension and r
parameters = nk.complexity_optimize(signal, show=True)

```

Optimization of Complexity Parameters



- **Compute complexity features**

- **Entropy:** Sample Entropy (SampEn), Approximate Entropy (ApEn), Fuzzy Entropy (FuzzEn), Multiscale Entropy (MSE), Shannon Entropy (ShEn)
- **Fractal dimensions:** Correlation Dimension D2, ...
- **Detrended Fluctuation Analysis**

```
nk.entropy_sample(signal)
nk.entropy_approximate(signal)
```

1.8.5 Signal Decomposition

```
# Create complex signal
signal = nk.signal_simulate(duration=10, frequency=1) # High freq
signal += 3 * nk.signal_simulate(duration=10, frequency=3) # Higher freq
signal += 3 * np.linspace(0, 2, len(signal)) # Add baseline and linear trend
signal += 2 * nk.signal_simulate(duration=10, frequency=0.1, noise=0) # Non-linear trend
signal += np.random.normal(0, 0.02, len(signal)) # Add noise

# Decompose signal using Empirical Mode Decomposition (EMD)
components = nk.signal_decompose(signal, method='emd')
nk.signal_plot(components) # Visualize components

# Recompose merging correlated components
recomposed = nk.signal_recompose(components, threshold=0.99)
nk.signal_plot(recomposed) # Visualize components
```

1.8.6 Signal Power Spectrum Density (PSD)

```
# Generate signal with frequencies of 5, 20 and 30
signal = nk.signal_simulate(frequency=5) + 0.5*nk.signal_simulate(frequency=20) +
↳nk.signal_simulate(frequency=30)

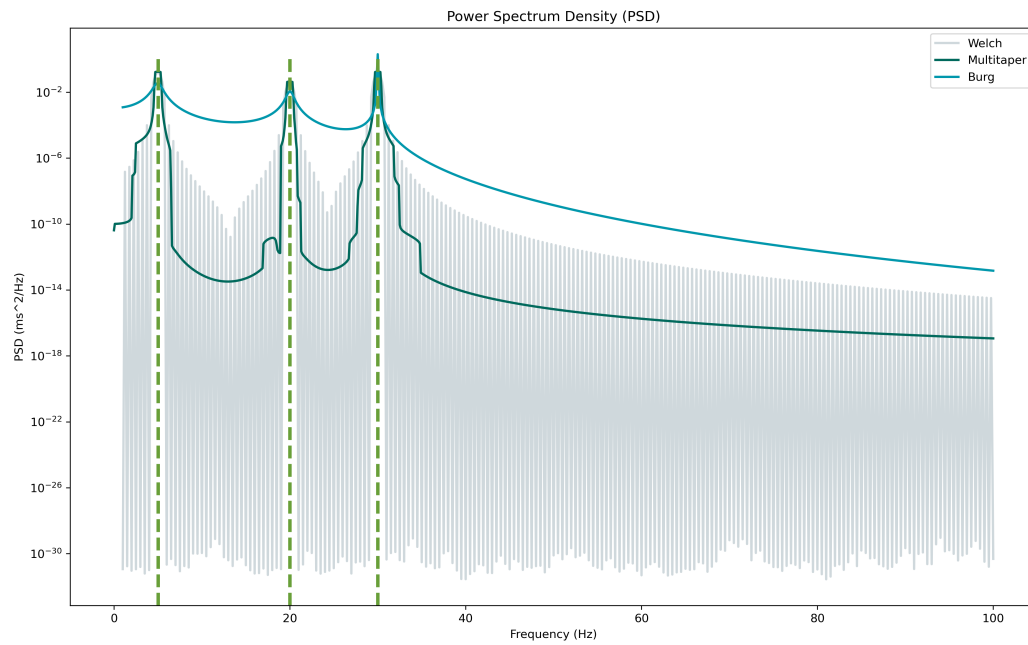
# Find Power Spectrum Density with different methods
# Multitaper
multitaper = nk.signal_psd(signal, method="multitapers", show=False, max_
↳frequency=100)
# Welch
welch = nk.signal_psd(signal, method="welch", min_frequency=1, show=False, max_
↳frequency=100)
# Burg
burg = nk.signal_psd(signal, method="burg", min_frequency=1, show=False, ar_
↳order=15, max_frequency=100)

# Visualize the different methods together
fig, ax = plt.subplots()

ax.plot(welch["Frequency"], welch["Power"], label="Welch", color="#CFD8DC",
↳linewidth=2)
ax.plot(multitaper["Frequency"], multitaper["Power"], label="Multitaper", color="
↳#00695C", linewidth=2)
ax.plot(burg["Frequency"], burg["Power"], label="Burg", color="#0097AC",
↳linewidth=2)

ax.set_title("Power Spectrum Density (PSD)")
ax.set_yscale('log')
ax.set_xlabel("Frequency (Hz)")
ax.set_ylabel("PSD (ms^2/Hz)")
ax.legend(loc="upper right")

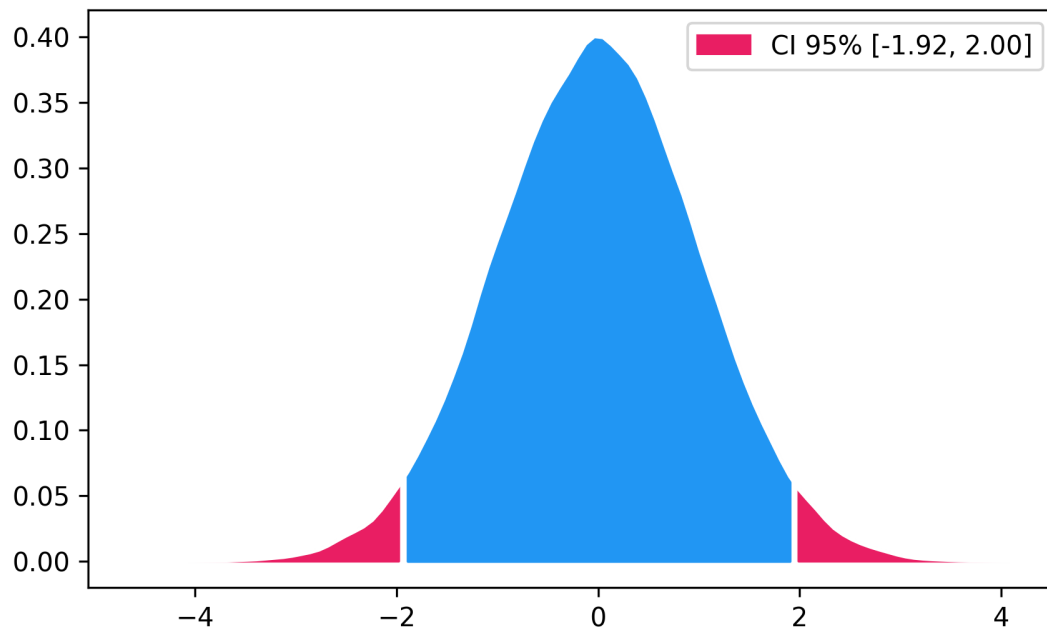
# Plot 3 frequencies of generated signal
ax.axvline(5, color="#689F38", linewidth=3, ymax=0.95, linestyle="--")
ax.axvline(20, color="#689F38", linewidth=3, ymax=0.95, linestyle="--")
ax.axvline(30, color="#689F38", linewidth=3, ymax=0.95, linestyle="--")
```



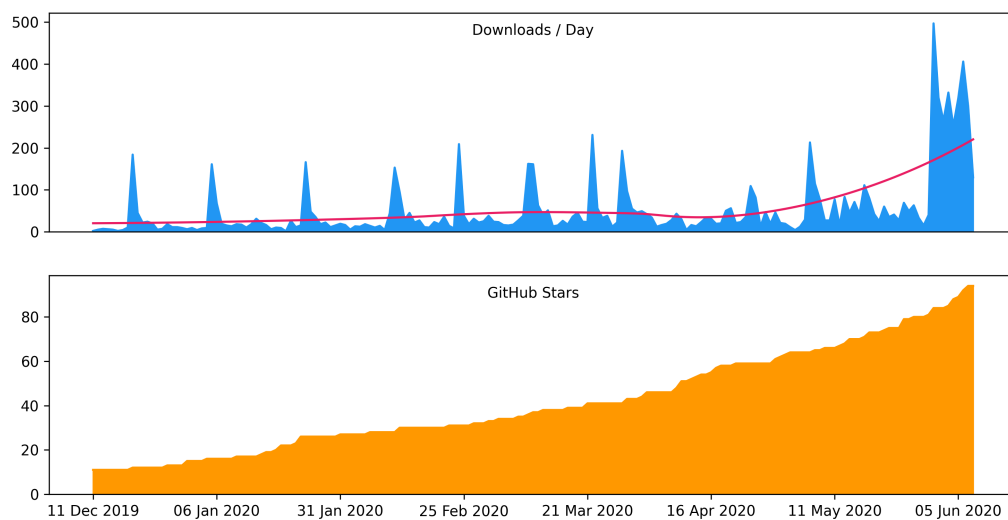
1.8.7 Statistics

- Highest Density Interval (HDI)

```
x = np.random.normal(loc=0, scale=1, size=100000)
ci_min, ci_max = nk.hdi(x, ci=0.95, show=True)
```



1.9 Popularity



1.10 Notes

The authors do not provide any warranty. If this software causes your keyboard to blow up, your brain to liquify, your toilet to clog or a zombie plague to break loose, the authors CANNOT IN ANY WAY be held responsible.

AUTHORS

Hint: Want to be a part of the project? Read how to [contribute and join us!](#)

2.1 Core team

- Dominique Makowski (*Nanyang Technological University, Singapore*)
- Tam Pham (*Nanyang Technological University, Singapore*)
- Zen Juen Lau (*Nanyang Technological University, Singapore*)
- Jan C. Brammer (*Radboud University, Netherlands*)
- François Lespinasse (*Université de Montréal, Canada*)

2.2 Contributors

- Hung Pham (*Eureka Robotics, Singapore*)
- Christopher Schölzel (*THM University of Applied Sciences, Germany*)
- Duy Le (*Hubble, Singapore*)
- Stavros Avramidis
- Tiago Rodrigues (*IST, Lisbon*)
- Mitchell Bishop (*NINDS, USA*)
- Robert Richer (*FAU Erlangen-Nürnberg, Germany*)
- Russell Anderson (*La Trobe Institute for Molecular Science, Australia*)

Thanks also to Gansheng Tan, Chuan-Peng Hu, @ucohen, Anthony Gatti, Julien Lamour, @renatosc, Nicolas Beaudoin-Gagnon and @rubinovitz for their contribution in NeuroKit 1.

More details [here](#).

INSTALLATION

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing ([see these guides](#)).

3.1 1. Python

3.1.1 Windows

Winpython

The advantage of Winpython is its portability (i.e., works out of a folder) and default setup (convenient for science).

1. Download a non-zero version of [Winpython](#)
2. Install it somewhere (the desktop is a good place). It creates a folder called *WPyXX-xxx*
3. In the *WPyXX-xxx* folder, open *WinPython Command Prompt.exe*
4. Run `pip install https://github.com/neuropsychology/NeuroKit/zipball/master`
5. Start *Spyder.exe*

Miniconda or Anaconda

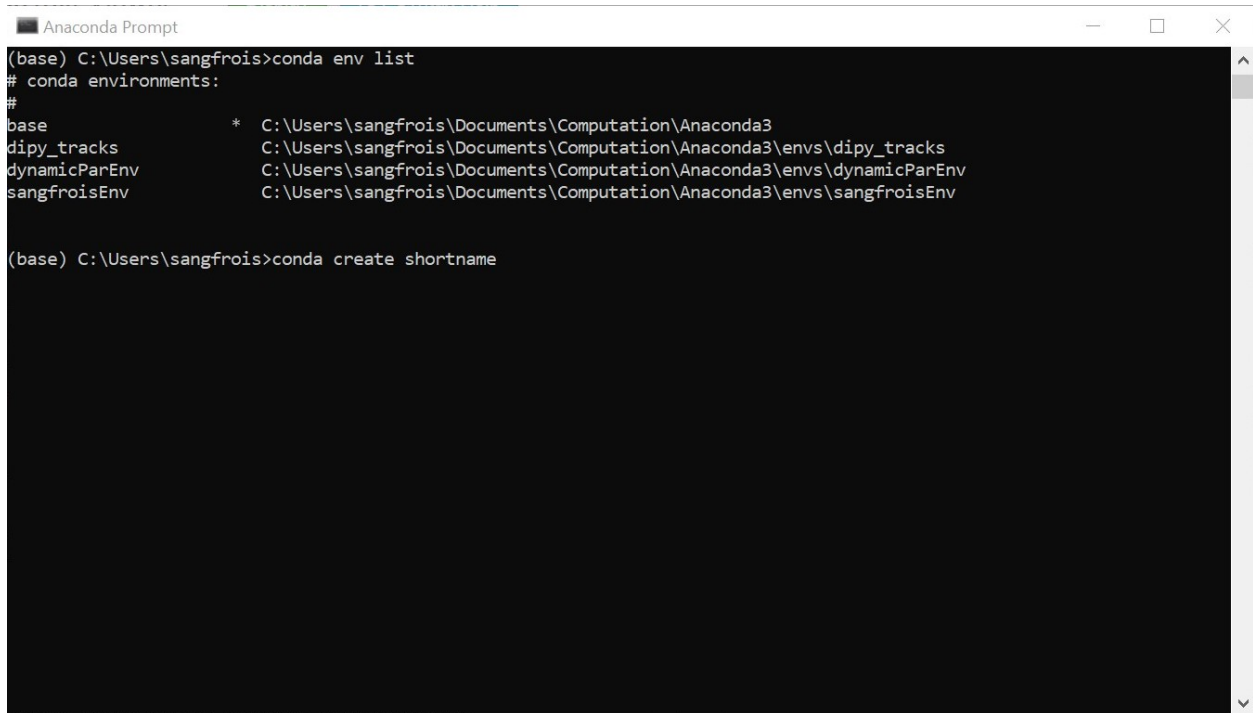
The difference between the two is straightforward, *miniconda* is recommended if you don't have much storage space and you know what you want to install. Similar to Winpython, Anaconda comes with a *base* environment, meaning you have basic packages pre-installed.

1. Download and install [Miniconda or Anaconda](#) (make sure the Anaconda3 directory is similar to this: `C:\Users\<username>\anaconda3\`)
2. Open the *Anaconda Prompt* (search for it on your computer)
3. Run `conda help` to see your options

Note: There should be a name in parentheses before your user's directory, e.g. `(base) C:\Users\<yourusername>`. That is the name of your computing environment. By default, you have a *base* environment. We don't want that, so create an environment.

4. Run `conda env create <yourenvname>`; activate it every time you open up conda by running `conda activate <yourenvname>`

5. Is pip (package installer for python) installed in this env? Prompt Anaconda using `pip list` it'll show you all the packages installed in that conda env

A screenshot of an Anaconda Prompt window. The title bar says "Anaconda Prompt". The terminal shows the following commands and output:

```
(base) C:\Users\sangfrois>conda env list
# conda environments:
#
base                * C:\Users\sangfrois\Documents\Computation\Anaconda3
dipy_tracks          C:\Users\sangfrois\Documents\Computation\Anaconda3\envs\dipy_tracks
dynamicParEnv        C:\Users\sangfrois\Documents\Computation\Anaconda3\envs\dynamicParEnv
sangfroisEnv         C:\Users\sangfrois\Documents\Computation\Anaconda3\envs\sangfroisEnv

(base) C:\Users\sangfrois>conda create shortname
```

3.1.2 Mac OS

1. Install [Anaconda](#)
2. Open the [terminal](#)
3. Run `source activate root`
4. Run `pip install neurokit2`
5. Start *Spyder.exe*

3.2 2. NeuroKit

If you already have python, you can install NeuroKit by running this command in your terminal:

```
pip install neurokit2
```

This is the preferred method to install NeuroKit, as it will always install the most stable release. It is also possible to install it directly from github:

```
pip install https://github.com/neuropsychology/neurokit/zipball/master
```

Hint: Enjoy living on the edge? You can always install the latest *dev* branch to access some work-in-progress features using `pip install https://github.com/neuropsychology/neurokit/zipball/dev`

If you don't have `pip` installed, this [Python installation guide](#) can guide you through the process.

GET STARTED

Contents:

4.1 Get familiar with Python in 10 minutes

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing ([see these guides](#)).

You have no experience in computer science? You are afraid of code? You feel betrayed because you didn't expect to do programming in psychology studies? **Relax!** We got you covered.

This tutorial will provide you with all you need to know to dive into the wonderful world of scientific programming. The goal here is not become a programmer, or a software designer, but rather to be able to use the power of programming to get **scientific results**.

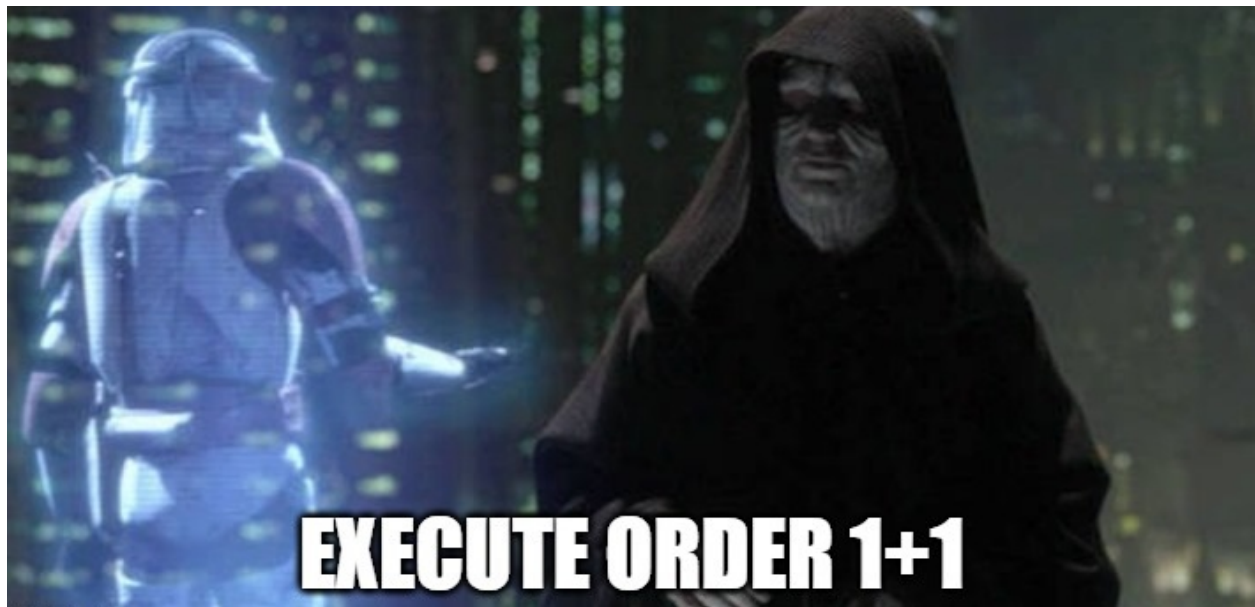
4.1.1 Setup

The first thing you will need is to **install Python** on your computer (we have a [tutorial for that](#)). In fact, this includes **two things**, installing Python (the *language*), and an *environment* to be able to use it. For this tutorial, we will assume you have something that looks like [Spyder](#) (called an IDE). But you can use [jupyter notebooks](#), or [anything else](#), it doesn't really matter.

There is one important concept to understand here: the difference between the **CONSOLE** and the **EDITOR**. The editor is like a *cooking table* where you prepare your ingredients to make a dish, whereas the console is like the *oven*, you only open it to put the dish in it and get the result.

Most of the code that you will write, you will write it in the editor. It's basically a text editor (such as notepad), except that it automatically highlights the code. Importantly, you can directly *execute* a line of code (which is equivalent to copy it and paste it the *console*).

For instance, try writing `1+1` somewhere in the file in the editor pane. Now if select the piece of code you just wrote, and press F9 (or CTRL + ENTER), it will **execute it**.



As a result, you should see in the console the order that you gave and, below, its **output** (which is 2).

Now that the distinction between where we write the code and where the output appears is clear, take some time to explore the settings and turn the editor background to **BLACK**. *Why?* Because it's more comfortable for the eyes, but most importantly, because it's cool .

The screenshot shows a Python IDE with a script editor on the left and an IPython console on the right. The script editor contains the following code:

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Apr 12 11:47:14 2020
4
5 @author: Hagrid
6 """
7
8
9
10
11
12
13
14
15
16 1+1
17
```

The IPython console on the right shows the following output:

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license()" for more information.

IPython 7.10.2 -- An enhanced Interactive Python.
Populating the interactive namespace from numpy and matplotlib

In [1]: 1+1
Out[1]: 2

In [2]:
```

Congrats, you've become a programmer, a wizard of the modern times.

You can now save the file (CTRL + S), which will be saved with a `.py` extension (i.e., a Python script). Try closing everything and reopening this file with the editor.

4.1.2 Variables

The most important concept of programming is **variables**, which is a fancy name for something that you already know. Do you remember, from your mathematics classes, the famous x , this placeholder for any value? Well, x was a variable, i.e., the name referring to some other thing.

Hint: Despite to what I just said, a variable in programming is not equivalent to a variable in statistics, in which it refers to some specific data (for instance, *age* is variable and contains multiple observations). In programming, a variable is simply the name that we give to some entity, that could be anything.

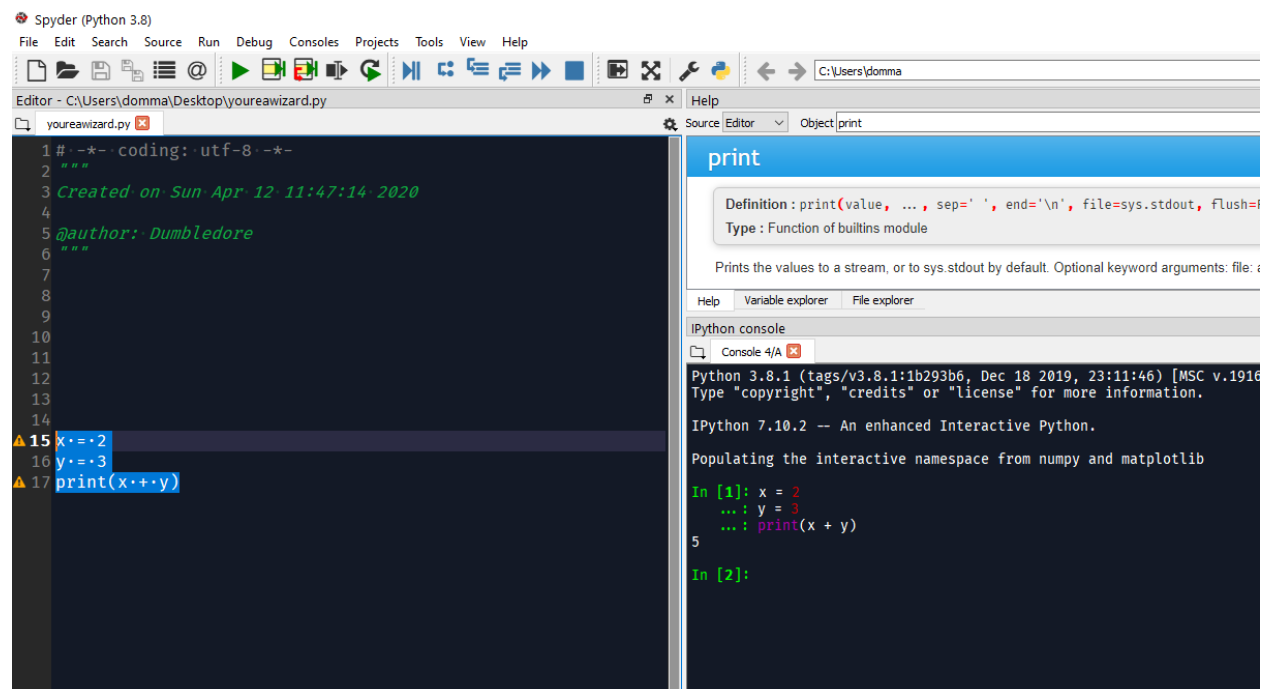
We can *assign* a value to a *variable* using the = sign, for instance:

```
x = 2
y = 3
```

Once we execute these two lines, Python will know that x refers to 2, and y to 3. We can now write:

```
print(x + y)
```

Which will print in the console the correct result.



We can also store the output in a third variable:

```
x = 2
y = 3
anothervariable = x + y
print(anothervariable)
```

4.1.3 Variables and data types

The next important thing to have in mind is that variables have **types**. Basic types include **integers** (numbers without decimals), **floats** (numbers with decimals), **strings** (character text) and **booleans** (True and False). Depending on their type, the variables will not behave in the same way. For example, try:

```
print(1 + 2)
print("1" + "2")
```

What happened here? Well, quotations ("I am quoted") are used to represent **strings** (i.e., text). So in the second line, the numbers that we added were not numbers, but text. And when you add strings together in Python, it *concatenates* them.

One can change the type of a variable with the following:

```
int(1.0) # transform the input to an integer
float(1) # transform the input to a float
str(1) # transform the input into text
```

Also, here I used the hashtag symbol to **make comments**, i.e., writing stuff that won't be executed by Python. This is super useful to annotate each line of your code to remember what you do - and why you do it.

Types are often the source of many errors as they usually are **incompatible** between them. For instance, you cannot add a *number* (int or float) with a *character string*. For instance, try running `3 + "a"`, it will throw a `TypeError`.

4.1.4 Lists and dictionaries

Two other important types are **lists** and **dictionaries**. You can think of them as **containers**, as they contain multiple variables. The main difference between them is that in a **list**, you access the individual elements that it contains **by its order** (for instance, “give me the third one”), whereas in a **dictionary**, you access an element by its name (also known as **key**), for example “give me the element named A”.

A list is created using square brackets, and a dictionary using curly brackets. Importantly, in a dictionary, you must specify a name to each element. Here's what it looks like:

```
mylist = [1, 2, 3]
mydict = {"A": 1, "B": 2, "C": 3}
```

Keep in mind that there are more types of containers, such as *arrays* and *dataframes*, that we will talk about later.

4.1.5 Basic indexing

There's no point in storing elements in containers if we cannot access them later on. As mentioned earlier, we can access elements from a **dictionary** by its key within square brackets (note that here the square brackets don't mean *list*, just mean *within the previous container*).

```
mydict = {"A": 1, "B": 2, "C": 3}
x = mydict["B"]
print(x)
```

Exercise time! If you have followed this tutorial so far, you should be able to guess what the following code will output:

```
mydict = {"1": 0, "2": 42, "x": 7}
x = str(1 + 1)
y = mydict[x]
print(y)
```

Answer: If you guessed **42**, you're right, congrats! If you guessed **7**, you have likely confused the **variable** named `x` (which represents `1+1` converted to a character), with the character `"x"`. And if you guessed **0**... what is wrong with you?

4.1.6 Indexing starts from 0

As mentioned earlier, one can access elements from a list by its **order**. However, **and there is very important to remember** (the source of many beginner errors), in Python, **the order starts from 0**. That means that the **first element is the 0th**.

So if we want the 2nd element of the list, we have to ask for the 1th:

```
mylist = [1, 2, 3]
x = mylist[1]
print(x)
```

4.1.7 Control flow (if and else)

One important notion in programming is control flow. You want the code to do something different depending on a condition. For instance, if `x` is lower than 3, print "lower than 3". In Python, this is done as follows:

```
x = 2
if x < 3:
    print("lower than 3")
```

One very important thing to notice is that the **if statement** corresponds to a "chunk" of code, as signified by the colon `:`. The chunk is usually written below, and has to be **indented** (you can indent a line or a chunk of code by pressing the TAB key).

What is indentation?

```
this
    is
        indentation
```

This indentation must be consistent: usually one level of indentation corresponds to 4 spaces. Make sure you respect that throughout your script, as this is very important in Python. If you break the rule, it will throw an **error**. Try running the following:

```
if 2 < 3:
print("lower than 3")
```

Finally, **if** statements can be followed by **else** statements, which takes care of what happens if the condition is not fulfilled:

```
x = 5
if x < 3:
    print("lower")
else:
    print("higher")
```

Again, note the **indentation** and how the **else** statement creates a new indented chunk.

4.1.8 For loops

One of the most used concept is **loops**, and in particular **for loops**. Loops are chunks of code that will be run several times, until a condition is complete.

The **for loops** create a *variable* that will successively take all the values of a list (or other **iterable** types). Let's look at the code below:

```
for var in [1, 2, 3]:  
    print(var)
```

Here, the **for loop** creates a variable (that we named *var*), that will successively take all the values of the provided list.

4.1.9 Functions

Now that you know what a **variable** is, as well as the purpose of little things like **if**, **else**, **for**, etc., the last most common thing that you will find in code are **function** calls. In fact, we have already used some of them! Indeed, things like `print()`, `str()` and `int()` were functions. And in fact, you've probably encountered them in secondary school mathematics! Remember $f(x)$?

One important thing about functions is that *most of the time* (not always though), it takes something **in**, and returns something **out**. It's like a **factory**, you give it some raw material and it outputs some transformed stuff.

For instance, let's say we want to transform a variable containing an *integer* into a character *string*:

```
x = 3  
x = str(x)  
print(x)
```

As we can see, our `str()` function takes *x* as an input, and outputs the transformed version, that we can collect using the equal sign `=` and store in the *x* variable to **replace** its content.

Another useful function is `range()`, that creates a sequence of integers, and is often used in combination with **for** loops. Remember our previous loop:

```
mylist = [1, 2, 3]  
for var in mylist:  
    print(var)
```

We can re-write it using the `range()` function, to create a sequence of **length 3** (which will be from 0 to 2; remember that Python indexing starts from 0!), and extracting and printing all of the elements in the list:

```
mylist = [1, 2, 3]  
for i in range(3):  
    print(mylist[i])
```

It's a bit more complicated than the previous version, it's true. But that's the beauty of programming, all things can be done in a near-infinite amount of ways, allowing for your creativity to be expressed.

Exercise time! Can you try making a loop so that we add :code: *1* to each element of the list?

Answer:


```
mylist = [1, 2, 3]
for i in range(3):
    mylist[i] = mylist[i] + 1
print(mylist)
```

If you understand what happened here, in this combination of lists, functions, loops and indexing, great! You are ready to move on.

4.1.10 Packages

Interestingly, Python alone does not include a lot of functions. **And that's also its strength**, because it allows to easily use functions developed by other people, that are stored in **packages** (or *modules*). A package is a collection of functions that can be downloaded and used in your code.

One of the most popular package is **numpy** (for *NUM**rical *PY*thon), including a lot of functions for maths and scientific programming. It is likely that this package is already ***installed** on your Python distribution. However, installing a package doesn't mean you can use it. In order to use a package, you have to **import it** (*load it*) in your script, before using it. This usually happens at the top of a Python file, like this:

```
import numpy
```

Once you have imported it (you have to run that line), you can use its functions. For instance, let's use the function to compute **square roots** included in this package:

```
x = numpy.sqrt(9)
print(x)
```

You will notice that we have to first **write the package name**, and then a **dot**, and then the `sqrt()` function. Why is it like that? Imagine you load two packages, both having a function named `sqrt()`. How would the program know which one to use? Here, it knows that it has to look for the `sqrt()` function in the `numpy` package.

You might think, *it's annoying to write the name of the package everytime*, especially if the package name is long. And this is why we sometimes use *aliases*. For instance, `numpy` is often loaded under the shortcut **np**, which makes it shorter to use:

```
import numpy as np

x = np.sqrt(9)
print(x)
```

4.1.11 Lists vs. vectors (arrays)

Packages can also add new **types**. One important type available through **numpy** is **arrays**.

In short, an array is a container, similar to a **list**. However, it can only contain one type of things inside (for instance, only *floats*, only *strings*, etc.) and can be multidimensional (imagine a 3D cube made of little cubes containing a value). If an array is one-dimensional (like a list, i.e., a sequence of elements), we can call it a **vector**.

A list can be converted to a vector using the `array()` function from the **numpy** package:

```
mylist = [1, 2, 3]
myvector = np.array(mylist)
print(myvector)
```

In signal processing, vectors are often used instead of lists to store the signal values, because they are more efficient and allow to do some cool stuff with it. For instance, remember our exercise above? In which we had to add `1` to each element of the list? Well using vectors, you can do this directly like this:

```
myvector = np.array([1, 2, 3])
myvector = myvector + 1
print(myvector)
```

Indeed, vectors allow for *vectorized* operations, which means that any operation is propagated on each element of the vector. And that's very useful for signal processing :)

4.1.12 Conditional indexing

Arrays can also be transformed in arrays of **booleans** (True or False) using a condition, for instance:

```
myvector = np.array([1, 2, 3, 2, 1])
vector_of_bools = myvector <= 2 # <= means inferior OR equal
print(vector_of_bools)
```

This returns a vector of the same length but filled with `True` (if the condition is respected) or `False` otherwise. And this new vector can be used as a **mask** to index and subset the original vector. For instance, we can select all the elements of the array that fulfills this condition:

```
myvector = np.array([1, 2, 3, 2, 1])
mask = myvector <= 2
subset = myvector[mask]
print(subset)
```

Additionally, we can also modify a subset of values on the fly:

```
myvector = np.array([1, 2, 3, 2, 1])
myvector[myvector <= 2] = 6
print(myvector)
```

Here we assigned a new value `6` to all elements of the vector that respected the condition (were inferior or equal to 2).

4.1.13 Dataframes

If you've followed everything until now, congrats! You're almost there. The last important type that we are going to see is **dataframes**. A dataframe is essentially a table with rows and columns. Often, the rows represent different **observations** and the columns different **variables**.

Dataframes are available in Python through the **pandas** package, another very used package, usually imported under the shortcut `pd`. A dataframe can be constructed from a *dictionary*: the **key** will become the **variable name**, and the list or vector associated will become the **variable values**.

```
import pandas as pd

# Create variables
var1 = [1, 2, 3]
var2 = [5, 6, 7]

# Put them in a dict
data = {"Variable1": var1, "Variable2": var2}
```

(continues on next page)

(continued from previous page)

```
# Convert this dict to a dataframe
data = pd.DataFrame.from_dict(data)

print(data)
```

This creates a dataframe with 3 rows (the observations) and 2 columns (the variables). One can access the variables by their name:

```
print(data["Variable1"])
```

Note that Python cares about the **case**: `tHiS` is not equivalent to `ThIs`. And `pd.DataFrame` has to be written with the *D* and *F* in capital letters. This is another common source of beginner errors, so make sure you put capital letters at the right place.

4.1.14 Reading data

Now that you know how to create a dataframe in Python, note that you also use **pandas** to read data from a file (.csv, excel, etc.) by its *path*:

```
import pandas as pd

data = pd.read_excel("C:/Users/Dumbledore/Desktop/myfile.xlsx") # this is an example
print(data)
```

Additionally, this can also read data directly from the internet! Try running the following:

```
import pandas as pd

data = pd.read_csv("https://raw.githubusercontent.com/neuropsychology/NeuroKit/master/
->data/bio_eventrelated_100hz.csv")
print(data)
```

4.1.15 Next steps

Now that you know the basis, and that you can distinguish between the different elements of Python code (functions calls, variables, etc.), we recommend that you dive in and try to follow our other examples and tutorials, that will show you some usages of Python to get something out of it.

4.2 Where to start

Hint: This page is under construction. Consider helping us developing it by [contributing](#).

Here are a few examples that are good for starting with NeuroKit.

- [Event-related Analysis](#)

EXAMPLES

The notebooks in this repo are meant to illustrate what you can do with NeuroKit. It is supposed to reveal how easy it has become to use cutting-edge methods, and still retain the liberty to change a myriad of parameters. These notebooks are organized in different sections that correspond to NeuroKit's modules.

5.1 Try the examples in your browser

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing ([see this tutorial](#)).

The notebooks in this repo are meant to illustrate what you can do with NeuroKit. It is supposed to reveal how easy it has become to use cutting-edge methods, and still retain the liberty to change a myriad of parameters. These notebooks are organized in different sections that correspond to NeuroKit's modules.

You are free to click on the link below to run everything... **without having to install anything!** There you'll find a Jupyterlab with notebooks ready to fire up. If you need [help figuring out the interface](#). (The secret is `shift+enter`).

5.2 1. Analysis Paradigm

Examples dedicated to specific analysis pipelines, such as for event related paradigms and resting state.

Ideas of examples to be implemented: > Preprocessing feature signals for machine learning Analysis > EEG + physiological activity during resting state > Comparing interval related activity from different "mental states" (e.g. meditation, induced emotion vs. neutral)

5.2.1 a) Event-related paradigm

`eventrelated.ipynb`

Description This notebook guides you through the initialization of events and epochs creation. It shows you how easy it is to compare measures you've extracted from different conditions.

5.2.2 b) Interval-related paradigm

`intervalrelated.ipynb`

Description Breaks down the step to extract characteristics of physiological activity for epochs of a minimum of a couple minutes

5.3 2. Biosignal Processing

Examples dedicated to processing pipelines, and measure extraction of multiple signals at a time. What's your thing ? How do you do it ?

Ideas of examples to be implemented:

```
> Batch preprocessing of multiple recordings
> PPG processing for respiration and temperature
> EMG overview (so many muscles to investigate)
> add yours...
```

5.3.1 a) Custom processing pipeline

`custom.ipynb`

Description This notebook breaks down the default NeuroKit pipeline used in `_process()` functions. It guides you in creating your own pipeline with the parameters best suited for your signals.

5.4 3. Heart rate and heart cycles

Examples dedicated to the analysis of ECG, PPG and HRV time series. Are you a fan of the [Neurovisceral integration model](#)? How would you infer a cognitive or affective process with HRV ? How do you investigate the asymmetry of cardiac cycles ?

Ideas of examples to be implemented:

```
> Benchmark different peak detection methods
> resting state analysis of HRV
> Comparing resting state and movie watching
> add yours
```

5.4.1 a) Detecting components of the cardiac cycle

`ecg_delineation.ipynb`

Description This notebook illustrate how reliable the peak detection is by analyzing the morphology of each cardiac cycles. It shows you how P-QRS-T components are extracted.

5.4.2 b) Looking closer at heart beats

heartbeats.ipynb

Description This notebook gives hints for a thorough investigation of ECG signals by visualizing individual heart beats, interactively.

5.5 4. Electrodermal activity

Examples dedicated to the analysis of EDA signals.

Ideas of examples to be implemented:

```
> Pain experiments
> Temperature
> add yours
```

5.5.1 a) Extracting information in EDA

eda.ipynb

Description This notebook goes at the heart of the complexity of EDA analysis by break down how Tonic and Phasic components are extracted from the signal.

5.6 5. Respiration rate and respiration cycles

Examples dedicated to the analysis of respiratory signals, i.e. as given by a belt, or eventually, with PPG.

Ideas of examples to be implemented:

```
> Meditation experiments
> Stress regulation
> add yours
```

5.6.1 a) Extracting Respiration Rate Variability metrics

rrv.ipynb

Description This notebook breaks down the extraction of variability metrics done by `rsp_rrv()`

5.7 6. Muscle activity

Examples dedicated to the analysis of EMG signals.

Ideas of examples to be implemented:

```
> Suggestion and muscle activation
> Sleep data analysis
>... nothing yet!
```

5.8 Simulate Artificial Physiological Signals

Neurokit's core signal processing functions surround electrocardiogram (ECG), respiratory (RSP), electrodermal activity (EDA), and electromyography (EMG) data. Hence, this example shows how to use Neurokit to simulate these physiological signals with customized parametric control.

```
[1]: import sys
      sys.stdout = open(os.devnull, 'w')

[2]: # Load NeuroKit and other useful packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

[3]: plt.rcParams['figure.figsize'] = 10, 6 # Bigger images
```

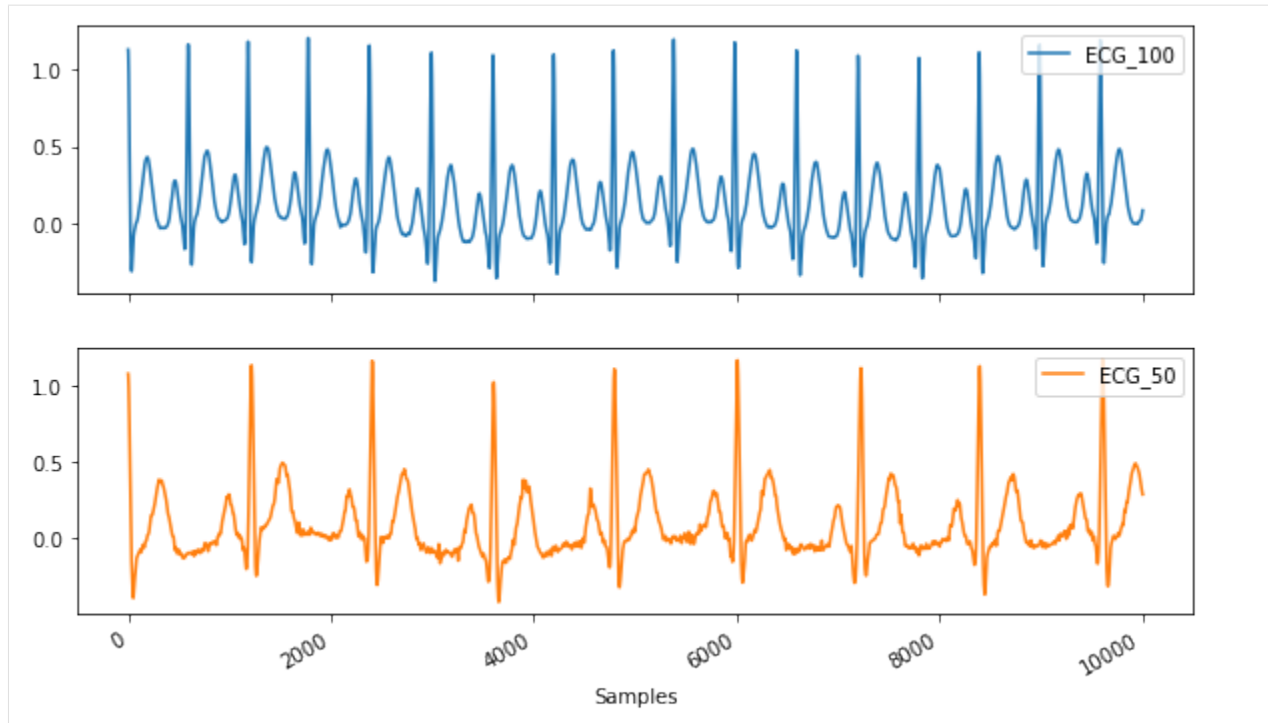
5.8.1 Cardiac Activity (ECG)

With `ecg_simulate()`, you can generate an artificial ECG signal of a desired length (in this case here, `duration=10`), noise, and heart rate. As you can see in the plot below, *ecg50* has about half the number of heart beats than *ecg100*, and *ecg50* also has more noise in the signal than the latter.

```
[4]: # Alternate heart rate and noise levels
     duration = 10, noise = 0.05, heart_rate = 50
     duration = 10, noise = 0.01, heart_rate = 100

# Visualize
     plt.figure(figsize=(10, 6))
     plt.plot(ecg_100)
     plt.plot(ecg_50)

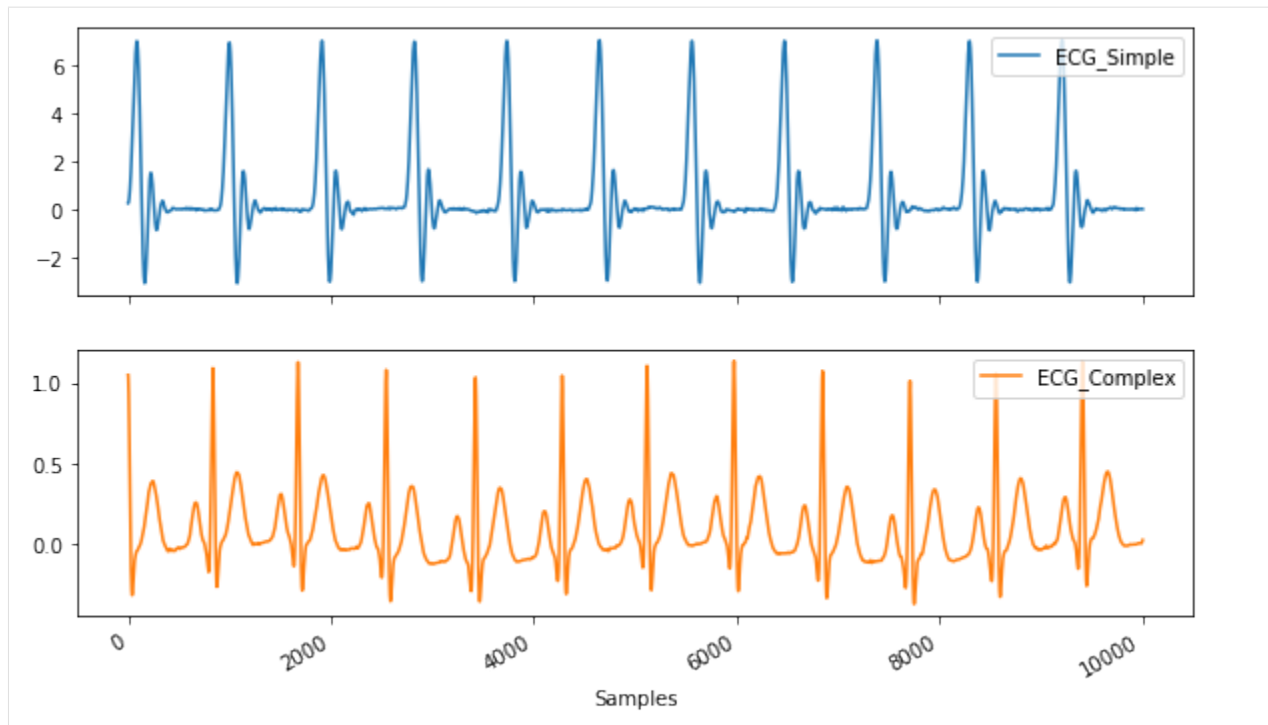
     plt.show()
```

You can also choose to generate the default, simple simulation based on Daubechies wavelets, which roughly approximates one cardiac cycle, or a more complex one by specifying `method="ecgsyn"`.

```
[5]: # Alternate methods
      = .                                =10      ="simple"
      = .                                =10      ="ecgsyn"

# Visualize
      = .                                "ECG_Simple"
      "ECG_Complex"
      .                                =True
```

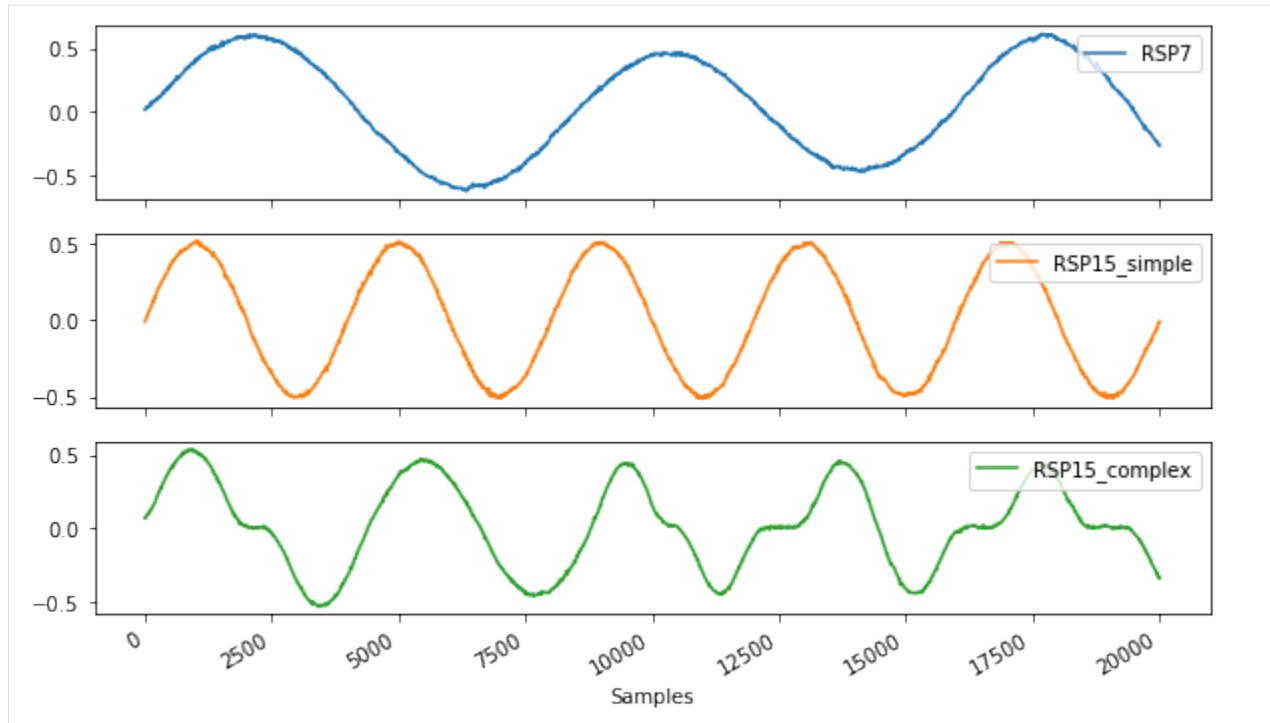


5.8.2 Respiration (RSP)

To simulate a synthetic respiratory signal, you can use `rsp_simulate()` and choose a specific duration and breathing rate. In this example below, you can see that `rsp7` has a lower breathing rate than `rsp15`. You can also decide which model you want to generate the signal. The *simple* `rsp15` signal incorporates `method = "sinusoidal"` which approximates a respiratory cycle based on the trigonometric sine wave. On the other hand, the *complex* `rsp15` signal specifies `method = "breathmetrics"` which uses a more advanced model by interpolating inhalation and exhalation pauses between each respiratory cycle.

```
[6]: # Simulate
      = .                                =20                                =15                                ="sinusoidal"
      = .                                =20                                =15                                ="breathmetrics"
      = .                                =20                                =7                                 ="breathmetrics"

# Visualize respiration rate
      = .                                "RSP7"
      "RSP15_simple"
      "RSP15_complex"
      =True
```

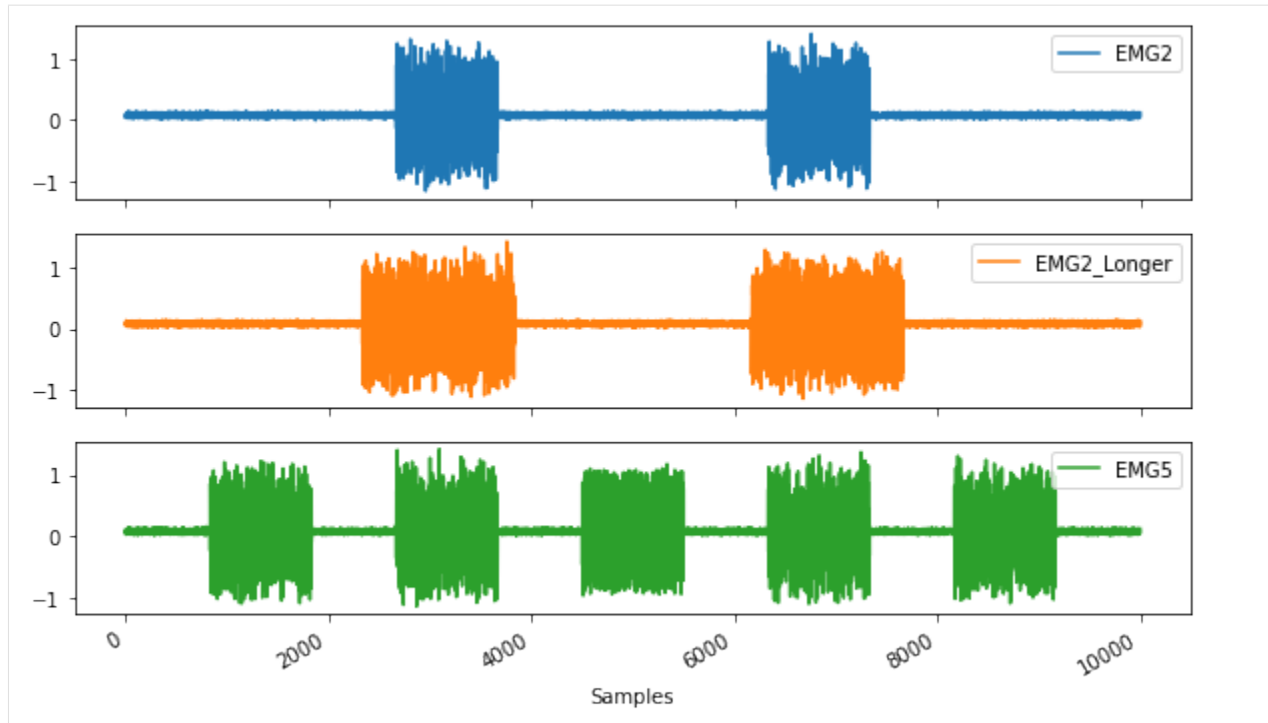


5.8.3 Electromyography (EMG)

Now, we come to generating an artificial EMG signal using `emg_simulate()`. Here, you can specify the number of bursts of muscular activity (`n_bursts`) in the signal as well as the duration of the bursts (`duration_bursts`). As you can see the active muscle periods in *EMG2_Longer* are greater in duration than that of *EMG2*, and *EMG5* contains more bursts than the former two.

```
[7]: # Simulate
      = . =10 =2 =1.0
      = . =10 =2 =1.5
      = . =10 =5 =1.0

# Visualize
      = . "EMG2"
          "EMG2_Longer"
          "EMG5"
      =True
```

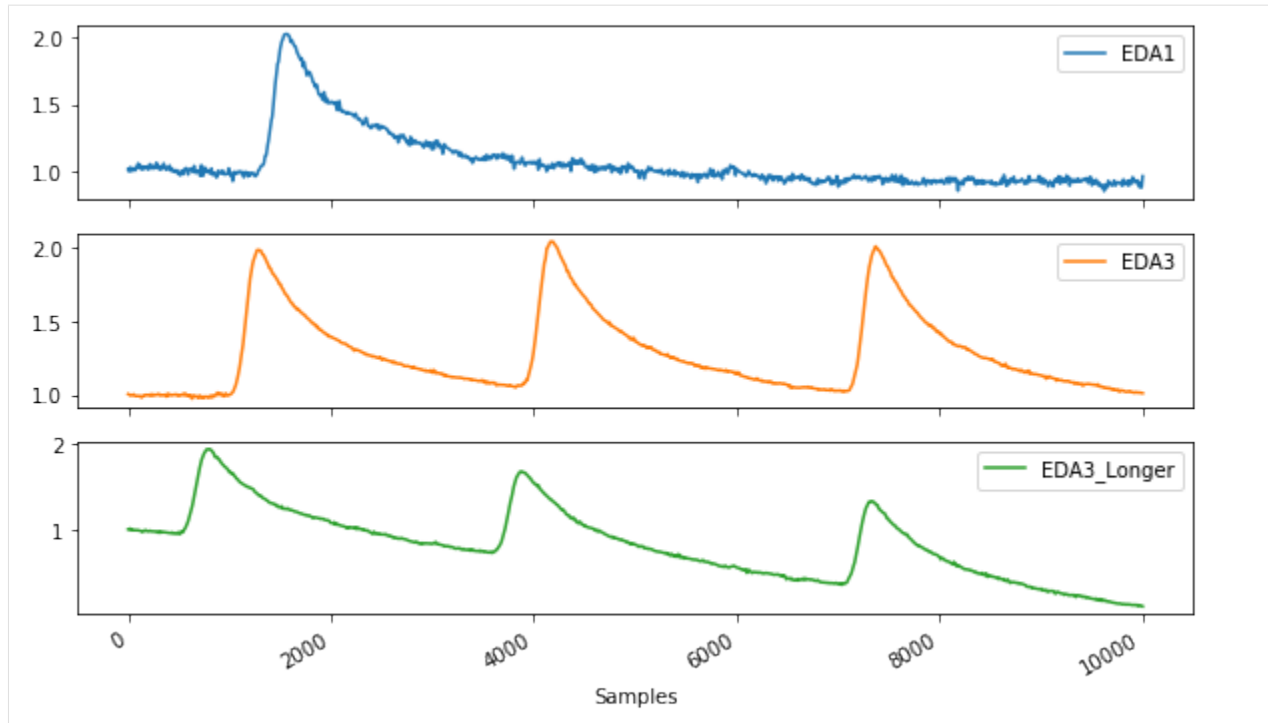


5.8.4 Electrodermal Activity (EDA)

Finally, `eda_simulate()` can be used to generate a synthetic EDA signal of a given duration, specifying the number of skin conductance responses or activity ‘peaks’ (`n_scr`) and the `drift` of the signal. You can also modify the noise level of the signal.

```
[8]: # Simulate
      = .                                =10          =1          =-0.01      =0.05
      = .                                =10          =3          =-0.01      =0.01
      = .                                =10          =3          =-0.1       =0.01

# Visualize
      = .                                "EDA1"
      = .                                "EDA3"
      = .                                "EDA3_Longer"
      = .                                =True
```



[]:

5.9 Customize your Processing Pipeline

While *NeuroKit* is designed to be beginner-friendly, experts who desire to have more control over their own processing pipeline are also offered the possibility to tune functions to their specific usage. This example shows how to use *NeuroKit* to customize your own processing pipeline for advanced users taking ECG processing as an example.

```
[1]: # Load NeuroKit and other useful packages
import sys
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
%matplotlib notebook

[2]: plt.rcParams['figure.figsize'] = 15, 9 # Bigger images
```

5.9.1 The Default NeuroKit processing pipeline

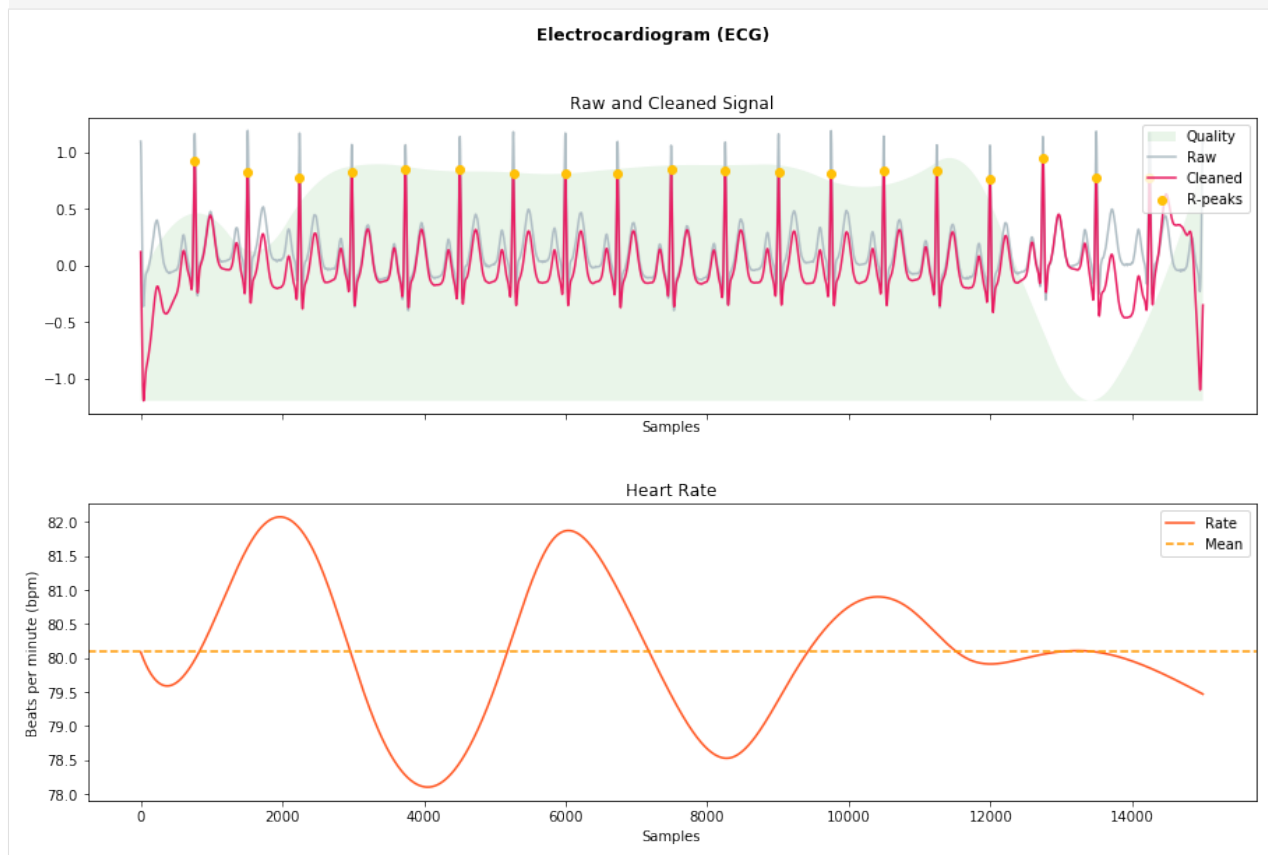
NeuroKit provides a very useful set of functions, `*_process()` (e.g. `ecg_process()`, `eda_process()`, `emg_process()`, ...), which are all-in-one functions that cleans, preprocesses and processes the signals. It includes good and sensible defaults that should be suited for most of users and typical use-cases. That being said, in some cases, you might want to have more control over the processing pipeline.

This is how `ecg_process()` is typically used:

```
[3]: # Simulate ecg signal (you can use your own one)
      = . =15 =1000 =80

      # Default processing pipeline
      = . =1000

      # Visualize
      = .
```



5.9.2 Building your own `process()` function

Now, if you look at the code of `ecg_process()` (https://github.com/neuropsychology/NeuroKit/blob/master/neurokit2/ecg/ecg_process.py#L49) (see [here](#) for how to explore the code), you can see that it is in fact very simple.

It uses what can be referred to as “mid-level functions”, such as `ecg_clean()`, `ecg_peaks()`, `ecg_rate()` etc.

This means that you can basically **re-create** the `ecg_process()` function very easily by calling these mid-level functions:

```
[22]: # Define a new function
def my_processing
    # Do processing

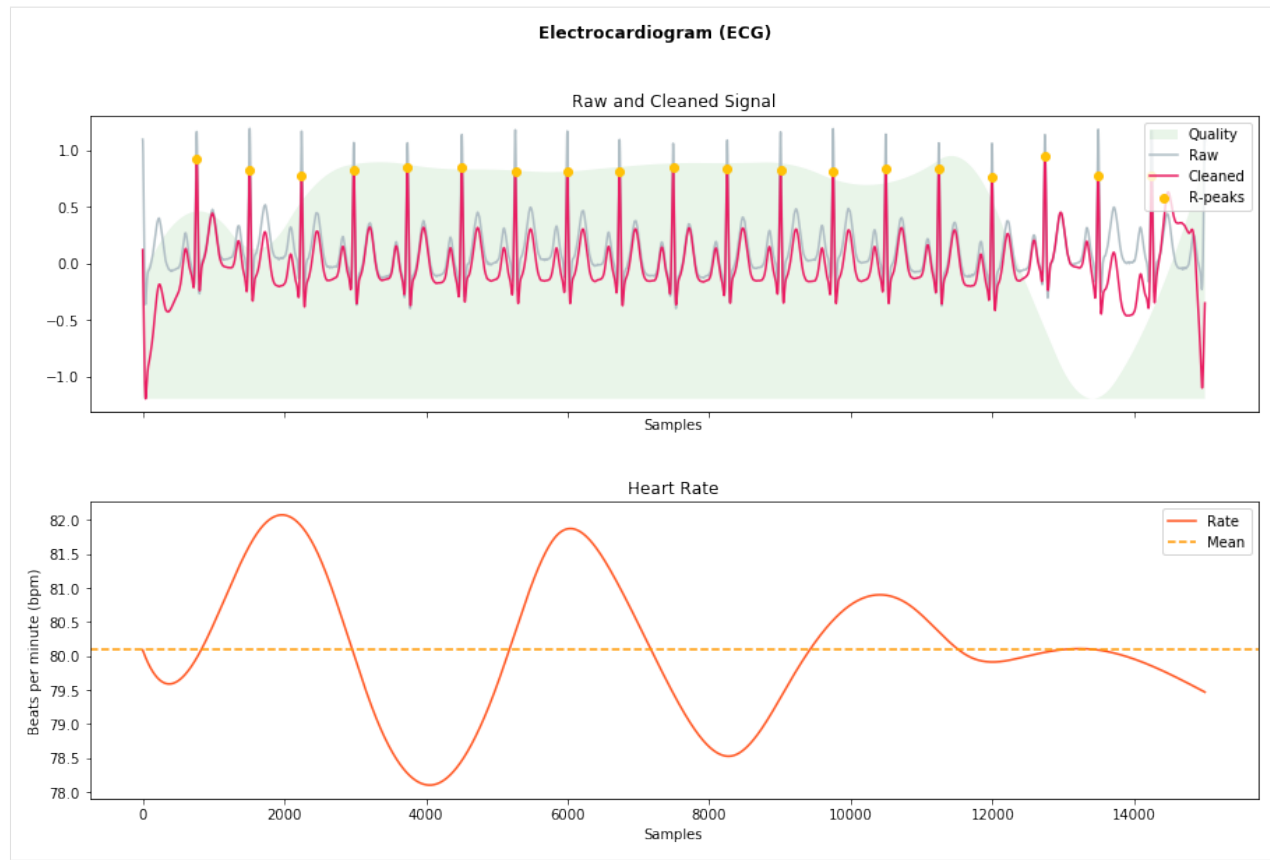
    # Prepare output

    return
```

You can now use this function as you would do with `ecg_process()`.

```
[23]: # Process the signal using previously defined function

# Visualize
```



5.9.3 Changing the processing parameters

Now, you might want to ask, why would you re-create the processing function? Well, it allows you to **change the parameters** of the inside as you please. Let's say you want to use a specific **cleaning** method.

First, let's look at the *documentation for ``ecg_clean()``* <https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.ecg_clean>, you can see that there are several different methods for cleaning which can be specified. The default is the **Neurokit** method, however depending on the quality of your signal (and several other factors), other methods may be more appropriate. It is up to you to make this decision.

You can now change the methods as you please for each function in your custom processing function that you have written above:

```
[24]: # Define a new function
def my_processing
    # Do processing
        = .
        = .
        =1000
        ="engzeemod2012"
        =1000
        =1000
        =1000
        =1000

    # Prepare output
        = .
        "ECG_Raw"
        "ECG_Clean"
        "ECG_Rate"
        "ECG_Quality"
```

(continues on next page)

(continued from previous page)

```

        = .
        =1
    =
    return

```

Similarly, you can select a different method for the peak detection.

5.9.4 Customize even more!

It is possible that none of these methods suit your needs, or that you want to test a new method. Rejoice yourself, as *NeuroKit* allows you to do that by providing what can be referred to as “low-level” functions.

For instance, you can rewrite the **cleaning** procedure by using the **signal processing tools** offered by NeuroKit:

```

[25]: def my_cleaning
        = .
        =1
        =
        =2
        =9
        = 'butterworth'
    return

```

You can use this function inside your custom processing written above:

```

[26]: # Define a new function
def my_processing
    # Do processing
        =
        =1000
        = .
        =1000
        =1000
        =1000
        =
        # Prepare output
        = .
        "ECG_Raw"
        "ECG_Clean"
        "ECG_Rate"
        "ECG_Quality"
        =
        =1
    return

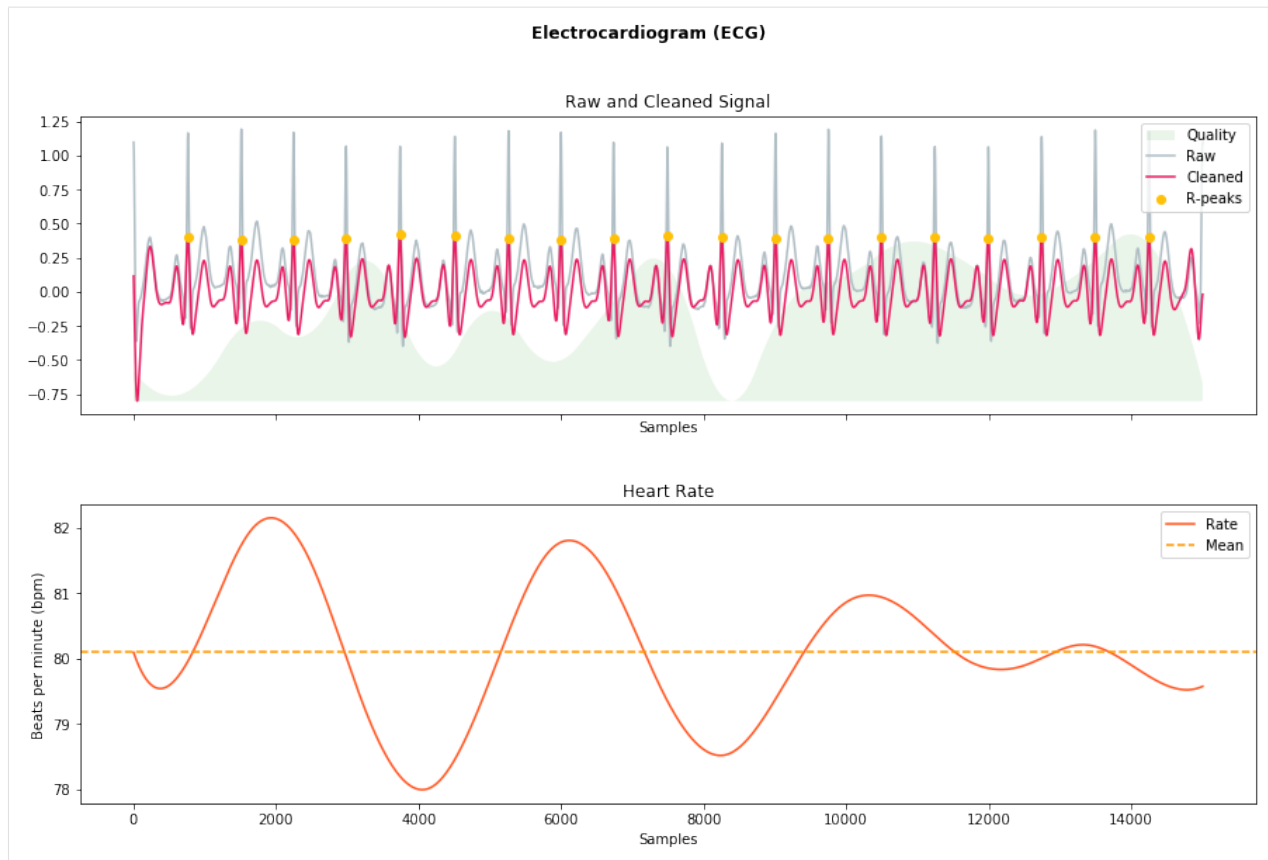
```

Congrats, you have created your own processing pipeline! Let’s see how it performs:

```

[27]: = .
        =

```



This doesn't look bad :) **Can you do better?**

5.10 Event-related Analysis

This example shows how to use Neurokit to extract epochs from data based on events localisation and its corresponding physiological signals. That way, you can compare *experimental conditions* with one another.

```
[1]: # Load NeuroKit and other useful packages
import sys as sys
import numpy as np as np
import matplotlib.pyplot as plt
import neurokit2 as nk
%matplotlib inline

[2]: . figure(figsize=(15, 5)) # Bigger images
      . font.size = 14
```

5.10.1 The Dataset

Use the `nk.data()` function to load the dataset located on Neurokit data folder.

It contains 2.5 minutes of biosignals recorded at a frequency of 100Hz ($2.5 \times 60 \times 100 = 15000$ data points).

Biosignals : **ECG, RSP, EDA + Photosensor (event signal)**

```
[3]: # Get data
      = . "bio_eventrelated_100hz"
```

This is the data from **1 participant** to whom was presented 4 images (emotional stimuli, [IAPS-like emotional faces](#)), which we will refer to as **events**.

Importantly, the images were marked by a small black rectangle on the screen, which led to the photosensor signal to go down (and then up again after the image). This is what will allow us to retrieve the location of these events.

They were 2 types (the **condition**) of images that were shown to the participant: **“Negative”** vs. **“Neutral”** in terms of emotion. Each picture was presented for 3 seconds. The following list is the condition order.

```
[4]: = "Negative" "Neutral" "Neutral" "Negative"
```

5.10.2 Find Events

These events can be localized and extracted using `events_find()`.

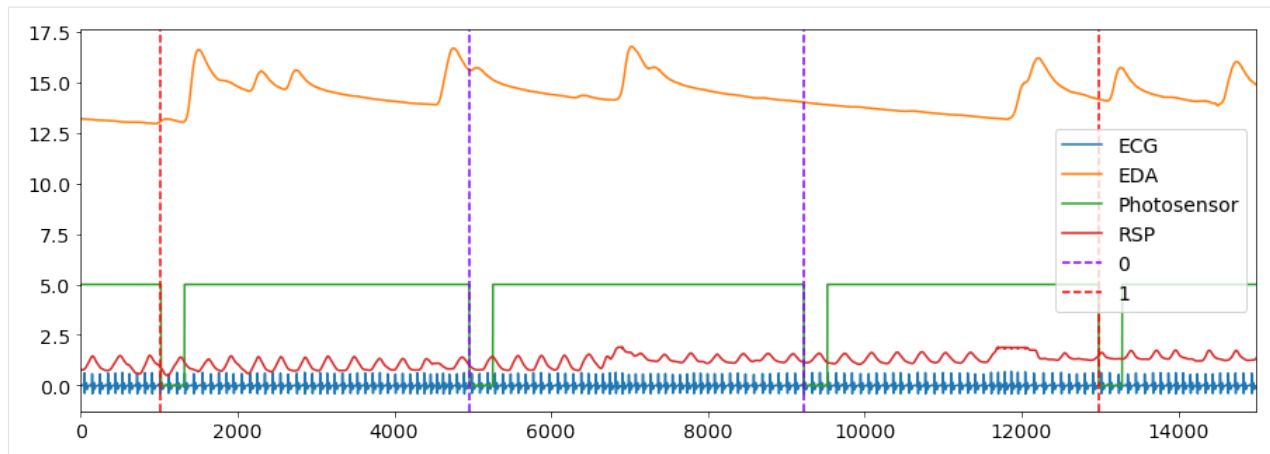
Note that you should also specify whether to select events that are higher or below the threshold using the `threshold_keep` argument.

```
[5]: # Find events
      = . "Photosensor" = 'below'
      ↪ =
```

```
[5]: {'onset': array([ 1024,  4957,  9224, 12984]),
      'duration': array([300, 300, 300, 300]),
      'label': array(['1', '2', '3', '4'], dtype='<U11'),
      'condition': ['Negative', 'Neutral', 'Neutral', 'Negative']}
```

As we can see, `events_find()` returns a dict containing onsets and durations for each corresponding event, based on the label for event identifiers and each event condition. Each event here lasts for 300 data points (equivalent to 3 seconds sampled at 100Hz).

```
[6]: # Plot the location of event with the signals
      = .
```



The output of `events_plot()` shows the corresponding events in the signal, with the blue dashed line representing a Negative event and red dashed line representing a Neutral event.

5.10.3 Process the Signals

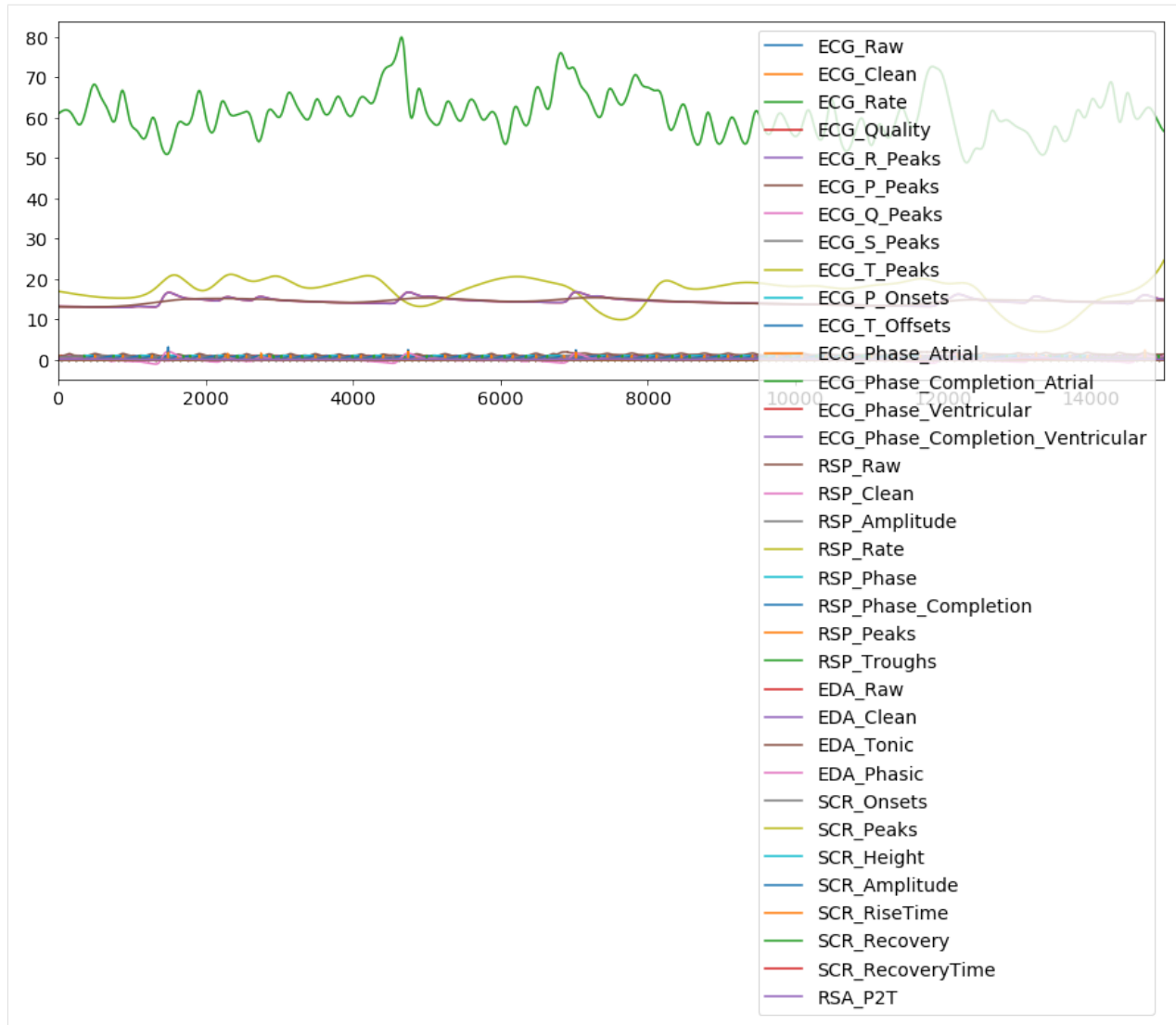
Now that we have the events location, we can go ahead and process the data.

Biosignals processing can be done quite easily using NeuroKit with the `bio_process()` function. Simply provide the appropriate biosignal channels and additional channels that you want to keep (for example, the photosensor), and `bio_process()` will take care of the rest. It will return a dataframe containing processed signals and a dictionary containing useful information.

```
[7]: # Process the signal
      = .
      = "ECG" = "RSP" = "EDA"
      =100

      # Visualize
      .
      # theres is a ton of features now, but_
      =not in epochs
```

```
[7]: <matplotlib.axes._subplots.AxesSubplot at 0x1c6b9b8c898>
```



5.10.4 Create Epochs

We now have to transform this dataframe into **epochs**, i.e. segments (chunks) of data around the **events** using `epochs_create()`.

```
1. We want it to start *1 second before the event onset*
2. and end *6 seconds* afterwards
```

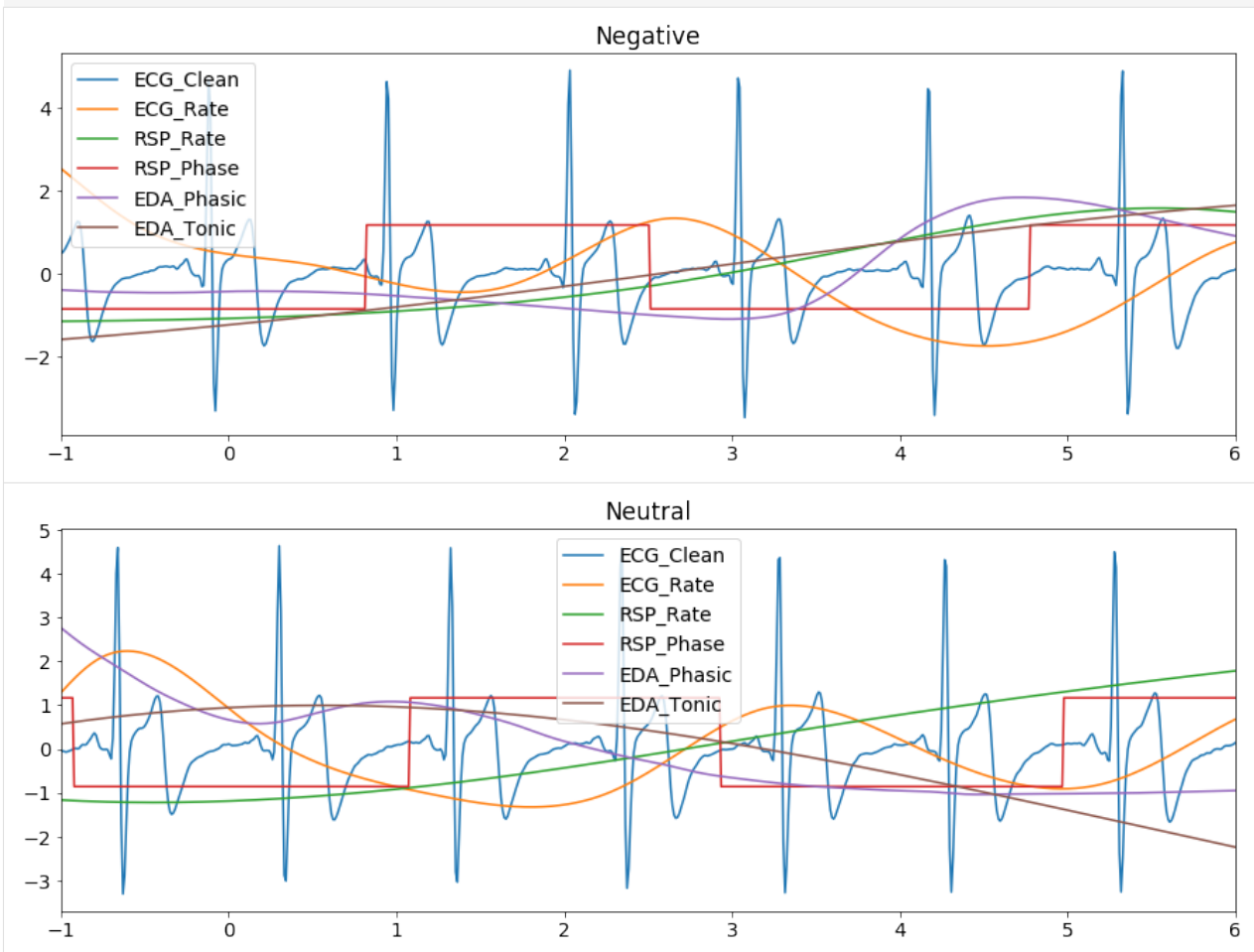
These are passed into the `epochs_start` and `epochs_end` arguments, respectively.

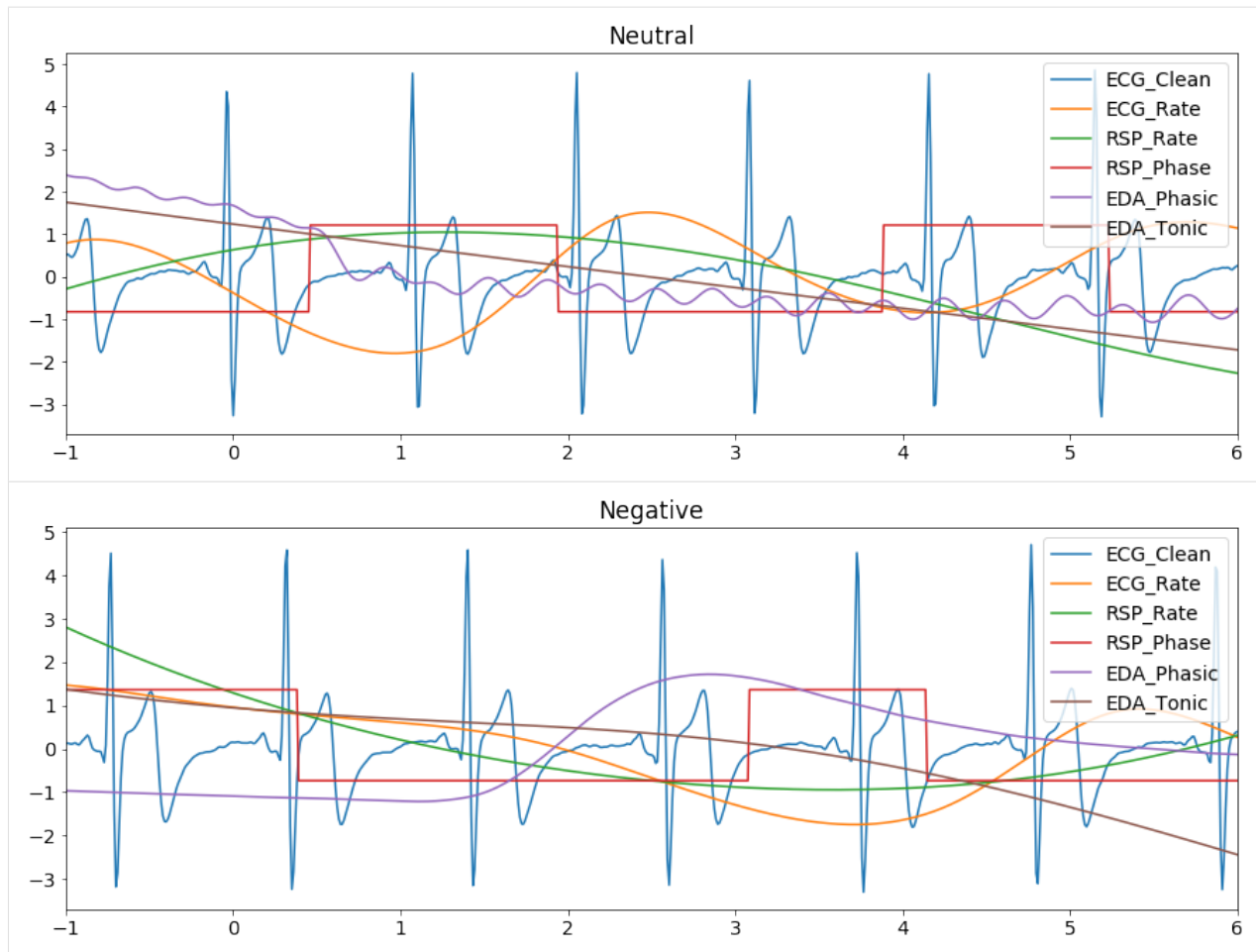
Our epochs will then cover the region from **-1 s to +6 s** (i.e., 700 data points since the signal is sampled at 100Hz).

```
[8]: # Build and plot epochs
      = . =100 =-1
      =6
```

Let's plot some of the signals of the first epoch (and transform them to the same scale for visualization purposes).

```
[9]: for condition in condition_list:
    # iterate epochs",
    for epoch in range(1, epochs+1):
        ECG_Clean, ECG_Rate, RSP_Rate, RSP_Phase, EDA_Phasic, EDA_Tonic = \
            get_signals(condition, epoch, scaled=True)  # Select relevant columns
            # get title from condition list",
            # Plot scaled signals"
```





5.10.5 Extract Event Related Features

With these segments, we are able to compare how the physiological signals vary across the different events. We do this by: 1. **Iterating through our object epochs**

2. **Storing the mean value of X feature of each condition in a new dictionary**

3. **Saving the results in a readable format**

We can call them *epochs-dictionary*, the *mean-dictionary* and our *results-dataframe*.

```
[10]: = # Initialize an empty dict,

for in # then Initialize an empty dict inside of it with the_
↪ iterative

    # Save a temp var with dictionary called <epoch_index> in epochs-dictionary
    =

    # We want its features:

    # Feature 1 ECG
```

(continues on next page)

(continued from previous page)

```

        = "ECG_Rate" . -100 0 . # Baseline
        = "ECG_Rate" . 0 400 . # Mean heart rate in the 0-4_
seconds
# Store ECG in df
    "ECG_Rate" = - # Correct for baseline

# Feature 2 EDA - SCR
    = "SCR_Amplitude" . 0 600 . # Maximum SCR peak
# If no SCR, consider the magnitude, i.e. that the value is 0
if .
    = 0
# Store SCR in df
    "SCR_Magnitude" =

# Feature 3 RSP
    = "RSP_Rate" . -100 0 . # Baseline
    = "RSP_Rate" . 0 600 .
# Store RSP in df
    "RSP_Rate" = - # Correct for baseline

= . . = "index" # Convert to a dataframe
"Condition" = # Add the conditions
# Print DataFrame

```

```

[10]:   ECG_Rate  SCR_Magnitude  RSP_Rate Condition
1 -4.286137      3.114808  2.729480   Negative
2 -5.387987      0.000000  2.094437    Neutral
3 -1.400696      0.000000 -0.062720    Neutral
4 -3.804883      1.675922 -1.674218   Negative

```

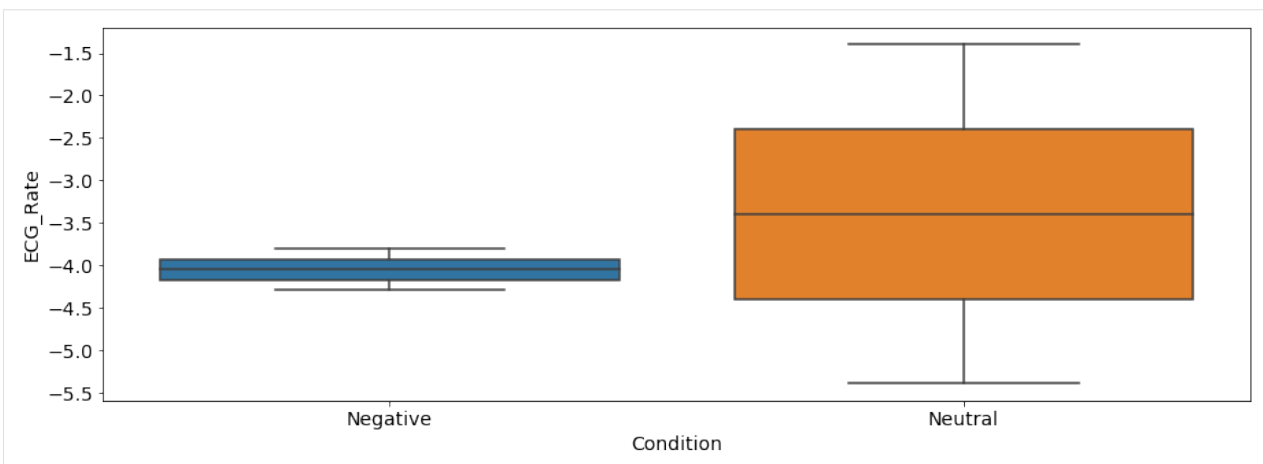
5.10.6 Plot Event Related Features

You can now plot and compare how these features differ according to the event of interest.

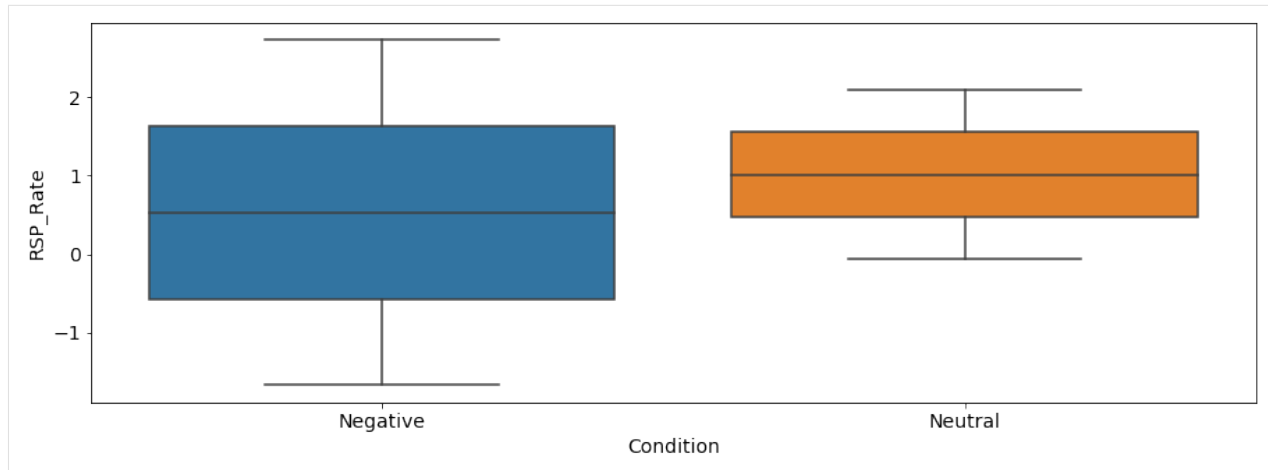
```

[11]: . = "Condition" = "ECG_Rate" =
[11]: <matplotlib.axes._subplots.AxesSubplot at 0x1c6bd25e828>

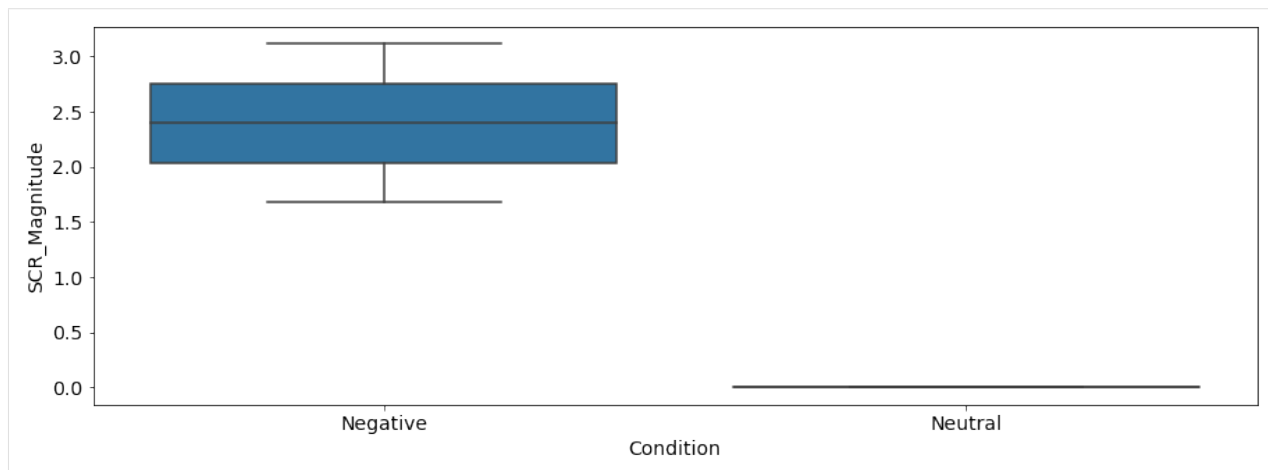
```




```
[12]: .           ="Condition"  ="RSP_Rate"      =
[12]: <matplotlib.axes._subplots.AxesSubplot at 0x1c6bd673c50>
```



```
[13]: .           ="Condition"  ="SCR_Magnitude"  =
[13]: <matplotlib.axes._subplots.AxesSubplot at 0x1c6a621a160>
```



Then interpret : As we can see, there seems to be a difference between the negative and the neutral pictures. Negative stimuli, as compared to neutral stimuli, were related to a stronger cardiac deceleration (i.e., higher heart rate variability), an accelerated breathing rate, and higher SCR magnitude.

5.10.7 Important remarks:

You can't break anything if you're on Binder, so have fun. Keep in mind that *this is for illustration purposes only*.

Data size limits on Github force us to downsample and have only one participant (sample rate would have to be >250 Hz, and you can't do stats with 4 observations in 1 subjects).

We invite you to read on reporting guidelines for biosignal measures. For ECG-PPG/HRV : [Quintana, Alvarez & Heathers, 2016 - GRAPH](#)

5.11 Interval-related Analysis

This example shows how to use Neurokit to analyze longer periods of data (i.e., greater than 10 seconds) such as **resting state data**. If you are looking to perform event-related analysis on epochs, you can refer to this [Neurokit example](#) here.

```
[1]: # Load NeuroKit and other useful packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

[2]: plt.rcParams['figure.figsize'] = 15, 9 # Bigger images
plt.rcParams['font.size'] = 13
```

5.11.1 The Dataset

First, download the dataset located on the GitHub repository.

It contains 5 minutes of physiological signals recorded at a frequency of 100Hz ($5 \times 60 \times 100 = 30000$ data points).

It contains the following signals : **ECG, PPG, RSP**

```
[3]: # Get data
url = "https://raw.githubusercontent.com/neuropsychology/NeuroKit/master/"
data_path = url + "data/bio_resting_5min_100hz.csv"
```

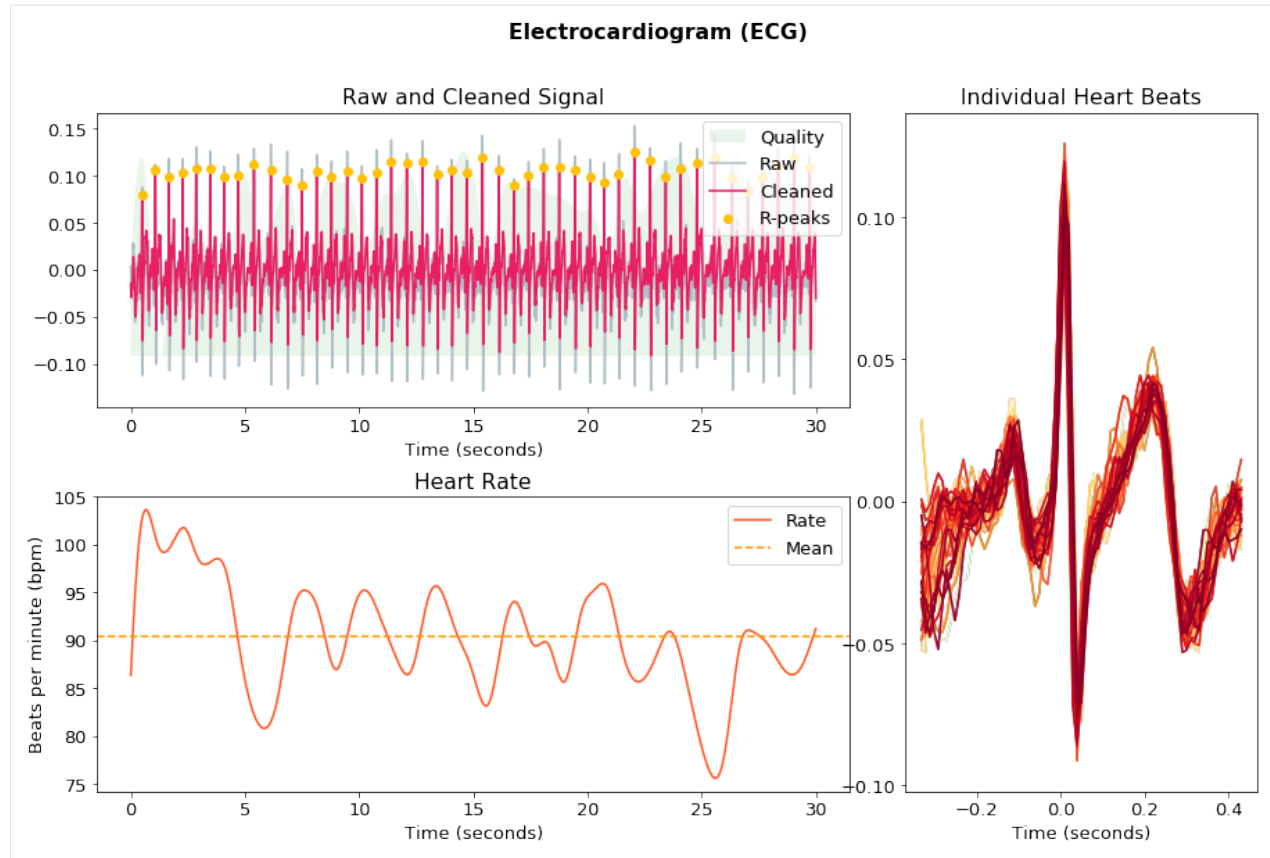
This is the resting state data from **1 participant** who was asked to close his/her eyes for 8 minutes, trying not to think of anything as well as not to fall asleep.

5.11.2 Process the Signals

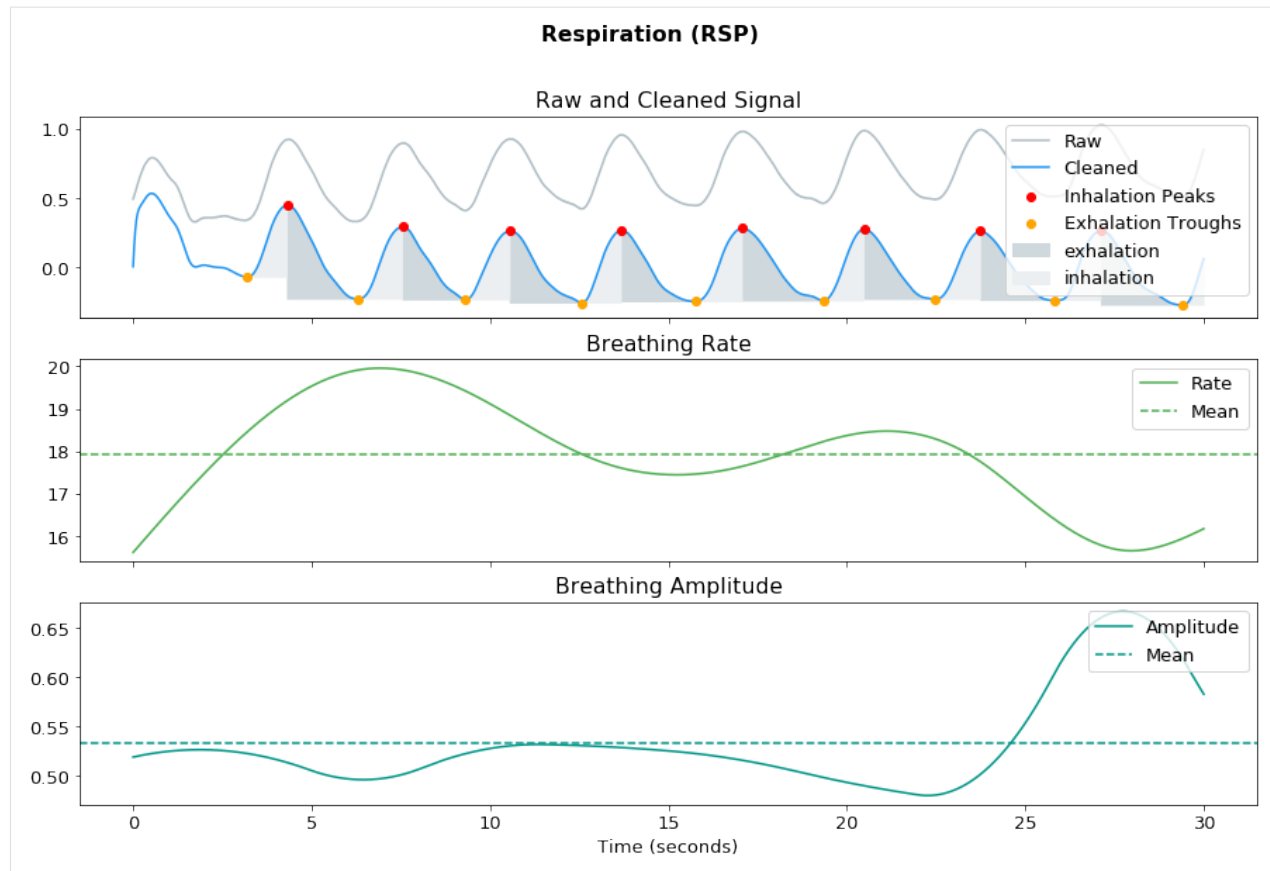
In this analysis here, we will focus on extracting **ECG** and **RSP** features. To process the respective physiological signals, you can use `ecg_process()` and `rsp_process()`. You can then visualize these signals using `ecg_plot()` and `rsp_plot()`. For the purposes of these example, we will select just 3000 datapoints (or 30s) to visualize.

Note: Do remember to specify the correct `sampling_rate` (in this case, to 100Hz) in which the signals were generated, in all the relevant functions.

```
[4]: # Process ecg
ecg = np.array(ecg_process(data_path, sampling_rate=100,
                           duration=3000, signal="ECG"))
```



```
[5]: # Process rsp
      = . "RSP" =100
      = . 3000 =100
```



5.11.3 Extract Features

Now that we have the processed signals, we can now perform the analysis using `ecg_intervalrelated()` and `rsp_intervalrelated()`. Simply provide the processed dataframe and these functions will return a dataframe of the features pertaining to the specific signal.

These features will be quite different from event-related features (See [Event-related analysis example](#)) as these signals were generated over a longer period of time. Hence, apart from the mean signal rate, variability metrics pertaining to heart rate variability (HRV) and respiratory rate variability (RRV) are also extracted here.

```
[6]: .
```

```
[6]:   ECG_Rate_Mean  HRV_RMSSD  HRV_MeanNN  HRV_SDNN  HRV_SDSD  HRV_CVNN  \
0      86.394304    3.883777    69.475638    4.903604    3.888256    0.07058

   HRV_CVSD  HRV_MedianNN  HRV_MadNN  HRV_MCVNN  ...  HRV_LFn  HRV_HFn  \
0    0.055901          69.0     4.4478    0.064461  ...    NaN    0.885628

   HRV_LnHF  HRV_SD1  HRV_SD2  HRV_SD2SD1  HRV_CSI  HRV_CVI  \
0    1.137604  2.749412  4.762122    1.732051    1.732051    2.32116

   HRV_CSI_Modified  HRV_SampEn
0          32.992946     1.978637

[1 rows x 30 columns]
```

```
[7]: .
[7]:   RSP_Rate_Mean  RSP_Amplitude_Mean  RRV_SDBB  RRV_RMSSD  RRV_SDSD  \
0      15.744233          0.397941  100.662988  119.691326  120.504248

   RRV_VLF  RRV_LF          RRV_HF  RRV_LFHF  RRV_LFn  RRV_HFn  RRV_SD1  \
0      NaN    NaN  793512.619536          NaN    NaN    NaN  85.209371

   RRV_SD2  RRV_SD2SD1  RRV_ApEn  RRV_SampEn  RRV_DFA_2
0  114.041384    1.338367  0.717675    1.504077    0.618535
```

5.11.4 Optional: Segmenting the Data

If you want to segment your data for analysis, such as analyzing two separate portions of your resting state data, you can simply do so by splitting the `ecg_signals` dataframe into **epochs** using `epochs_create()`. Using this example dataset, let's say you want to analyze the first half and the second half of the ECG data. This means that each halved data would last for $60 \times 2.5s = 150s$.

In this function, we would also specify the onset of events to be at the 0th (for the first half of the data) and the 15000th datapoint (for the second half of the data), since there are 30000 data points in total.

```
[8]: # Half the data
      = .
      = 0 15000 =100
      =0 =150
```

This returns a dictionary of 2 processed ECG dataframes, which you can then enter into `ecg_intervalrelated()`.

```
[9]: # Analyze
      .
[9]:   ECG_Rate_Mean  HRV_RMSSD  HRV_MeanNN  HRV_SDNN  HRV_SDSD  HRV_CVNN  \
1      86.377089    3.638450    69.497674    5.167181    3.645389    0.074350
2      86.411519    4.032578    69.460465    4.648090    4.042033    0.066917

   HRV_CVSD  HRV_MedianNN  HRV_MadNN  HRV_MCVNN  ...  HRV_LFn  HRV_HFn  \
1    0.052354          69.0    4.4478    0.064461  ...    NaN    NaN
2    0.058056          69.0    4.4478    0.064461  ...    NaN    NaN

   HRV_LnHF  HRV_SD1  HRV_SD2  HRV_SD2SD1  HRV_CSI  HRV_CVI  \
1      NaN    2.577680    4.464672    1.732051    1.732051    2.265138
2      NaN    2.858149    4.950459    1.732051    1.732051    2.354850

   HRV_CSI_Modified  HRV_SampEn
1      30.932155    1.252763
2      34.297785    1.881786

[2 rows x 30 columns]
```

This then returns a dataframe of the analyzed features, with the rows representing the respective segmented signals. Try doing this with your own signals!

5.12 Analyze Electrodermal Activity (EDA)

This example shows how to use NeuroKit2 to extract the features from Electrodermal Activity (EDA) .

```
[6]: # Load the NeuroKit package and other useful packages
import sys
import numpy as np
%matplotlib inline

[7]: plt.figure(figsize=(15, 5)) # Bigger images
```

5.12.1 Extract the cleaned EDA signal

In this example, we will use a simulated EDA signal. However, you can use any signal you have generated (for instance, extracted from the dataframe using `read_acqknowledge()`).

```
[9]: # Simulate 10 seconds of EDA Signal (recorded at 250 samples / second)
eda = nk.simulate_eda(duration=10, sampling_rate=250, noise_level=3, noise_std=0.01)
```

Once you have a raw EDA signal in the shape of a vector (i.e., a one-dimensional array), or a list, you can use `eda_process()` to process it.

```
[10]: # Process the raw EDA signal
eda_processed = nk.eda_process(eda)
```

Note: It is critical that you specify the correct sampling rate of your signal throughout many processing functions, as this allows NeuroKit to have a time reference.

This function outputs two elements, a *dataframe* containing the different signals (e.g., the raw signal, clean signal, SCR samples marking the different features etc.), and a *dictionary* containing information about the Skin Conductance Response (SCR) peaks (e.g., onsets, peak amplitude etc.).

5.12.2 Locate Skin Conductance Response (SCR) features

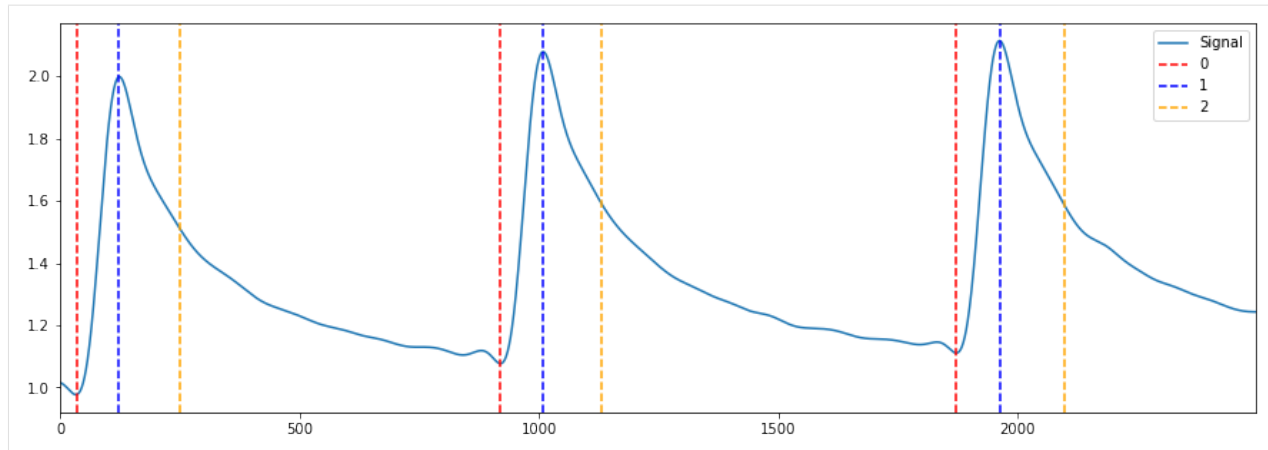
The processing function does two important things for our purpose: Firstly, it cleans the signal. Secondly, it detects the location of 1) peak onsets, 2) peak amplitude, and 3) half-recovery time. Let's extract these from the output.

```
[11]: # Extract clean EDA and SCR features
eda_clean, scr_features = nk.eda_process(eda)

eda_clean = eda_clean["EDA_Clean"]
scr_features = scr_features["SCR_Onsets"]
scr_features = scr_features["SCR_Peaks"]
scr_features = scr_features["SCR_Recovery"]
```

We can now visualize the location of the peak onsets, the peak amplitude, as well as the half-recovery time points in the cleaned EDA signal, respectively marked by the red dashed line, blue dashed line, and orange dashed line.

```
[12]: # Visualize SCR features in cleaned EDA signal
plt.plot(eda_clean, color='red', linestyle='dashed')
plt.plot(eda_clean, color='blue', linestyle='dashed')
plt.plot(eda_clean, color='orange', linestyle='dashed')
```



5.12.3 Decompose EDA into Phasic and Tonic components

We can also decompose the EDA signal into its phasic and tonic components, or more specifically, the **Phasic Skin Conductance Response (SCR)** and the **Tonic Skin Conductance Level (SCL)** respectively. The SCR represents the stimulus-dependent fast changing signal whereas the SCL is slow-changing and continuous. Separating these two signals helps to provide a more accurate estimation of the true SCR amplitude.

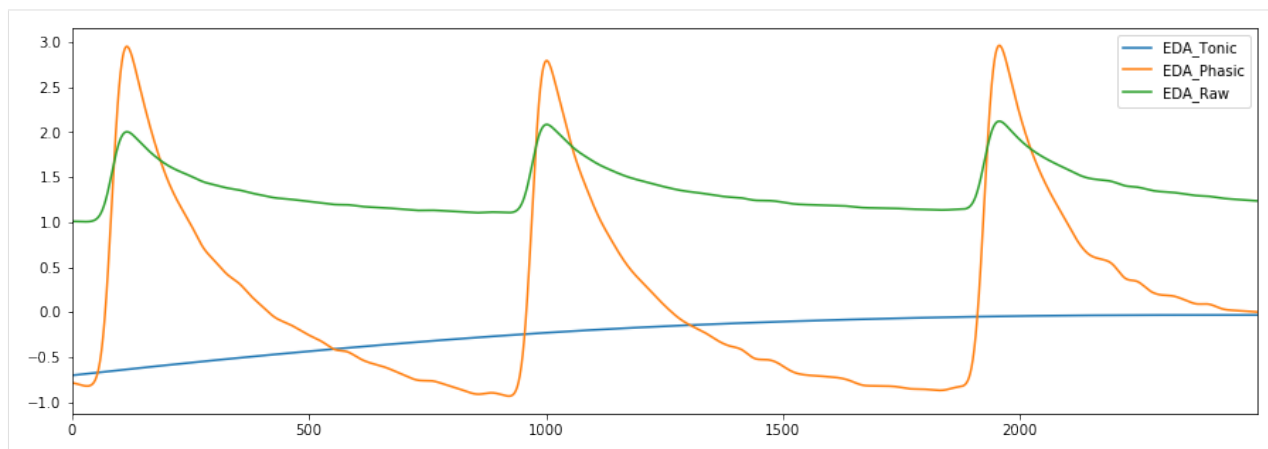
```
[13]: # Filter phasic and tonic components
      = . . =250
```

Note: here we standardized the raw EDA signal before the decomposition, which can be useful in the presence of high inter-individual variations.

We can now add the raw signal to the dataframe containing the two signals, and plot them!

```
[14]: "EDA_Raw" = # Add raw signal
      .
```

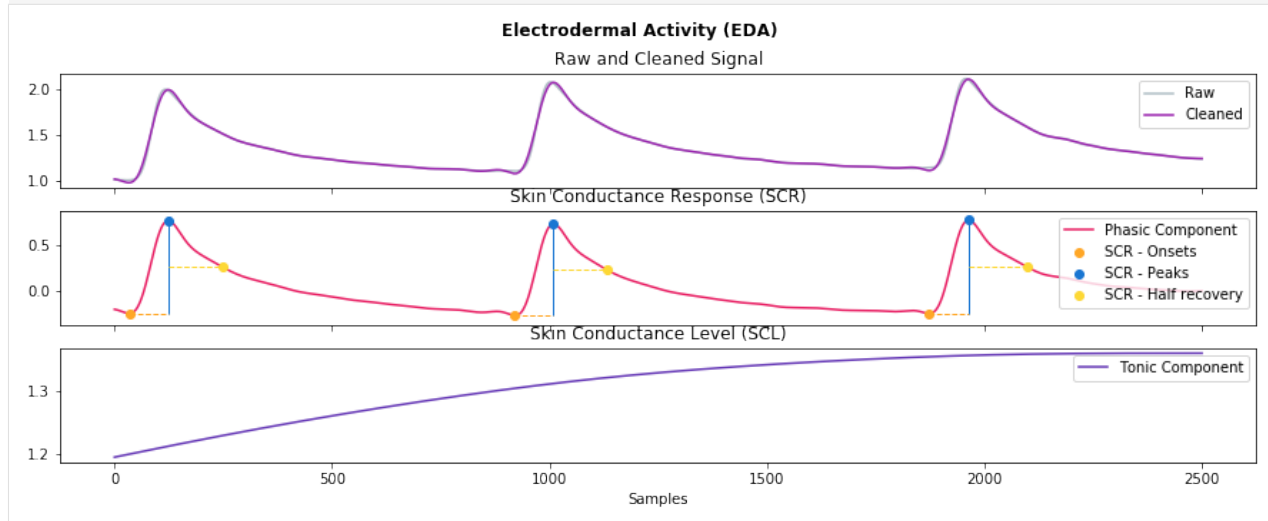
```
[14]: <matplotlib.axes._subplots.AxesSubplot at 0x29adaa05358>
```



5.12.4 Quick Plot

You can obtain all of these features by using the `eda_plot()` function on the dataframe of processed EDA.

```
[15]: # Plot EDA signal
```



5.13 Analyze Respiratory Rate Variability (RRV)

Respiratory Rate Variability (RRV), or variations in respiratory rhythm, are crucial indices of general health and respiratory complications. This example shows how to use NeuroKit to perform RRV analysis.

5.13.1 Download Data and Extract Relevant Signals

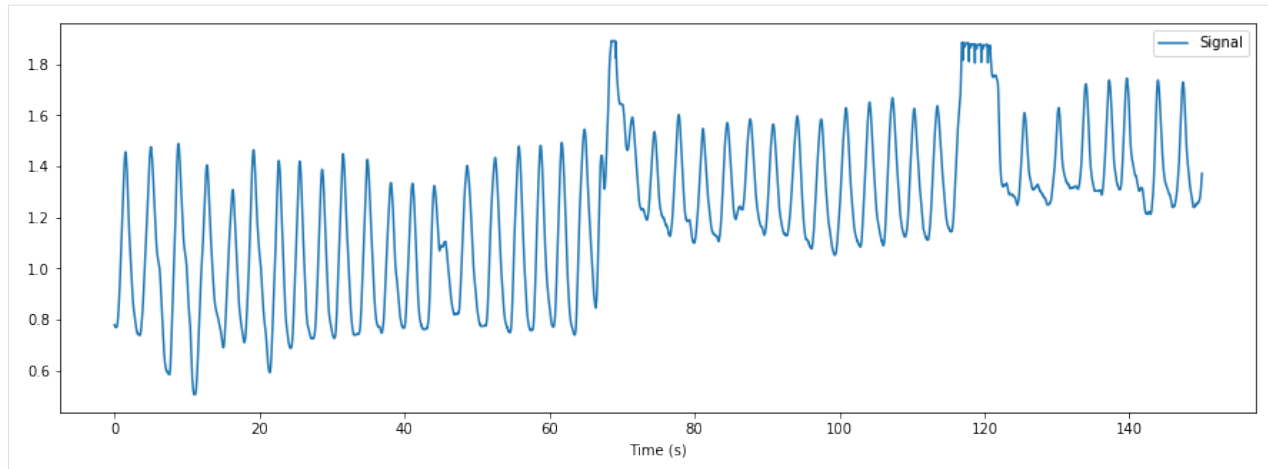
```
[1]: # Load NeuroKit and other useful packages
import sys
import os
import numpy as np
import matplotlib inline
```

```
[2]: # Set figure size
figure(figsize=(15, 5)) # Bigger images
```

In this example, we will download a dataset that contains electrocardiogram, respiratory, and electrodermal activity signals, and extract only the respiratory (RSP) signal.

```
[3]: # Get data
url = "https://raw.githubusercontent.com/neuropsychology/NeuroKit/master/"
data_path = os.path.join(url, "data/bio_eventrelated_100hz.csv")
RSP = np.loadtxt(data_path, delimiter=',', skiprows=1, dtype=float)

# Visualize
plt.plot(RSP[0:100])
```

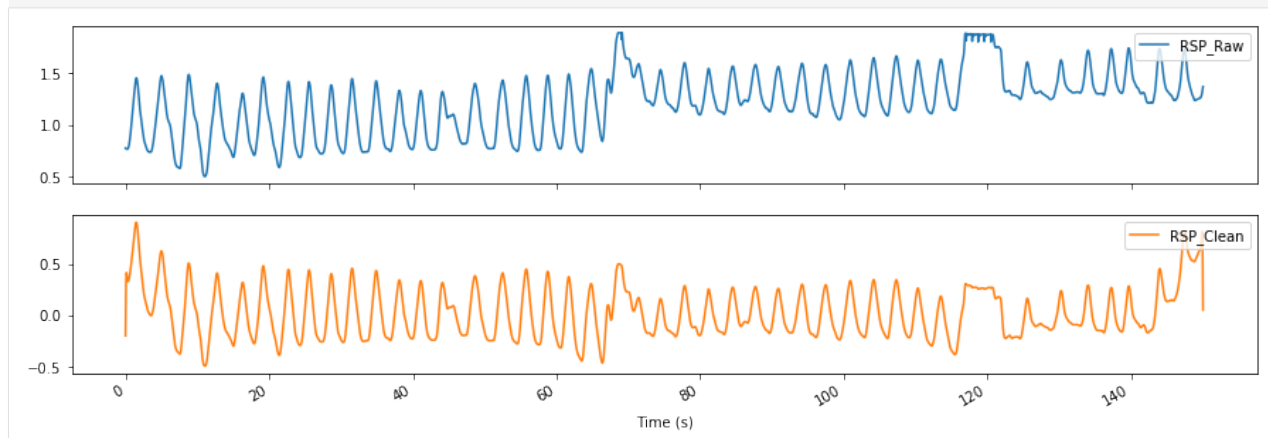
You now have the raw RSP signal in the shape of a vector (i.e., a one-dimensional array). You can then clean it using `rsp_clean()` and extract the inhalation peaks of the signal using `rsp_peaks()`. This will output 1) a *dataframe* indicating the occurrences of inhalation peaks and exhalation troughs (“1” marked in a list of zeros), and 2) a *dictionary* showing the samples of peaks and troughs.

Note: As the dataset has a frequency of 100Hz, make sure the ``sampling_rate`` is also set to 100Hz. It is critical that you specify the correct sampling rate of your signal throughout all the processing functions.

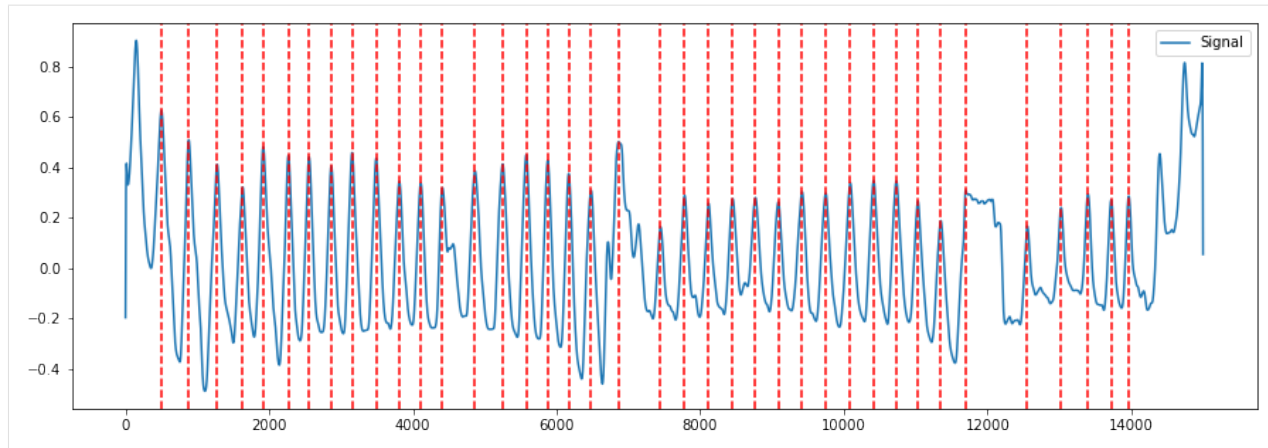
```
[4]: # Clean signal
      = . =100

      # Extract peaks
      = . =
      = . =
      ↪ "RSP_Peaks"
```

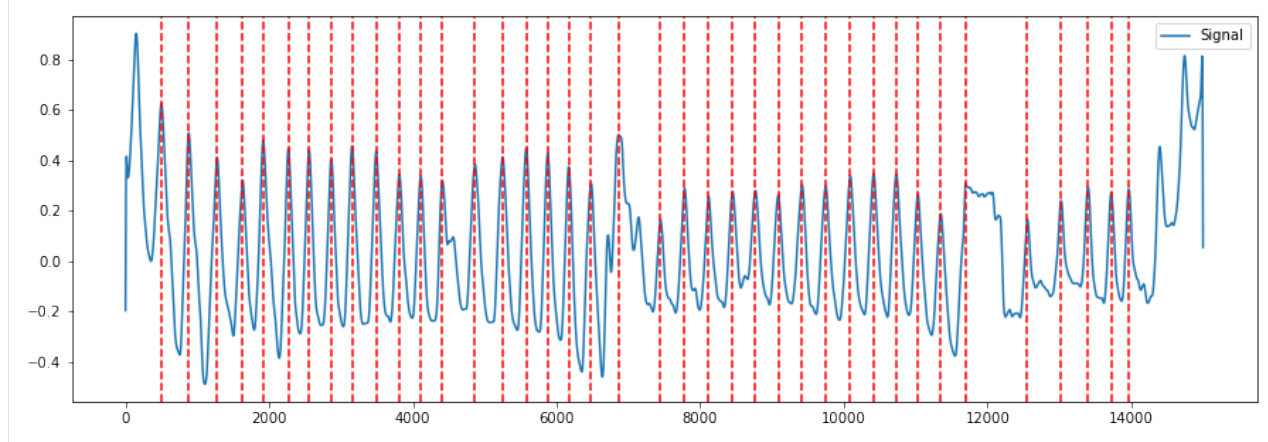
```
[5]: . "RSP_Raw" "RSP_Clean"
      ↪ =100 =True
```



```
[6]: = . 'RSP_Peaks'
```



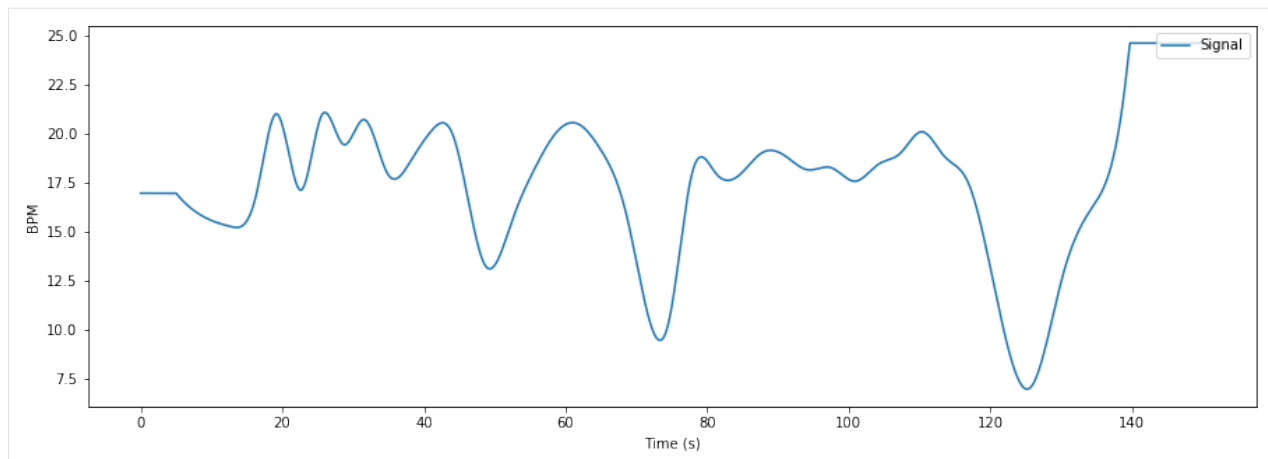
```
[7]: = . 'RSP_Peaks'
```



```
[8]: # Extract rate
      = . =None =100 # Note: You_
      ↳ can also replace info with peaks dictionary

# Visualize
      . =100
      . 'BPM'
```

```
[8]: Text(0, 0.5, 'BPM')
```



5.13.2 Analyse RRV

Now that we have extracted the respiratory rate signal and the peaks dictionary, you can then input these into `rsp_rrv()`. This outputs a variety of RRV indices including time domain, frequency domain, and nonlinear features. Examples of time domain features include RMSSD (root-mean-squared standard deviation) or SDBB (standard deviation of the breath-to-breath intervals). Power spectral analyses (e.g., LF, HF, LFHF) and entropy measures (e.g., sample entropy, SampEn where smaller values indicate that respiratory rate is regular and predictable) are also examples of frequency domain and nonlinear features respectively.

A Poincaré plot is also shown when setting `show=True`, plotting each breath-to-breath interval against the next successive one. It shows the distribution of successive respiratory rates.

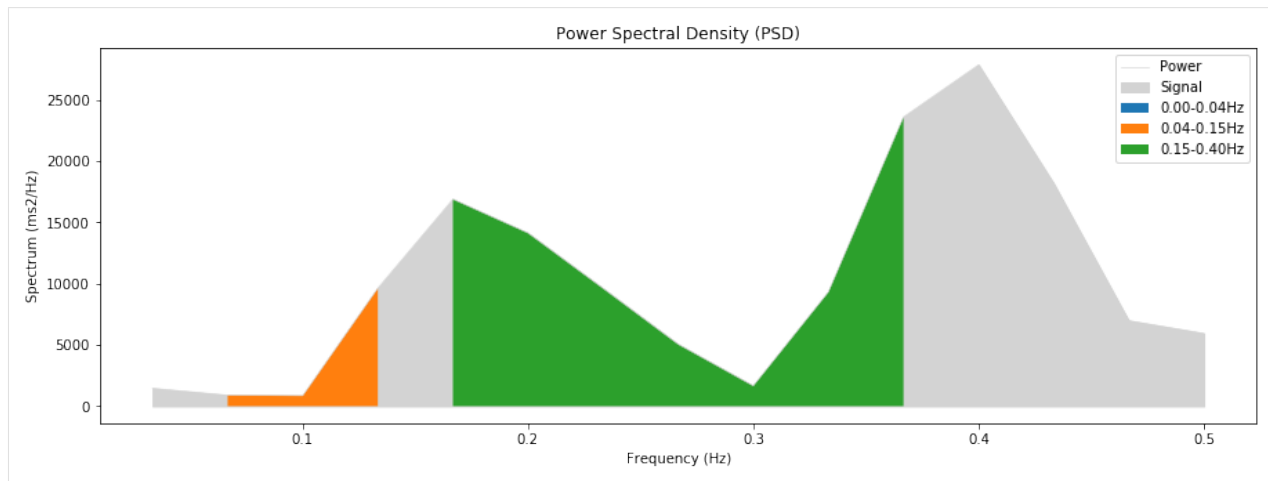
```
[9]: = . =100 =True
```

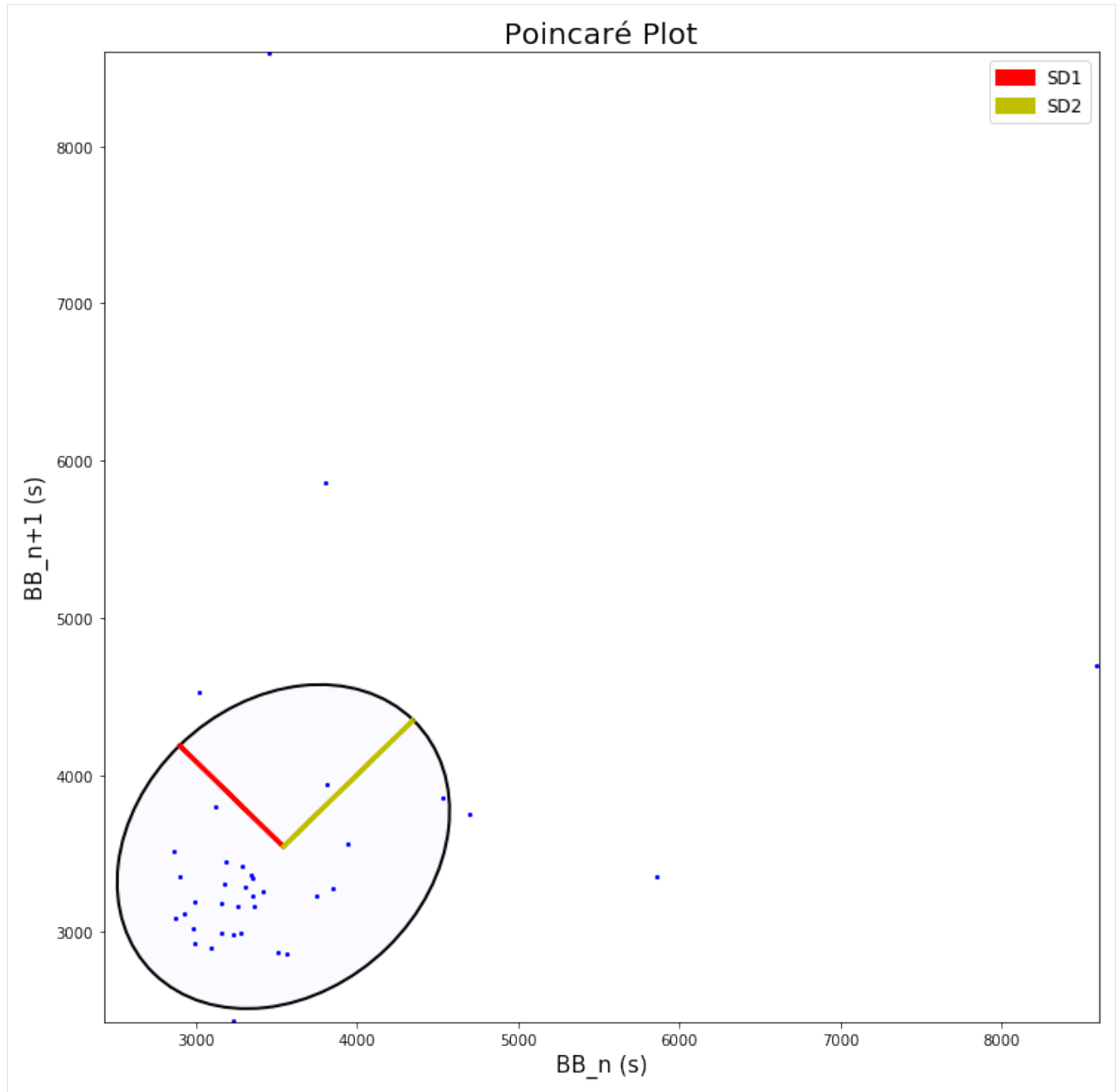
Neurokit warning: `signal_psd()`: The duration of recording is too short to support a sufficiently long window for high frequency resolution. Consider using a longer recording or increasing the `min_frequency`

```
[9]:      RRV_SDBB      RRV_RMSSD      RRV_SDSD      RRV_VLF      RRV_LF      RRV_HF  \
0  1030.411296  1269.625397  1286.590811      0.0  203.846501  2000.465169

      RRV_LFHF      RRV_LFn      RRV_HFn      RRV_SD1      RRV_SD2      RRV_SD2SD1      RRV_ApEn  \
0      0.1019      0.092476      0.907524      909.757087      1138.34833      1.251266      0.496939

      RRV_SampEn      RRV_DFA
0      0.693147      0.755783
```





This is a simple visualization tool for short-term (SD1) and long-term variability (SD2) in respiratory rhythm.

See documentation for full reference

RRV method taken from : Soni et al. 2019

5.14 ECG-Derived Respiration (EDR) Analysis

ECG-derived respiration (EDR) is the extraction of respiratory information from ECG and is a noninvasive method to monitor respiration activity under instances when respiratory signals are not recorded. In clinical settings, this presents convenience as it allows the monitoring of cardiac and respiratory signals simultaneously from a recorded ECG signal. This example shows how to use Neurokit to perform EDR analysis.

```
[1]: # Load NeuroKit and other useful packages
import sys
import os
import numpy as np
import matplotlib inline

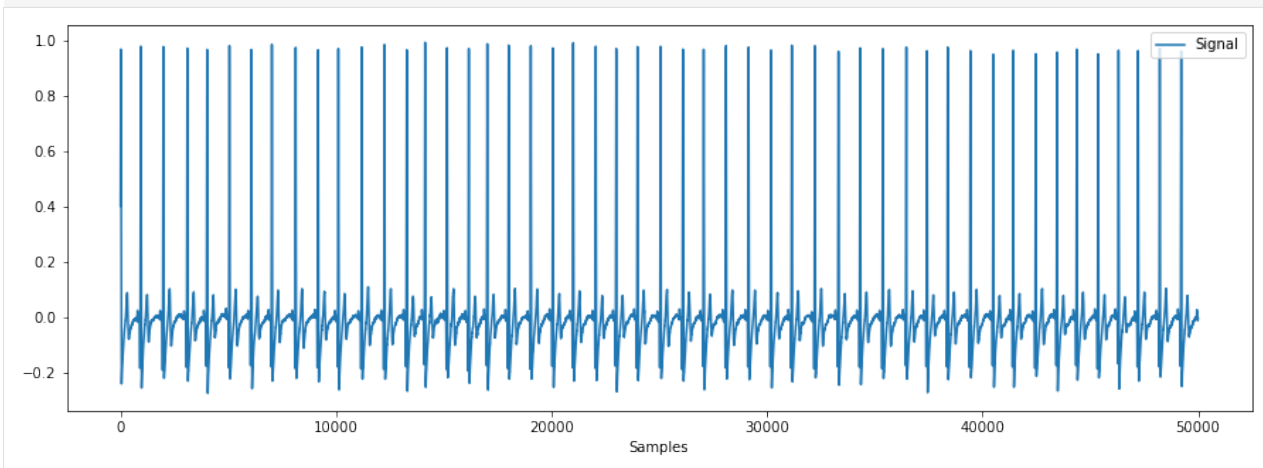
[2]: # Set figure size
plt.figure(figsize=(15, 5)) # Bigger images
```

5.14.1 Download ECG Data

In this example, we will download a dataset containing an ECG signal sampled at 1000 Hz.

```
[3]: # Get data
url = "https://raw.githubusercontent.com/neuropsychology/NeuroKit/dev/data/ecg_1000hz.csv"
data = np.loadtxt(url)

# Visualize signal
plt.plot(data)
```



5.14.2 Extraction of ECG Features

Now you can extract the R peaks of the signal using `ecg_peaks()` and compute the heart period using `ecg_rate()`.

Note: As the dataset has a frequency of 1000Hz, make sure the ``sampling_rate`` is also set to 1000Hz. It is critical that you specify the correct sampling rate of your signal throughout all the processing functions.

```
[4]: # Extract peaks
      = .
      =1000

# Compute rate
      = .
      =1000      =
```

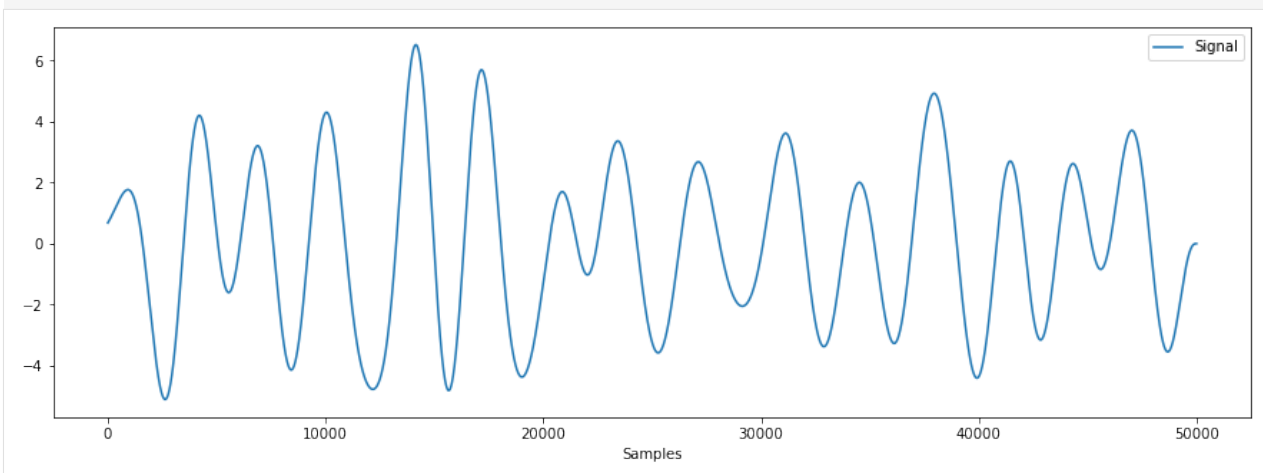
5.14.3 Analyse EDR

Now that we have an array of the heart period, we can then input this into `ecg_rsp()` to extract the EDR.

Default method is by Van Gent et al. 2019 ; see the full reference in documentation (run : `nk.ecg_rsp?`)

```
[5]: = .
      =1000

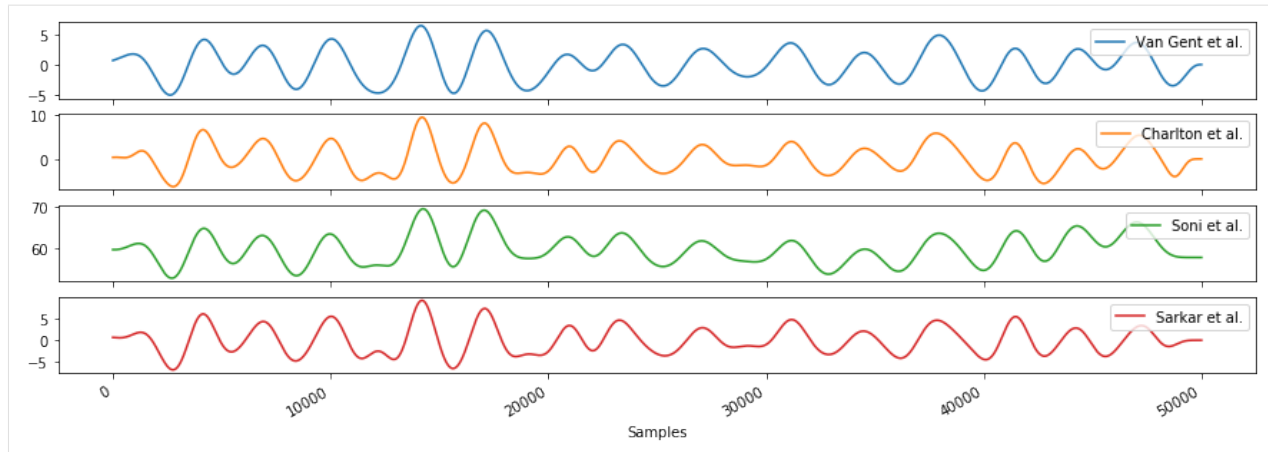
# Visual comparison
      .
```



The function `ecg_rsp()` incorporates different methods to compute EDR. For a visual comparison of the different methods, we can create a dataframe of EDR columns each of which are produced by different methods, and then plot it, like so:

```
[16]: = . "Van Gent et al." . =1000 =1000
      "Charlton et al." . =1000 =
      ↪ "charlton2016"
      "Soni et al." . =1000 = "soni2019"
      "Sarkar et al." . =1000 =
      ↪ "sarkar2015"
```

```
[17]: . =True
```



```
[ ]:
```

5.15 Extract and Visualize Individual Heartbeats

This example shows how to use NeuroKit to extract and visualize the QRS complexes (individual heartbeats) from an electrocardiogram (ECG).

```
[17]: # Load NeuroKit and other useful packages
import sys
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

[18]: plt.figure(figsize=(15, 9)) # Bigger images
```

5.15.1 Extract the cleaned ECG signal

In this example, we will use a simulated ECG signal. However, you can use any of your signal (for instance, extracted from the dataframe using the `read_acqknowledge()`).

```
[19]: # Simulate 30 seconds of ECG Signal (recorded at 250 samples / second)
signal = np.zeros((30, 250))
```

Once you have a raw ECG signal in the shape of a vector (i.e., a one-dimensional array), or a list, you can use `ecg_process()` to process it.

Note: It is critical that you specify the correct sampling rate of your signal throughout many processing functions, as this allows NeuroKit to have a time reference.

```
[20]: # Automatically process the (raw) ECG signal
processed_signal, info = ecg_process(signal, sampling_rate=250)
```

This function outputs two elements, a *dataframe* containing the different signals (raw, cleaned, etc.) and a *dictionary* containing various additional information (peaks location, ...).

5.15.2 Extract R-peaks location

The processing function does two important things for our purpose: 1) it cleans the signal and 2) it detects the location of the R-peaks. Let's extract these from the output.

```
[21]: # Extract clean ECG and R-peaks location
      =      "ECG_R_Peaks"
      =      "ECG_Clean"
```

Great. We can visualize the R-peaks location in the signal to make sure it got detected correctly by marking their location in the signal.

```
[22]: # Visualize R-peaks in ECG signal
      =      .

<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

Once that we know where the R-peaks are located, we can create windows of signal around them (of a length of for instance 1 second, ranging from 400 ms before the R-peak), which we can refer to as *epochs*.

5.15.3 Segment the signal around the heart beats

You can now epoch all these individual heart beats, synchronized by their R peaks with the `ecg_segment()` function.

```
[23]: # Plotting all the heart beats
      =      .                                =None                =250                =True
```

This create a dictionary of dataframes for each 'epoch' (in this case, each heart beat).

5.15.4 Advanced Plotting

This section is written for a more advanced purpose of plotting and visualizing all the heartbeats segments. The code below uses packages other than *NeuroKit2* to manually set the colour gradient of the signals and to create a more interactive experience for the user - by hovering your cursor over each signal, an annotation of the signal corresponding to the heart beat index is shown.

Custom colors and legend

Here, we define a function to create the epochs. It takes in `cleaned` as the cleaned signal dataframe, and `peaks` as the array of R-peaks locations.

```
[29]: %matplotlib notebook
      .      'figure(figsize' = 10 6      # resize

# Define a function to create epochs
def plot_heartbeats
    =      .                                =None                =-0.3
    =0.4      =      =
    return
    =      =      =250
    .
```



```
[29]:      Signal  Index Label      Time
      0 -0.188295   141     1 -0.300000
      1 -0.182860   142     1 -0.295977
      2 -0.177281   143     1 -0.291954
      3 -0.171530   144     1 -0.287931
      4 -0.165576   145     1 -0.283908
```

We then pivot the dataframe so that each column corresponds to the signal values of one channel, or *Label*.

```
[30]:      =      .      ='Time'      ='Label'      ='Signal'
```

```
[30]: Label      1      10      11      12      13      14  \
      Time
      -0.300000 -0.188295 -0.130393 -0.133485 -0.142082 -0.129604 -0.128826
      -0.295977 -0.182860 -0.129565 -0.132160 -0.141467 -0.128912 -0.128071
      -0.291954 -0.177281 -0.128556 -0.130639 -0.140687 -0.128097 -0.127219
      -0.287931 -0.171530 -0.127327 -0.128864 -0.139683 -0.127114 -0.126234
      -0.283908 -0.165576 -0.125839 -0.126767 -0.138381 -0.125906 -0.125059

      Label      15      16      17      18  ...      32      33  \
      Time
      -0.300000 -0.134212 -0.126986 -0.128488 -0.122148  ... -0.129433 -0.155517
      -0.295977 -0.132954 -0.125978 -0.127522 -0.121446  ... -0.128514 -0.154457
      -0.291954 -0.131499 -0.124859 -0.126416 -0.120663  ... -0.127415 -0.153226
      -0.287931 -0.129797 -0.123588 -0.125143 -0.119759  ... -0.126073 -0.151793
      -0.283908 -0.127774 -0.122108 -0.123664 -0.118693  ... -0.124416 -0.150128

      Label      34      35      4      5      6      7  \
      Time
      -0.300000 -0.097891 -0.186199 -0.115438 -0.133537 -0.136313 -0.137583
      -0.295977 -0.096876 -0.190515 -0.114778 -0.132642 -0.134961 -0.136883
      -0.291954 -0.095801 -0.194216 -0.113998 -0.131618 -0.133385 -0.136045
      -0.287931 -0.094629 -0.197353 -0.113073 -0.130441 -0.131522 -0.135024
      -0.283908 -0.093312 -0.199973 -0.111967 -0.129071 -0.129285 -0.133753

      Label      8      9
      Time
      -0.300000 -0.137964 -0.129776
      -0.295977 -0.137485 -0.128841
      -0.291954 -0.136902 -0.127814
      -0.287931 -0.136184 -0.126646
      -0.283908 -0.135299 -0.125278
```

[5 rows x 35 columns]

```
[31]: # Import dependencies
      import      as

      # Prepare figure
      =      .

      .      "Individual Heart Beats"
      .      "Time (seconds)"

      # Aesthetics
      =
      = 'Channel ' + for in      # Set labels for each signal
```

(continues on next page)

(continued from previous page)

```

    = . . . 0 1 "Label" . #
    ↪Get color map
    = # Create empty list to contain the plot of each signal

for . in . = '%s' % =

# Show figure

<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>

```

Interactivity

This section of the code incorporates the aesthetics and interactivity of the plot produced. Unfortunately, the interactivity is not active in this example but it should work in your console! As you hover your cursor over each signal, annotation of the channel that produced it is shown. The below figure that you see is a standstill image.

Note: you need to install the ``mplcursors`` package for the interactive part (``pip install mplcursors``)

```

[32]: # Import packages
import . as
from . import

import

[33]: # Obtain hover cursor
    . =True =True . "add" lambda .
    ↪ . .
# Return figure

<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>

{
  "cells": [
    { "cell_type": "markdown", "metadata": {}, "source": [
      "# Locate P, Q, S and T waves in ECG"
    ]
  }, {
    "cell_type": "markdown", "metadata": {}, "source": [
      "This example shows how to use Neurokit to delineate the ECG peaks in Python using NeuroKit. This means detecting and locating all components of the QRS complex, including P-peaks and T-peaks, as well their onsets and offsets from an ECG signal."
    ]
  }
]

```

```

}, {
    "cell_type": "code", "execution_count": 4, "metadata": {}, "outputs": [], "source": [
        "# Load NeuroKit and other useful packages", "import neurokit2 as nkn", "import
        numpy as npn", "import pandas as pdn", "import matplotlib.pyplot as pltn", "import
        seaborn as snsn", "%matplotlib inlinen", "plt.rcParams['figure.figsize'] = [8, 5] # Bigger
        images"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "In this example, we will use a short segment of ECG signal with sampling rate of 3000
        Hz. "
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "## Find the R peaks"
    ]
}, {
    "cell_type": "code", "execution_count": 5, "metadata": {}, "outputs": [], "source": [
        "# Retrieve ECG data from data folder (sampling rate= 1000 Hz)n", "ecg_signal =
        nk.data(dataset="ecg_3000hz")['ECG']n", "# Extract R-peaks locationsn", "_", rpeaks
        = nk.ecg_peaks(ecg_signal, sampling_rate=3000)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "The [ecg_peaks()](https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.
        ecg_peaks>) function will return a dictionary contains the samples at which R-peaks are
        located. "
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Let's visualize the R-peaks location in the signal to make sure it got detected correctly."
    ]
}, {
    "cell_type": "code", "execution_count": 6, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEExCAYAAAC+ipGRAAAABHNCS"
            "text/plain": [
                "<Figure size 576x360 with 1 Axes>"
            ]
        }
    ]
}

```



```

    "cell_type": "markdown", "metadata": {}, "source": [
        "First, let's take a look at the 'peak' method and its output."
    ]
}, {
    "cell_type": "code", "execution_count": 7, "metadata": {}, "outputs": [], "source": [
        "# Delineate the ECG signaln", "_", waves_peak = nk.ecg_delineate(ecg_signal,
        rpeaks, sampling_rate=3000)"
    ]
}, {
    "cell_type": "code", "execution_count": 8, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAAwAAAC+ipGAAAAABHNQ"
                "<Figure size 576x360 with 1 Axes>"
            },
            "metadata": {
                "needs_background": "light"
            },
            "output_type": "display_data"
        }, {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAAwAAAC+ipGAAAAABHNQ"
                "text/plain": [
                    "<Figure size 576x360 with 1 Axes>"
                ]
            },
            "metadata": {
                "needs_background": "light"
            },
            "output_type": "display_data"
        }
    ], "source": [
        "# Visualize the T-peaks, P-peaks, Q-peaks and S-peaksn",
        "plot = nk.events_plot([waves_peak['ECG_T_Peaks'], n",
        "waves_peak['ECG_P_Peaks'],n", "waves_peak['ECG_Q_Peaks'],n",
        "waves_peak['ECG_S_Peaks']], ecg_signaln", "n", "# Zooming into
        "the first 3 R-peaks, with focus on T-peaks, P-peaks, Q-peaks and S-
        "peaksn", "plot = nk.events_plot([waves_peak['ECG_T_Peaks'][:3], n",
        "waves_peak['ECG_P_Peaks'][:3],n", "waves_peak['ECG_Q_Peaks'][:3],n",
        "waves_peak['ECG_S_Peaks'][:3]], ecg_signal[:12500])n"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [

```

“Visually, the ‘peak’ method seems to have correctly identified the P-peaks, Q-peaks, S-peaks and T-peaks for this signal, at least, for the first few complexes. Well done, *peak!*”, “n”, “However, it can be quite tiring to be zooming in to each complex and inspect them one by one. To have a better overview of all complexes at once, you can make use of the *show* argument in `[ecg_delineate()]`(https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.ecg_delineate) as below.”

```

]
}, {
  "cell_type": "code", "execution_count": 9, "metadata": {}, "outputs": [
    {
      "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAEwAAACAYAAAC+ipGAAAAABHNQ" },
      "text/plain": [
        "<Figure size 576x360 with 1 Axes>"
      ]
    },
    { "metadata": {
      "needs_background": "light"
    }, "output_type": "display_data" }
  ], "source": [
    "# Delineate the ECG signal and visualizing all peaks of ECG complexes", "_",
    "waves_peak = nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, show=True, ",
    "show_type='peaks')",
  ]
}, {
  "cell_type": "markdown", "metadata": {}, "source": [
    "The ‘peak’ method is doing a glamorous job with identifying all the ECG peaks for ",
    "this piece of ECG signal.n", "n", "On top of the above peaks, the peak method also ",
    "identify the wave boundaries, namely the onset of P-peaks and offset of T-peaks. You ",
    "can vary the argument show_type to specify the information you would like plot.n",
    "n", "Let’s visualize them below:"
  ]
}, {
  "cell_type": "code", "execution_count": 10, "metadata": {}, "outputs": [
    {
      "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAEwAAACAYAAAC+ipGAAAAABHNQ" },
      "text/plain": [
        "<Figure size 576x360 with 1 Axes>"
      ]
    },
    { "metadata": {
      "needs_background": "light"
    }
  ]
}
```

```

        }, "output_type": "display_data"
    }
], "source": [
    "# Delineate the ECG signal and visualizing all P-peaks boundaries", "signal_peak,
    waves_peak = nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, show=True,
    show_type='bounds_P')"
]
}, {
    "cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAEwAAACAYAAAC+ipGAAAAABHNQ
            "text/plain": [
                "<Figure size 576x360 with 1 Axes>"
            ]
        }, "metadata": {
            "needs_background": "light"
        }, "output_type": "display_data"
    ]
}, "source": [
    "# Delineate the ECG signal and visualizing all T-peaks boundaries", "signal_peak,
    waves_peak = nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, show=True,
    show_type='bounds_T')"
]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Both the onsets of P-peaks and the offsets of T-peaks appears to have been cor-
        rectly identified here. This information will be used to delineate cardiac phases
        in [ecg_phase()](https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.
        ecg_phase>).n", "n", "Let's next take a look at the continuous wavelet method."
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "### Continuous Wavelet Method (CWT)"
    ]
}
], {
    "cell_type": "code", "execution_count": 12, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAEwAAACAYAAAC+ipGAAAAABHNQ
            "text/plain": [

```

```
        "<Figure size 576x360 with 1 Axes>"
    ]
    }, "metadata": {
        "needs_background": "light"
    }, "output_type": "display_data"
    }
], "source": [
    "# Delineate the ECG signaln", "signal_cwt, waves_cwt =",
    "nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method='cwt'",
    "show=True, show_type='all')]"
], {
    "cell_type": "markdown", "metadata": {}, "source": [
        "By specifying 'all' in the show_type argument, you can plot all delineated informa-",
        "tion output by the cwt method. However, it could be hard to evaluate the accuracy of",
        "the delineated information with everything plotted together. Let's tease them apart!"
    ]
}, {
    "cell_type": "code", "execution_count": 13, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAEwAAACAYAAAC+ipGAAAAABHN",
                "text/plain": [
                    "<Figure size 576x360 with 1 Axes>"
                ]
            }, "metadata": {
                "needs_background": "light"
            }, "output_type": "display_data"
        }
    ], "source": [
        "# Visualize P-peaks and T-peaks", "signal_cwt, waves_cwt =",
        "nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method='cwt'",
        "show=True, show_type='peaks')]"
    ], {
        "cell_type": "code", "execution_count": 15, "metadata": {}, "outputs": [
            {
                "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAwAAEwAAACAYAAAC+ipGAAAAABHN",
                    "text/plain": [
                        "<Figure size 576x360 with 1 Axes>"
                    ]
                },
```



```

        ]
        }, "metadata": {
            "needs_background": "light"
        }, "output_type": "display_data"
    }
], "source": [
    "# Visualize T-waves boundariesn", "signal_cwt, waves_cwt =",
    "nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method='cwt',",
    "show=True, show_type='bounds_T')",
]
}, {
    "cell_type": "code", "execution_count": 14, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEExCAYAAAC+ipGAAAAABHNC",
                "text/plain": [
                    "<Figure size 576x360 with 1 Axes>"
                ]
            }, "metadata": {
                "needs_background": "light"
            }, "output_type": "display_data"
        }
    ], "source": [
        "# Visualize P-waves boundariesn", "signal_cwt, waves_cwt =",
        "nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method='cwt',",
        "show=True, show_type='bounds_P')",
    ]
}, {
    "cell_type": "code", "execution_count": 15, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEExCAYAAAC+ipGAAAAABHNC",
                "text/plain": [
                    "<Figure size 576x360 with 1 Axes>"
                ]
            }, "metadata": {
                "needs_background": "light"
            }, "output_type": "display_data"
        }
    ], "source": [

```

```

        "# Visualize R-waves boundaries", "signal_cwt", waves_cwt =
        nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="cwt",
        show=True, show_type='bounds_R')"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Unlike the peak method, the continuous wavelet method does not identify the Q-peaks and S-peaks. However, it provides more information regarding the boundaries of the waves", "n", "Visually, except a few exception, CWT method is doing a great job. However, the P-waves boundaries are not very clearly identified here.", "n", "Last but not least, we will look at the third method in Neurokit [ecg_delineate()](https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.ecg\_delineate) function: the discrete wavelet method. "
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "#### Discrete Wavelet Method (DWT)"
    ]
}, {
    "cell_type": "code", "execution_count": 16, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEyCAYAAAC+ipGAAAAABHN...",
                    "text/plain": [
                        "<Figure size 576x360 with 1 Axes>"
                    ]
            }, "metadata": {
                "needs_background": "light"
            }, "output_type": "display_data"
        }
    ], "source": [
        "# Delineate the ECG signal", "signal_dwt", waves_dwt =
        nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="dwt",
        show=True, show_type='all')"
    ]
}, {
    "cell_type": "code", "execution_count": 17, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEyCAYAAAC+ipGAAAAABHN...",
                    "text/plain": [
                        "<Figure size 576x360 with 1 Axes>"
                    ]
            }, "metadata": {
                "needs_background": "light"
            }, "output_type": "display_data"
        }
    ], "source": [

```

```

        ]
        }, "metadata": {
            "needs_background": "light"
        }, "output_type": "display_data"
    }
], "source": [
    "# Visualize P-peaks and T-peaks", "signal_dwt", waves_dwt =
    nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="dwt",
    show=True, show_type='peaks')
]
}, {
    "cell_type": "code", "execution_count": 18, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEyCAYAAAC+ipGAAAAABHNQ
            "text/plain": [
                "<Figure size 576x360 with 1 Axes>"
            ]
        }, "metadata": {
            "needs_background": "light"
        }, "output_type": "display_data"
    ]
}, "source": [
    "# visualize T-wave boundaries", "signal_dwt", waves_dwt =
    nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="dwt",
    show=True, show_type='bounds_T')
]
}, {
    "cell_type": "code", "execution_count": 21, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEyCAYAAAC+ipGAAAAABHNQ
            "text/plain": [
                "<Figure size 576x360 with 1 Axes>"
            ]
        }, "metadata": {
            "needs_background": "light"
        }, "output_type": "display_data"
    ]
}, "source": [

```

```

        "# Visualize P-wave boundaries", "signal_dwt", waves_dwt =
        nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="dwt",
        show=True, show_type='bounds_P')
    ]
}, {
    "cell_type": "code", "execution_count": 19, "metadata": {}, "outputs": [
        {
            "data": { "image/png": "iVBORw0KGgoAAAANSUhEUgAAAewAAAEyCAYAAAC+ipGAAAAABHNQ
            "text/plain": [
                "<Figure size 576x360 with 1 Axes>"
            ]
        }, "metadata": {
            "needs_background": "light"
        }, "output_type": "display_data"
    ]
}, "source": [
    "# Visualize R-wave boundaries", "signal_dwt", waves_dwt =
    nk.ecg_delineate(ecg_signal, rpeaks, sampling_rate=3000, method="dwt",
    show=True, show_type='bounds_R')
]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "Visually, from the plots above, the delineated outputs of DWT appear to be more ac-
        curate than CWT, especially for the P-peaks and P-wave boundaries.n", "n", "Overall,
        for this signal, the peak and DWT methods seem to be superior to the CWT."
    ]
}
], "metadata": {
    "kernelspec": { "display_name": "Python 3", "language": "python", "name": "python3"
    }, "language_info": {
        "codemirror_mode": { "name": "ipython", "version": 3
        }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python", "nbcon-
        vert_exporter": "python", "pygments_lexer": "ipython3", "version": "3.7.3"
    }
}, "nbformat": 4, "nbformat_minor": 2
}

```

5.16 How to create epochs

So, in your experiment, participants undergo a number of trials (events) and these events are possibly of different conditions. And you are wondering how can you locate these events on your signals and perhaps make them into epochs for future analysis?

This example shows how to use Neurokit to extract epochs from data based on events localisation. In case you have multiple data files for each subject, this example also shows you how to create a loop through the subject folders and put the files together in an epoch format for further analysis.

```
[1]: # Load NeuroKit and other useful packages
import           as
import           as
%matplotlib inline

[10]: .           'figure.figsize' = 15 5    # Bigger images
```

In this example, we will use a short segment of data which has ECG, EDA and respiration (RSP) signals.

5.16.1 One signal with multiple event markings

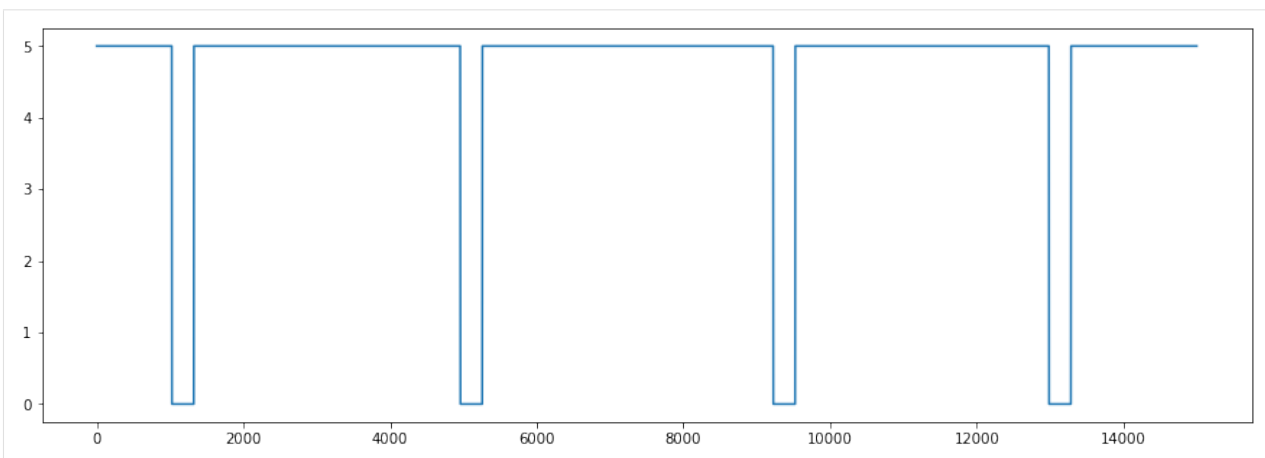
```
[11]: # Retrieve ECG data from data folder (sampling rate= 1000 Hz)
      = .           "bio_eventrelated_100hz"
```

Besides the signal channels, this data also has a fourth channel which consists of a string of 0 and 5. This is a binary marking of the Digital Input channel in BIOPAC.

Let's visualize the event-marking channel below.

```
[12]: # Visualize the event-marking channel
      .           'Photosensor'

[12]: [<matplotlib.lines.Line2D at 0x2109054b6d8>]
```



Depends on how you set up your experiment, the onset of the event can either be marked by signal going from 0 to 5 or vice versa. Specific to this data, the onsets of the events are marked where the signal in the event-marking channel goes from 5 to 0 and the offsets of the events are marked where the signal goes from 0 to 5.

As shown in the above figure, there are four times the signal going from 5 to 0, corresponding to the 4 events (4 trials) in this data.

There were 2 types (the condition) of images that were shown to the participant: “Negative” vs. “Neutral” in terms of emotion. Each condition had 2 trials. The following list is the condition order.

```
[13]: condition_order = "Negative" "Neutral" "Neutral" "Negative"
```

Before we can epoch the data, we have to locate the events and extract their related information. This can be done using Neurokit function `events_find()`.

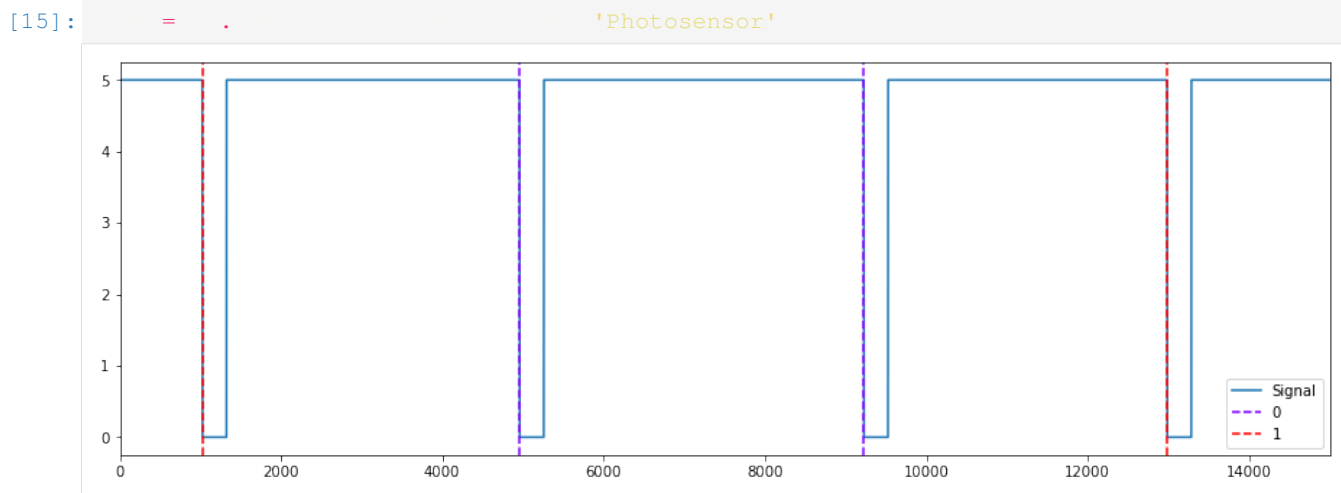
```
[14]: # Find events
      = .
      = "Photosensor"
      ='below'
      =

[14]: {'onset': array([ 1024,  4957,  9224, 12984]),
      'duration': array([300, 300, 300, 300]),
      'label': array(['1', '2', '3', '4'], dtype='<U11'),
      'condition': ['Negative', 'Neutral', 'Neutral', 'Negative']}
```

The output of `events_find()` gives you a dictionary that contains the information of event onsets, event duration, event label and event condition.

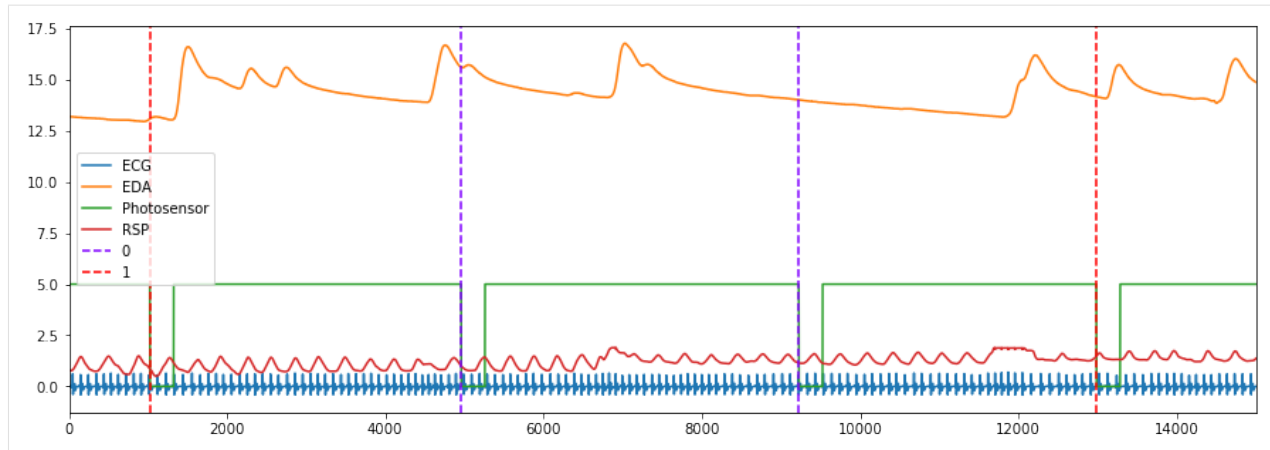
As stated, as the event onsets of this data are marked by event channel going from 5 to 0, the `threshold_keep` is set to below. Depends on your data, you can customize the arguments in `events_find()` to correctly locate the events.

You can use the `events_plot()` function to plot the events that have been found together with your event channel to confirm that it is correct.



Or you can visualize the events together with the all other signals.

```
[16]: = .
```



After you have located the events, you can now create epochs using the NeuroKit `epochs_create()` function. However, we recommend to process your signal first before cutting them to smaller epochs. You can read more about processing of physiological signals using NeuroKit in [Custom your Processing Pipeline Example](#).

```
[19]: # Process the signal
      = .
      = "ECG"
      = "RSP"
      = "EDA"
      =100
```

Now, let's think about how we want our epochs to be like. For this example, we want:

1. Epochs to start **1 second before the event onset**
2. Epochs to end **6 seconds** afterwards

These are passed into the `epochs_start` and `epochs_end` arguments, respectively.

Our epochs will then cover the region from **-1 s** to **+6 s** relative to the onsets of events (i.e., 700 data points since the signal is sampled at 100Hz).

```
[20]: # Build and plot epochs
      = .
      =100
      =-1
      =6
```

And as easy as that, you have created a dictionary of four dataframes, each correspond to an epoch of the event.

Here, in the above example, all your epochs have the same starting time and ending time, specified by `epochs_start` and `epochs_end`. Nevertheless, you can also pass a list of different timings to these two arguments to customize the duration of the epochs for each of your events.

5.16.2 One subject with multiple data files

In some experimental designs, instead of having one signal file with multiple events, each subject can have multiples files where each file is the record of one event.

In the following example, we will show you how to create a loop through the subject folders and put the files together in an epoch format for further analysis.

Firstly, let's say your data is arranged as the following where each subject has a folder and in each folder there are multiple data files corresponding to different events:

```
[Experiment folder]
├── Data
│   ├── Subject_001/
│   │   ├── event_1.[csv]
│   │   ├── event_2.[csv]
│   │   └── .....
│   └── Subject_002/
│       ├── event_1.[csv]
│       ├── event_2.[csv]
│       └── .....
└── analysis_script.py
```

The following will illustrate how your analysis script might look like. Try to re-create such data structure and the analysis script in your computer!

Now, in our analysis scripts, let's load the necessary packages:

```
[21]: # Load packages
import as
import
```

Assuming that your working directory is now at your analysis script, and you want to read all the data files of Subject_001.

Your analysis script should look something like below:

```
[ ]: # Your working directory should be at Experiment folder
      = 'Subject_001'

      =100

# List all data files in Subject_001 folder
      = . 'Data/' +

# Create an empty directory to store your files (events)
      =

# Loop through each file in the subject folder
for in
    # Read the file
    = . 'Data/' + + '/' +
    # Add a Label column (e.g Label 1 for epoch 1)
    'Label' = . +1
    # Set index of data to time in seconds
    = . /
    = .
    # Append the file into the dictionary
    + 1 =
```

And tah dah! You now should have a dictionary called `epochs` that resembles the output of `NeuroKit epochs_create()`. Each `DataFrame` in the `epochs` corresponds to an event (a trial) that Subject_001 underwent.

The `epochs` is now ready for further analysis!

5.17 Complexity Analysis of Physiological Signals

A **complex system**, can be loosely defined as one that comprises of many components that interact with each other. In science, this approach is used to investigate how relationships between a system's parts results in its collective behaviours and how the system interacts and forms relationships with its environment.

In recent years, there has been an increase in the use of complex systems to model **physiological systems**, such as for medical purposes where the dynamics of physiological systems can distinguish which systems are healthy and which are impaired. One prime example of how complexity exists in physiological systems is heart-rate variability (HRV), where higher complexity (greater HRV) has been shown to be an indicator of health, suggesting enhanced ability to adapt to environmental changes.

Although complexity is an important concept in health science, it is still foreign to many health scientists. This tutorial aims to provide a simple guide to the main tenets of complexity analysis in relation to physiological signals.

5.17.1 Basic Concepts

Definitions

Complex systems are examples of **nonlinear dynamical systems**.

A dynamical system can be described by a vector of numbers, called its **state**, which can be represented by a point in a phase space. In terms of physiology, this vector might include values of **variables** such as lung capacity, heart rate and blood pressure. This state aims to provide a complete description of the system (in this case, health) at some point in time.

The set of all possible states is called the **state space** and the way in which the system evolves over time (e.g., change in person's health state over time) can be referred to as **trajectory**.

After a sufficiently long time, different trajectories may evolve or converge to a common subset of state space called an **attractor**. The presence and behavior of attractors gives intuition about the underlying dynamical systems. This attractor can be a fixed-point where all trajectories converge to a single point (i.e., homeostatic equilibrium) or it can be periodic where trajectories follow a regular path (i.e., cyclic path).

Time-delay embedding

Nonlinear time-series analysis is used to understand, characterize and predict dynamical information about human physiological systems. This is based on the concept of **state-space reconstruction**, which allows one to reconstruct the full dynamics of a nonlinear system from a single time series (a signal).

One standard method for state-space reconstruction is *time-delay embedding* (or also known as delay-coordinate embedding). It aims to identify the state of a system at a particular point in time by searching the past history of observations for similar states, and by studying how these similar states evolve, in turn predict the future course of the time series.

In conclusion, the purpose of time-delay embeddings is to reconstruct the state and dynamics of an unknown dynamical system from measurements or observations of that system taken over time.

In this gif here, you can see how the phase space is constructed by plotting delayed signals against the original signal (where each time series is an embedded version i.e., delayed version of the original). Each point in the 3D reconstruction can be thought of as a time segment, with different points capturing different segments of history of variable X. Credits go to this short [illustration](#).

Embedding Parameters

For the reconstructed dynamics to be identical to the full dynamics of the system, some basic parameters need to be optimally determined for time-delayed embedding:

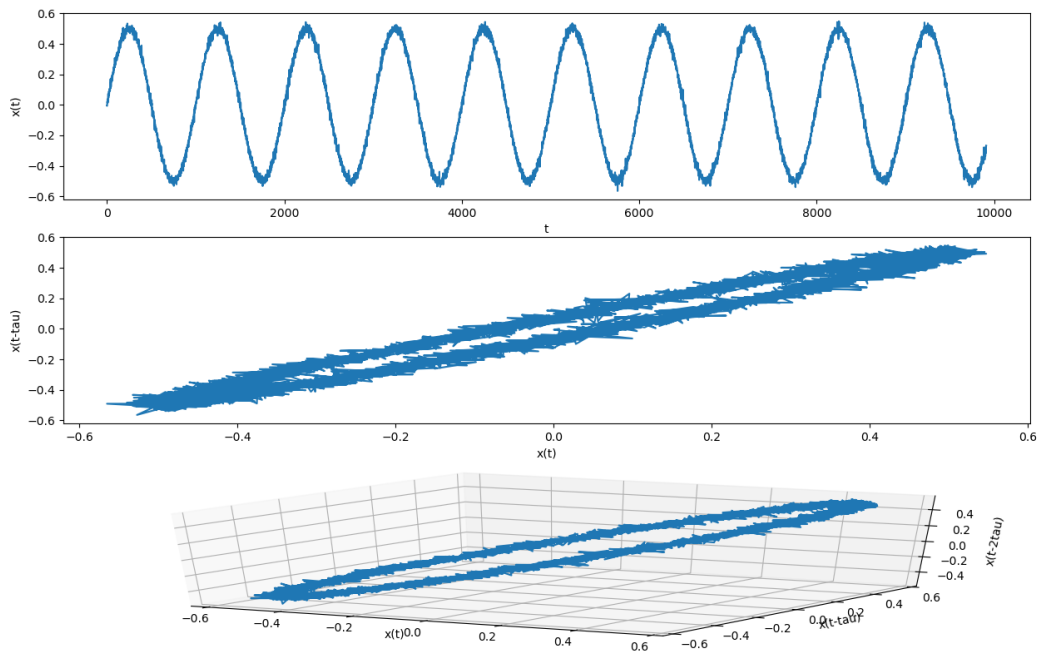
- **Time delay: τ ,**
 - A measure of time that sets basic delay
 - Generates the respective axes of the reconstruction: $x(t)$, $x(t-\tau)$, $x(t-2\tau)$...
 - E.g., if $\tau=1$, the state $x(t)$ would be plotted against its prior self $x(t-1)$
 - If τ is too small, constructed signals are too much alike and if too large, the reconstructed trajectory

will show connections between states very far in the past and to those far in the future (no relationship), which might make the reconstruction extremely complex

- **Embedding dimension, m**
 - Number of vectors to be compared (i.e. no. of additional signals of time delayed values of τ)
 - Dictates how many axes will be shown in the reconstruction space i.e. how much of the system's history is shown
 - Dimensionality must be sufficiently high to generate relevant information and create a rich history of states over time, but also low enough to be easily understandable
- **Tolerance threshold, r**
 - Tolerance for accepting similar patterns

Visualize Embedding

This is how a typical sinusoidal signal looks like, when embedded in 2D and 3D respectively.



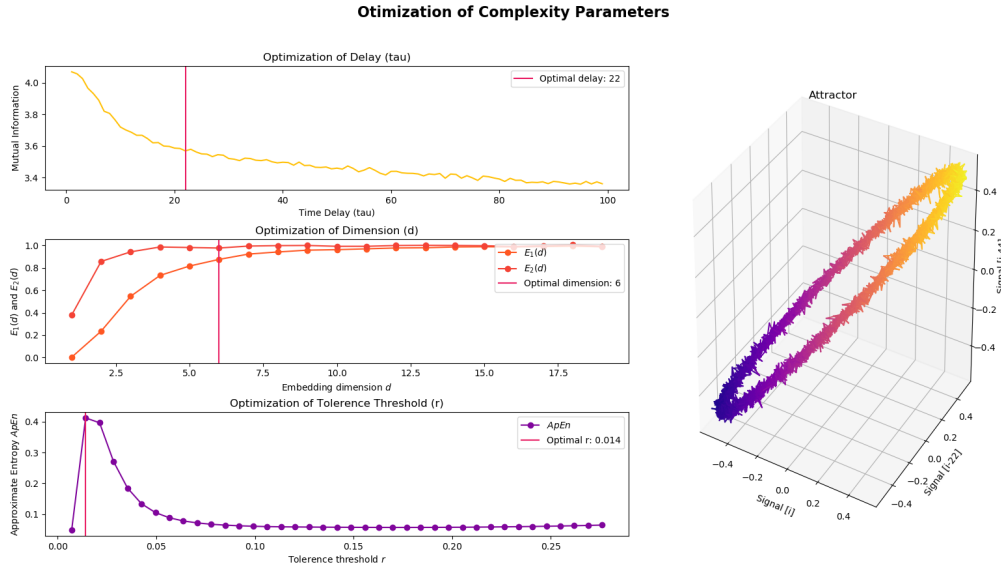
Using NeuroKit

There are different methods to guide the choice of parameters. In NeuroKit, you can use `nk.complexity_optimize()` to estimate the optimal parameters, including time delay, embedding dimension and tolerance threshold.

```
import neurokit2 as nk

signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01)
parameters = nk.complexity_optimize(signal, show=True)

parameters
>>> {'delay': 13, 'dimension': 6, 'r': 0.014}
```



In the above example, the optimal time delay is estimated using the *Mutual Information* method (Fraser & Swinney, 1986), the optimal embedding dimension is estimated using the *Average False Nearest Neighbour* (Cao, 1997) and the optimal r is obtained using the *Maximum Approximate Entropy* (Lu et al., 2008). These are the default methods in `nk.complexity_optimize()`. Nevertheless, you can specify your preferred method via the `method` arguments.

More of these methods can be read about in this [chapter](#) here.

5.17.2 Entropy as measures of Complexity

The complexity of physiological signals can be represented by the entropy of these non-linear, dynamic physiological systems.

Entropy can be defined as the measure of *disorder* in a signal.

Shannon Entropy (ShEn)

- call `nk.entropy_shannon()`

Approximate Entropy (ApEn)

- Quantifies the amount of regularity and the unpredictability of fluctuations over time-series data.
- Advantages of ApEn: lower computational demand (can be designed to work for small data samples i.e. less than 50 data points and can be applied in real time) and less sensitive to noise.
- Smaller values indicate that the data is more regular and predictable, and larger values corresponding to more complexity or irregularity in the data.
- call `nk.entropy_approximate()`

Examples of use

Reference	Signal	Parameters	Findings
Caldirola et al. (2004)	17min breath-by-breath recordings of respiration parameters	m=1, r=0.2	Panic disorder patients showed higher ApEn indexes in baseline RSP patterns (all parameters) than healthy subjects
Burioka et al. (2003)	30 mins of Respiration, 20s recordings of EEG	m=2, r=0.2, =1.1s for respiration, 0.09s for EEG	Lower ApEn of respiratory movement and EEG in stage IV sleep than other stages of consciousness
Boettger et al. (2009)	64s recordings of QT and RR intervals	m=2, r=0.2	Higher ratio of ApEn(QT) to ApEn(RR) for higher intensities of exercise, reflecting sympathetic activity
Taghavi et al. (2011)	2mis recordings of EEG	m=2, r=0.1	Higher ApEn of normal subjects than schizophrenic patients particularly in limbic areas of the brain

Sample Entropy (SampEn)

- A modification of approximate entropy
- Advantages over ApEn: data length independence and a relatively trouble-free implementation.
- Large values indicate high complexity whereas smaller values characterize more self-similar and regular signals.
- call `nk.entropy_sample()`

Examples of use

Reference	Signal	Parameters	Findings
Lake et al. (2002)	25min recordings of RR intervals	m=3, r=0.2	SampEn is lower in the course of neonatal sepsis and sepsislike illness
Lake et al. (2011)	24h recordings of RR intervals	m=1, r=to vary	In patients over 40 years old, SampEn has high degrees of accuracy in distinguishing atrial fibrillation from normal sinus rhythm in 12-beat calculations performed hourly
Estrada et al. (2015)	EMG diaphragm signal	m=1, r=0.3	fSampEn (fixed SampEn) method to extract RSP rate from respiratory EMG signal
Kapidzic et al. (2014)	RR intervals and its corresponding RSP signal	m=2, r=0.2	During paced breathing, significant reduction of SampEn(Resp) and SampEn(RR) with age in male subjects, compared to smaller and non-significant SampEn decrease in females
Abásolo et al. (2006)	5min recordings of EEG in 5 second epochs	m=1, r=0.25	Alzheimer's Disease patients had lower SampEn than controls in parietal and occipital regions

Fuzzy Entropy (FuzzyEn)

- Similar to ApEn and SampEn
- call `nk.entropy_fuzzy()`

Multiscale Entropy (MSE)

- Expresses different levels of either ApEn or SampEn by means of multiple factors for generating multiple time series
- Captures more useful information than using a scalar value produced by ApEn and SampEn
- call `nk.entropy_multiscale()`

5.17.3 Detrended Fluctuation Analysis (DFA)

5.18 Analyze Electrooculography EOG data (eye blinks, saccades, etc.)

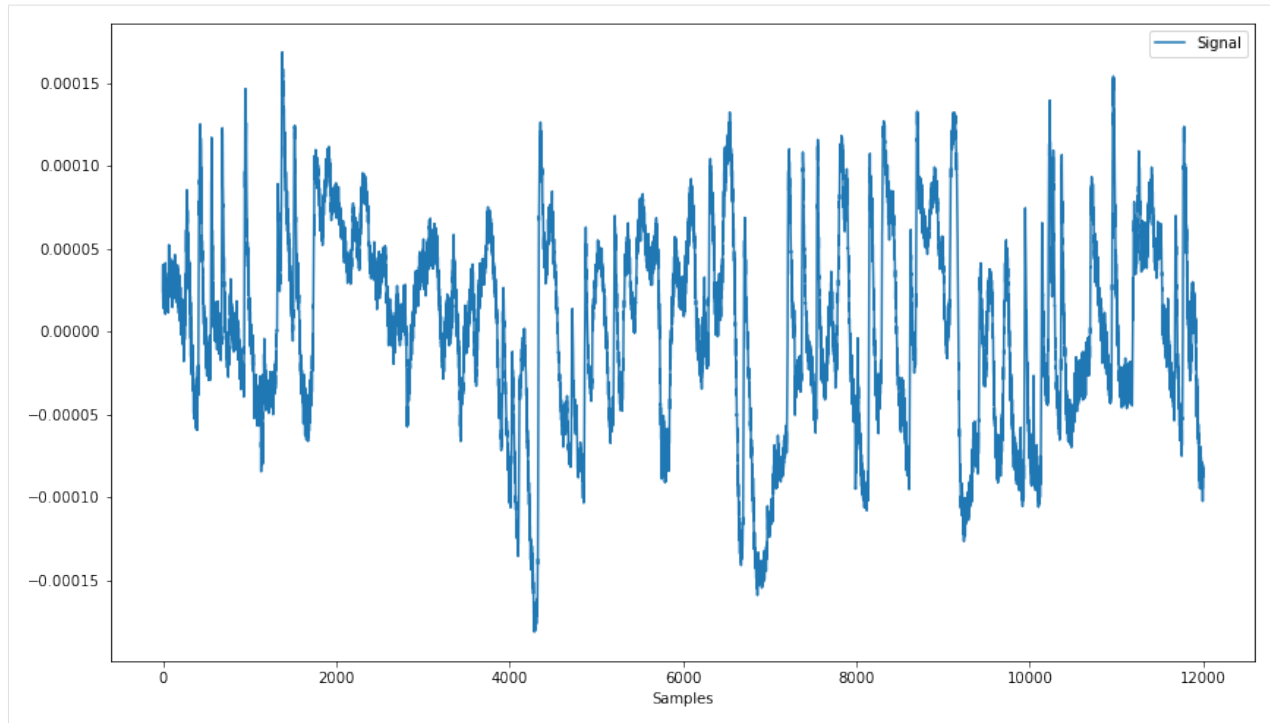
```
[26]: # This is temporary to load the dev version of NK, needs to be removed once it's in_
↳master
import
.      'D:/Dropbox/RECHERCHE/N/NeuroKit/'
# -----
import      as
import      as
import      as
import      as

.      'figure.figsize' = 14 8    # Increase plot size
```

5.18.1 Explore the EOG signal

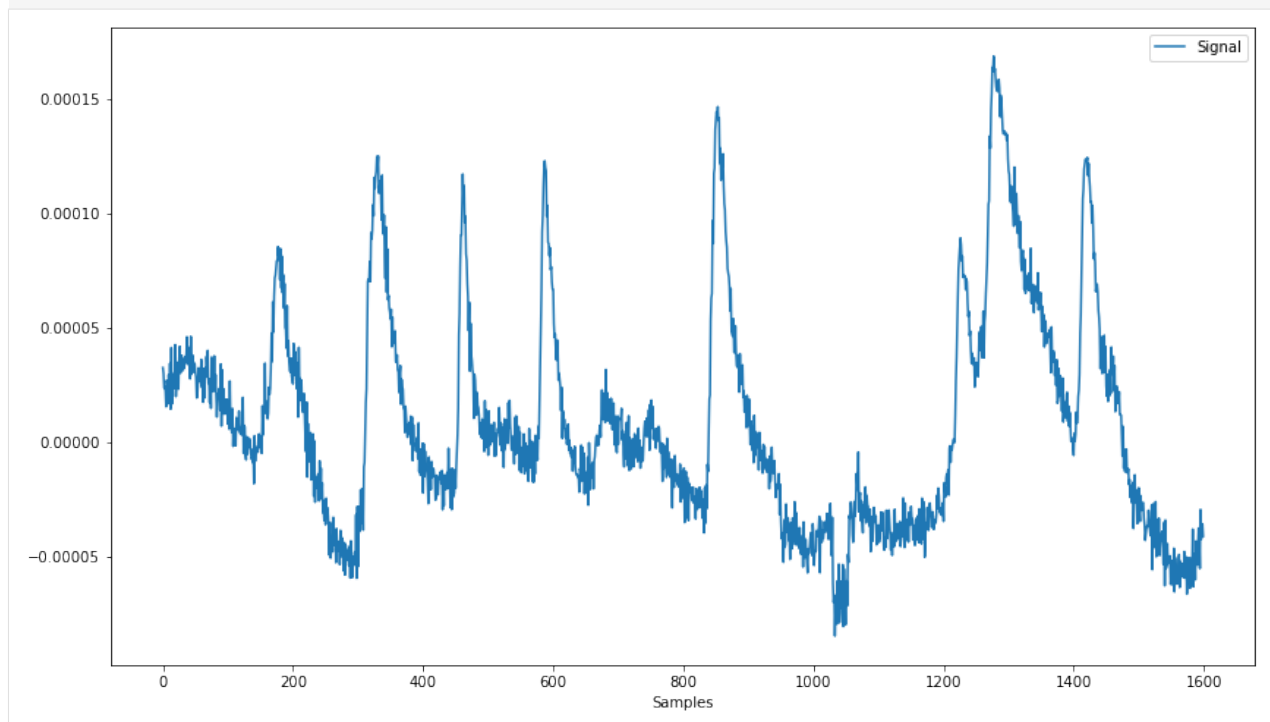
Let's load the example dataset corresponding to a vertical EOG signal.

```
[47]:      = .      "data/eog_100hz.csv"
      = .      # Extract the only column as a vector
      .
```



Let's zoom in to some areas where clear blinks are present.

```
[48]: . 100 1700
```



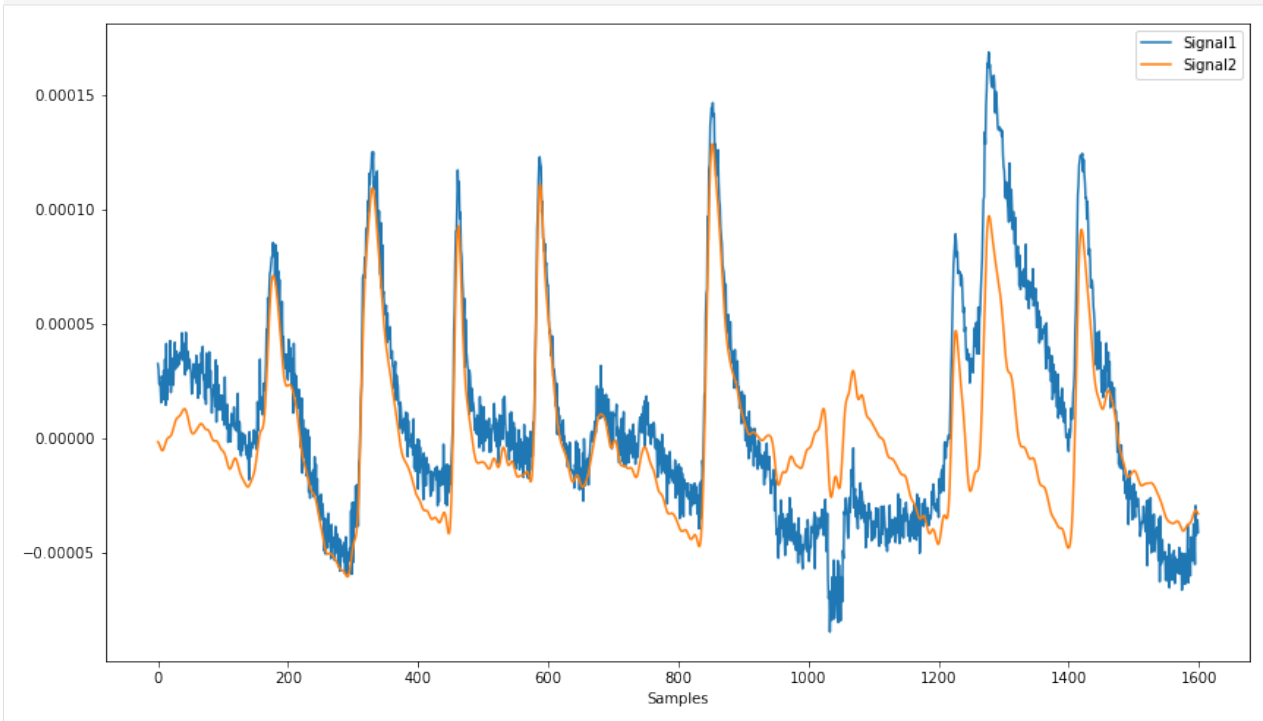
5.18.2 Clean the signal

We can now filter the signal to remove some noise and trends.

```
[49]: cleaned_signal = neurokit2.filters.clean_signal(signal, order=100, method='neurokit')
```

Let's visualize the same chunk and compare the clean version with the original signal.

```
[50]: plt.figure(figsize=(10, 5))
plt.plot(signal[100:1700], label='Signal1')
plt.plot(cleaned_signal[100:1700], label='Signal2')
```



5.18.3 Detect and visualize eye blinks

We will now run a peak detection algorithm to detect peaks location.

```
[54]: peaks = neurokit2.detection.detect_blinks(signal, order=100, method='mne')
```

```
[54]: array([[ 277,   430,   562,   688,   952,  1378,  1520,  1752,  3353,
          3783,  3928,  4031,  4168,  4350,  4723,  4878,  5213,  5365,
          5699,  5904,  6312,  6539,  6714,  7224,  7382,  7553,  7827,
          8014,  8154,  8312,  8626,  8702,  9140,  9425,  9741,  9948,
          10142, 10230, 10368, 10708, 10965, 11256, 11683, 11775],
          dtype=int64)
```

```
[77]: plt.figure(figsize=(10, 5))
plt.plot(signal[100:1700], label='Signal1')
plt.plot(cleaned_signal[100:1700], label='Signal2')
```

```
[79]: peaks = peaks.reshape((-1, 2))
peaks = peaks * 1000
# Convert to 2D array
# Rescale so that all the blinks are on the same scale
# Plot with their median (used here as a robust average)
```

(continues on next page)

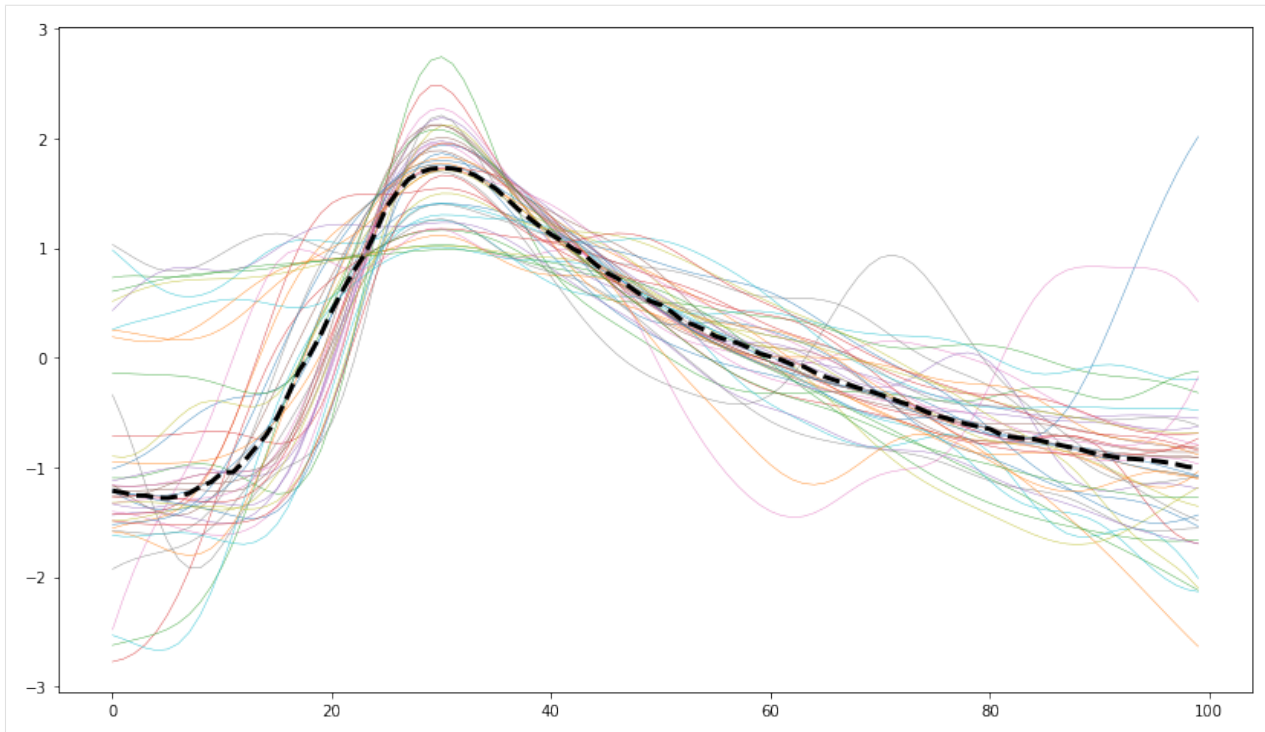
(continued from previous page)

```

.          =0.4
.          =1          =3          ='--'          ="black"

```

```
[79]: [matplotlib.lines.Line2D at 0x18fc51f7e50]
```



```
[ ]:
```

5.19 Fit a function to a signal

This small tutorial will show you how to use Python to estimate the best fitting line to some data. This can be used to find the optimal line passing through a signal.

```

[4]: # Load packages
import      as
import      as
import
import      as
import      as

.          'figure.figsize' = 14 8 # Increase plot size

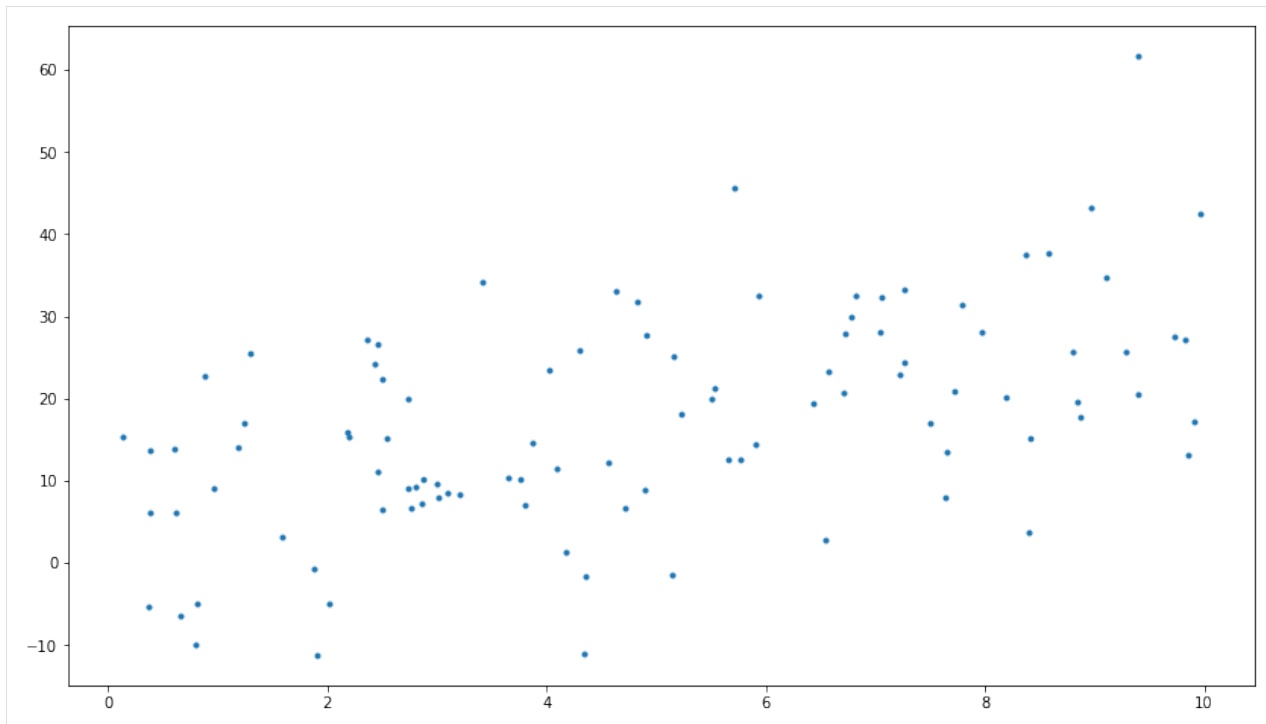
```

5.19.1 Fit a linear function

We will start by generating some random data on a scale from 0 to 10 (the x-axis), and then pass them through a function to create its y values.

```
[7]: = . . 0. 10. =100
     = 3. * + 2. + . . 0. 10. 100
     . . . . .
```

```
[7]: [<matplotlib.lines.Line2D at 0x2137926a4f0>]
```



Now in this case we **know** that the best fitting line will be a linear function (i.e., a straight line), and we want to find its parameters. A linear function has two parameters, the **intercept** and the **slope**.

First, we need to create this function, that takes some x values, the parameters, and return the y value.

```
[9]: def function_linear
     = + *
     return
```

Now, using the power of **scipy**, we can optimize this function based on our data to find the parameters that minimize the least square error. It just needs the function and the data's x and y values.

```
[16]: = . .
```

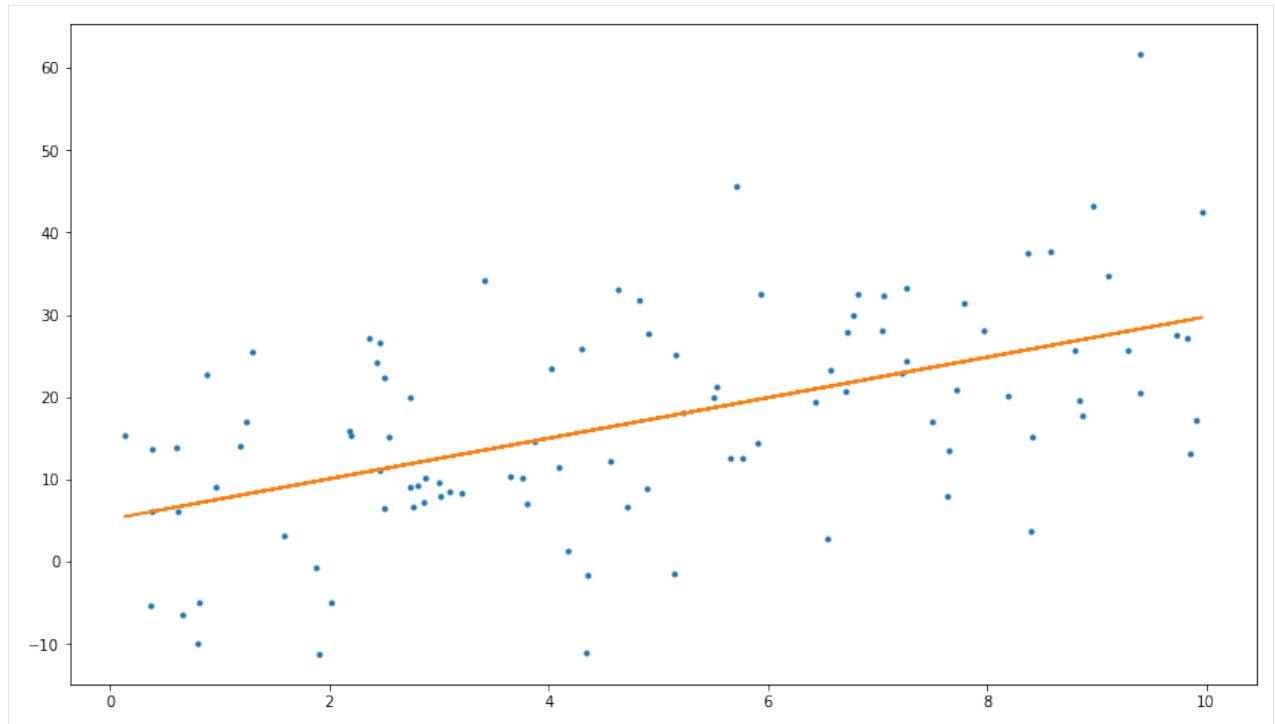
```
[16]: array([5.16622964, 2.46102004])
```

So the optimal parameters (in our case, the **intercept** and the **slope**) are returned in the **params** object. We can unpack these parameters (using the star symbol *****) into our linear function to use them, and create the predicted y values to our x axis.

```
[17]: = *
```

(continues on next page)

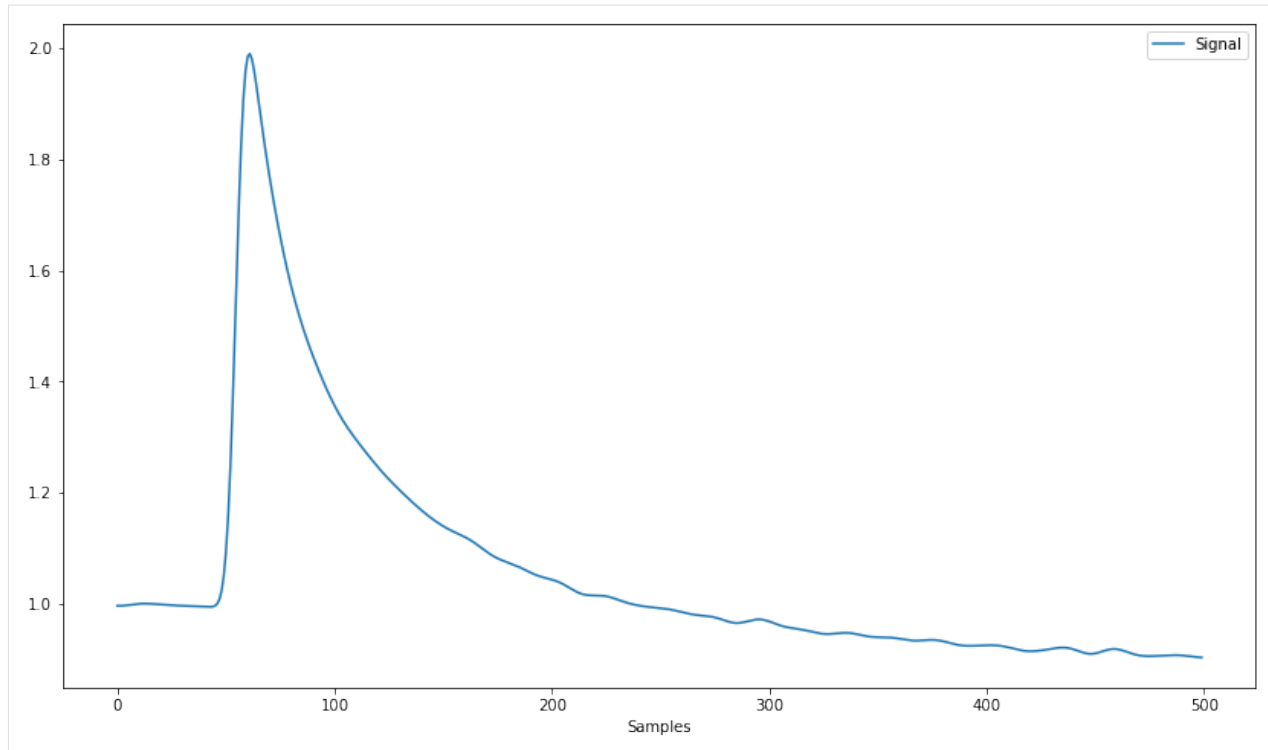
```
[17]: [matplotlib.lines.Line2D at 0x21379293370]
```



5.19.2 Non-linear curves

We can also use that to approximate non-linear curves.

[22]: $\frac{1}{2} = 0.5$



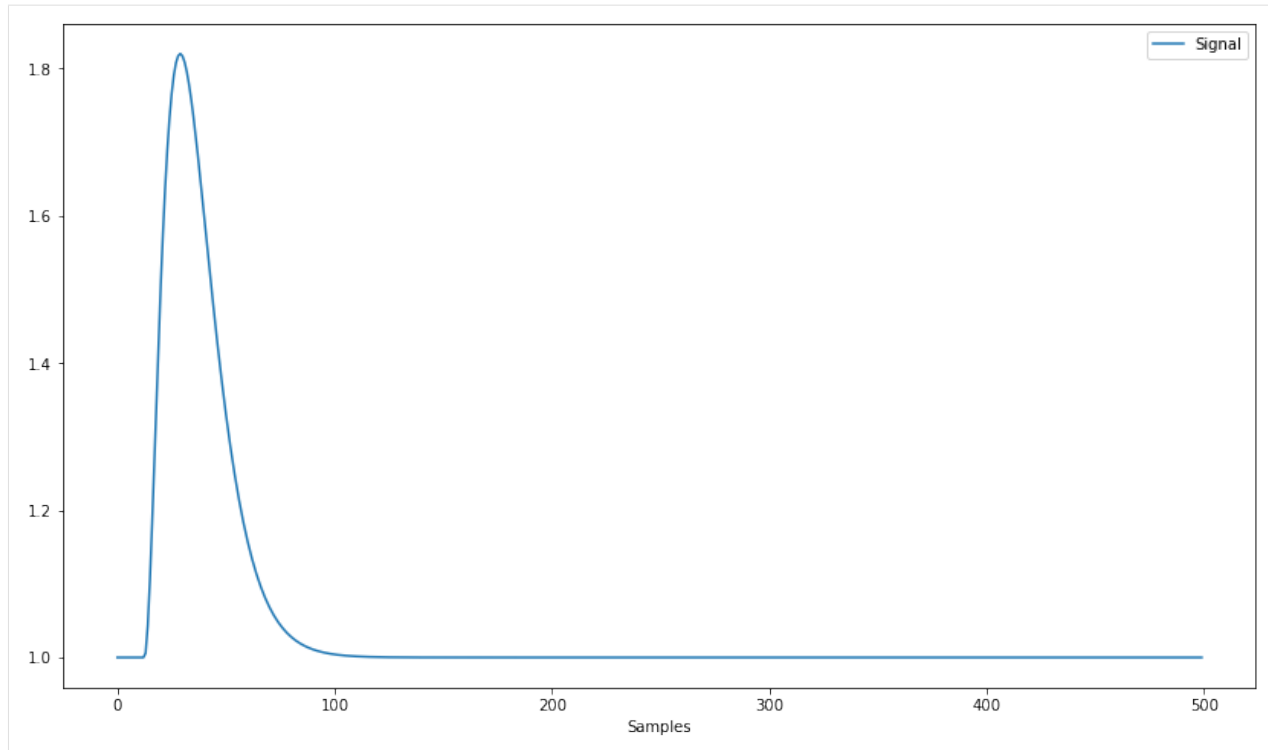
In this example, we will try to approximate this Skin Conductance Response (SCR) using a [gamma distribution](#), which is quite a flexible distribution defined by 3 parameters (**a**, **loc** and **scale**).

On top of these 3 parameters, we will add 2 more, the **intercept** and a **size** parameter to give it more flexibility.

```
[24]: def function_gamma
      =
      .
      +
      *
      =
      =
      =
      return
```

We can start by visualizing the function with a “arbitrary” parameters:

```
[34]: = . 0 20 =500
      =
      =1 =1 =3 =0.5 =0.33
      .
```



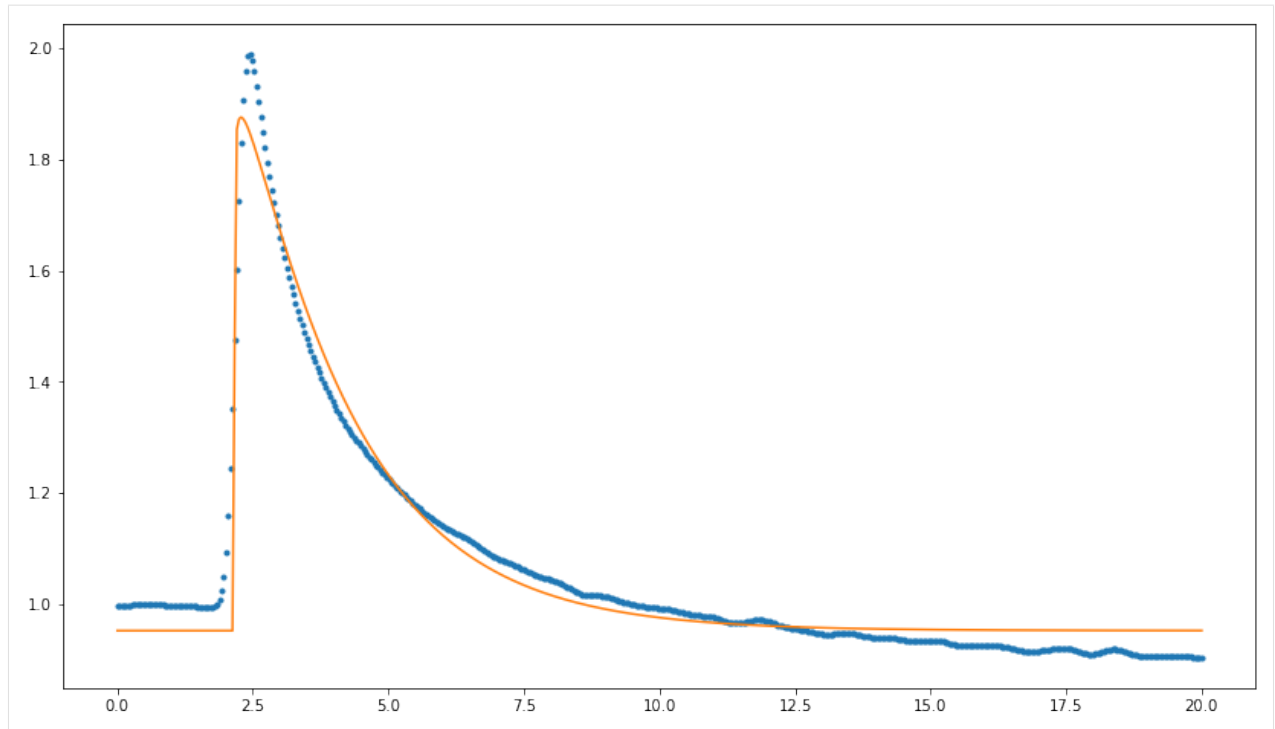
Since these values are already a good start, we will use them as “starting point” (through the `p0` argument), to help the estimation algorithm converge (otherwise it could never find the right combination of parameters).

```
[35]: fit = fit_gaussian( signal, p0=[0.5, 0.33, 1, 1, 3], plot=True )
```

```
[35]: array([0.9518406 , 2.21035698, 1.05863983, 2.16432609, 1.97046573])
```

```
[36]: fit.plot()
```

```
[36]: [<matplotlib.lines.Line2D at 0x2137cf92f40>]
```



RESOURCES

Contents:

6.1 Recording good quality signals

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing ([see this tutorial](#)).

This tutorial is a work in progress.

6.1.1 Recording

- Electrocardiogram (ECG) electrodes placement
- Facial EMG electrodes placement
- RSP belt placement
- Best Practices for Collecting Physiological Data

6.1.2 Signal quality

- Improving Data Quality of ECG
- Improving Data Quality of EDA

6.1.3 Artifacts and Anomalies

- Identifying and Handling Cardiac Arrhythmia

6.2 What software for physiological signal processing

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing ([see this tutorial](#)).

If you're here, it's probably that you have (or plan to have) some **physiological data** (*aka* biosignals, e.g., ECG for cardiac activity, RSP for respiration, EDA for electrodermal activity, EMG for muscle activity etc.), that you plan to **process and analyze these signals** and that **you don't know where to start**. Whether you are an undergrad, a master or PhD student, a postdoc or a full professor, you're at the right place.

So let's discuss a few things to consider to best decide on your options.

6.2.1 Software vs. programming language (packages)

In this context, a software would be a program that you download, install (through some sort of *.exe* file), and start similarly to most of the other programs installed on our computers.

They are appealing because of their (**apparent**) **simplicity and familiarity** of usage: you can click on icons and menus and you can *see* all the available options, which makes it easy for exploration. In a way, it also feels *safe*, because you can always close the software, press "*do not save the changes*", and start again.

Unfortunately, when it comes to science, this comes with a set of **limitations**; they are in general quite expensive, are limited to the set of included features (it's not easy to use one feature from one software, and then another from another one), have a slow pace of updates (and thus often don't include the most recent and cutting-edge algorithms, but rather well-established ones), and are not open-source (and thus prevent to run fully reproducible analyses).

- **Software for biosignals processing**

- **AcqKnowledge**: General physiological analysis software (ECG, PPG, RSP, EDA, EMG, ...).
- **Kubios**: Heart-rate variability (HRV).

Unfortunately, it's the preferred option for many researchers. *Why?* For PIs, it's usually because they are established tools backed by some sort of company behind them, with experts, advertisers and sellers that do their job well. The companies also offer some guaranty in terms of training, updates, issues troubleshooting, etc. For younger researchers starting with physiological data analysis, it's usually because they don't have much (or any) **experience with programming languages**. They feel like there is already a lot of things to learn on the theoretical side of physiological signal processing, so they don't want to add on top of that, **learning a programming language**.

However, it is important to understand that you don't necessarily have to **know how to code** to use some of the packages. Moreover, some of them include a GUI (see below), which makes them very easy to use and a great alternative to the software mentioned above.

Note: **TLDR**; Closed proprietary software, even though seemingly appealing, might not a good investment of time or money.

6.2.2 GUI vs. code

TODO.

- **Packages with a GUI**
 - `Ledalab`: EDA (*Matlab*).
 - `PsPM`: Primarily EDA (*Matlab*).
 - `biopeaks`: ECG, PPG (*Python*).
 - `mnelab`: EEG (*Python*).

Note: TLDR; While GUIs can be good alternatives and a first step to dive into programming language-based tools, coding will provide you with more freedom, incredible power and the best fit possible for your data and issues.

6.2.3 Matlab vs. Python vs. R vs. Julia

What is the best programming language for physiological data analysis?

Matlab is the historical main contender. However... *TODO.*

- **Python-based packages**
 - `NeuroKit2`: ECG, PPG, RSP, EDA, EMG.
 - `BioSPPy`: ECG, RSP, EDA, EMG.
 - `PySiology`: ECG, EDA, EMG.
 - `pyphysio`: ECG, PPG, EDA.
 - `HeartPy`: ECG.
 - `hrv`: ECG.
 - `hrv-analysis`: ECG.
 - `pyhrv`: ECG.
 - `py-ecg-detectors`: ECG.
 - `Systole`: PPG.
 - `eda-explorer`: EDA.
 - `Pypsy`: EDA.
 - `MNE`: EEG.
 - `tensorpac`: EEG.
 - `PyGaze`: Eye-tracking.
 - `PyTrack`: Eye-tracking.
- **Matlab-based packages**
 - `BreatheEasyEDA`: EDA.
 - `EDA`: EDA.
 - `unfold`: EEG.

6.3 Additional Resources

Hint: Would like to add something? Join us by contributing ([see this tutorial](#)).

6.3.1 General Neuroimaging

- Seven quick tips for analysis scripts in neuroimaging (van Vliet, 2020).
- Neuroimaging tutorials and resources

6.3.2 ECG

- Improving Data Quality: ECG
- Understanding ECG waves
- Analysing a Noisy ECG Signal
- All About HRV Part 4: Respiratory Sinus Arrhythmia

6.3.3 EDA

- Improving Data Quality: EDA
- All About EDA Part 1: Introduction to Electrodermal Activity
- All About EDA Part 2: Components of Skin Conductance
- Skin Conductance Response – What it is and How to Measure it

6.3.4 EEG

- Electroencephalogram (EEG) Recording Protocol (Farrens et al., 2019)
- Compute the average bandpower of an EEG signal

FUNCTIONS

7.1 ECG

Submodule for NeuroKit.

ecg_analyze (*data*, *sampling_rate*=1000, *method*='auto')

Performs ECG analysis on either epochs (event-related analysis) or on longer periods of data such as resting-state data.

Parameters

- **data** (*Union[dict, pd.DataFrame]*) – A dictionary of epochs, containing one DataFrame per epoch, usually obtained via *epochs_create()*, or a DataFrame containing all epochs, usually obtained via *epochs_to_df()*. Can also take a DataFrame of processed signals from a longer period of data, typically generated by *ecg_process()* or *bio_process()*. Can also take a dict containing sets of separate periods of data.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **method** (*str*) – Can be one of ‘event-related’ for event-related analysis on epochs, or ‘interval-related’ for analysis on longer periods of data. Defaults to ‘auto’ where the right method will be chosen based on the mean duration of the data (‘event-related’ for duration under 10s).

Returns *DataFrame* – A dataframe containing the analyzed ECG features. If event-related analysis is conducted, each epoch is indicated by the *Label* column. See *ecg_eventrelated()* and *ecg_intervalrelated()* docstrings for details.

See also:

bio_process(), *ecg_process()*, *epochs_create()*, *ecg_eventrelated()*,
ecg_intervalrelated()

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example 1: Download the data for event-related analysis
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data for event-related analysis
>>> df, info = nk.bio_process(ecg=data["ECG"], sampling_rate=100)
>>> events = nk.events_find(data["Photosensor"], threshold_keep='below',
```

(continues on next page)

(continued from previous page)

```

...             event_conditions=["Negative", "Neutral",
...                               "Neutral", "Negative"])
>>> epochs = nk.epochs_create(df, events, sampling_rate=100, epochs_start=-0.1,
↳ epochs_end=1.9)
>>>
>>> # Analyze
>>> nk.ecg_analyze(epochs, sampling_rate=100)
  Label Condition  ... ECG_Phase_Completion_Ventricular  ECG_Quality_Mean
1      1  Negative  ...                               ...              ...
2      2   Neutral  ...                               ...              ...
3      3   Neutral  ...                               ...              ...
4      4  Negative  ...                               ...              ...

```

[4 rows x 17 columns] >>> >>> # Example 2: Download the resting-state data >>> data = nk.data("bio_resting_5min_100hz") >>> >>> # Process the data >>> df, info = nk.ecg_process(data["ECG"], sampling_rate=100) >>> >>> # Analyze >>> nk.ecg_analyze(df, sampling_rate=100) #doctest: +ELLIPSIS

ECG_Rate_Mean HRV_RMSSD ...

0 ...

[1 rows x 37 columns]

ecg_clean (*ecg_signal*, *sampling_rate=1000*, *method='neurokit'*)

Clean an ECG signal.

Prepare a raw ECG signal for R-peak detection with the specified method.

Parameters

- **ecg_signal** (*Union[list, np.array, pd.Series]*) – The raw ECG channel.
- **sampling_rate** (*int*) – The sampling frequency of *ecg_signal* (in Hz, i.e., samples/second). Defaults to 1000.
- **method** (*str*) – The processing pipeline to apply. Can be one of ‘neurokit’ (default), ‘biosppy’, ‘pantompkins1985’, ‘hamilton2002’, ‘elgendi2010’, ‘engzeemod2012’.

Returns *array* – Vector containing the cleaned ECG signal.

See also:

ecg_findpeaks(), *signal_rate()*, *ecg_process()*, *ecg_plot()*

Examples

```

>>> import pandas as pd
>>> import neurokit2 as nk
>>> import matplotlib.pyplot as plt
>>>
>>> ecg = nk.ecg_simulate(duration=10, sampling_rate=1000)
>>> signals = pd.DataFrame({"ECG_Raw" : ecg,
...                         "ECG_NeuroKit" : nk.ecg_clean(ecg, sampling_rate=1000,
↳ method="neurokit"),
...                         "ECG_BioSPPy" : nk.ecg_clean(ecg, sampling_rate=1000,
↳ method="biosppy"),
...                         "ECG_PanTompkins" : nk.ecg_clean(ecg, sampling_
↳ rate=1000, method="pantompkins1985"),
...                         "ECG_Hamilton" : nk.ecg_clean(ecg, sampling_rate=1000,
↳ method="hamilton2002"),

```

(continues on next page)

(continued from previous page)

```

...                               "ECG_Elgendy" : nk.ecg_clean(ecg, sampling_rate=1000,
method="elgendy2010"),
...                               "ECG_EngZeeMod" : nk.ecg_clean(ecg, sampling_
rate=1000, method="engzeemod2012"))
>>> signals.plot()
<matplotlib.axes._subplots.AxesSubplot object at ...>

```

References

- Jiapu Pan and Willis J. Tompkins. A Real-Time QRS Detection Algorithm. In: IEEE Transactions on Biomedical Engineering BME-32.3 (1985), pp. 230–236.
- Hamilton, Open Source ECG Analysis Software Documentation, E.P.Limited, 2002.

ecg_delineate (*ecg_cleaned*, *rpeaks=None*, *sampling_rate=1000*, *method='peak'*, *show=False*, *show_type='peaks'*, *check=False*)
Delineate QRS complex.

Function to delineate the QRS complex.

- **Cardiac Cycle:** A typical ECG heartbeat consists of a P wave, a QRS complex and a T wave. The P wave represents the wave of depolarization that spreads from the SA-node throughout the atria. The QRS complex reflects the rapid depolarization of the right and left ventricles. Since the ventricles are the largest part of the heart, in terms of mass, the QRS complex usually has a much larger amplitude than the P-wave. The T wave represents the ventricular repolarization of the ventricles. On rare occasions, a U wave can be seen following the T wave. The U wave is believed to be related to the last remnants of ventricular repolarization.

Parameters

- **ecg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned ECG channel as returned by *ecg_clean()*.
- **rpeaks** (*Union[list, np.array, pd.Series]*) – The samples at which R-peaks occur. Accessible with the key “ECG_R_Peaks” in the info dictionary returned by *ecg_findpeaks()*.
- **sampling_rate** (*int*) – The sampling frequency of *ecg_signal* (in Hz, i.e., samples/second). Defaults to 500.
- **method** (*str*) – Can be one of ‘peak’ (default) for a peak-based method, ‘cwt’ for continuous wavelet transform or ‘dwt’ for discrete wavelet transform.
- **show** (*bool*) – If True, will return a plot to visualizing the delineated waves information.
- **show_type** (*str*) – The type of delineated waves information showed in the plot.
- **check** (*bool*) – Defaults to False.

Returns

- **waves** (*dict*) – A dictionary containing additional information. For derivative method, the dictionary contains the samples at which P-peaks, Q-peaks, S-peaks, T-peaks, P-onsets and T-offsets occur, accessible with the key “ECG_P_Peaks”, “ECG_Q_Peaks”, “ECG_S_Peaks”, “ECG_T_Peaks”, “ECG_P_Onsets”, “ECG_T_Offsets” respectively.

For wavelet methods, the dictionary contains the samples at which P-peaks, T-peaks, P-onsets, P-offsets, T-onsets, T-offsets, QRS-onsets and QRS-offsets occur, accessible with the key “ECG_P_Peaks”, “ECG_T_Peaks”, “ECG_P_Onsets”, “ECG_P_Offsets”,

“ECG_T_Onsets”, “ECG_T_Offsets”, “ECG_R_Onsets”, “ECG_R_Offsets” respectively.

- **signals** (*DataFrame*) – A *DataFrame* of same length as the input signal in which occurrences of peaks, onsets and offsets marked as “1” in a list of zeros.

See also:

`ecg_clean()`, `signal_fixpeaks()`, `ecg_peaks()`, `signal_rate()`, `ecg_process()`, `ecg_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=10, sampling_rate=1000)
>>> cleaned = nk.ecg_clean(ecg, sampling_rate=1000)
>>> _, rpeaks = nk.ecg_peaks(cleaned)
>>> signals, waves = nk.ecg_delineate(cleaned, rpeaks, sampling_rate=1000, method=
-> "peak")
>>> nk.events_plot(waves["ECG_P_Peaks"], cleaned)
<Figure ...>
>>> nk.events_plot(waves["ECG_T_Peaks"], cleaned)
<Figure ...>
```

References

- Martínez, J. P., Almeida, R., Olmos, S., Rocha, A. P., & Laguna, P. (2004). A wavelet-based ECG delineator: evaluation on standard databases. *IEEE Transactions on biomedical engineering*, 51(4), 570-581.

ecg_eventrelated (*epochs*, *silent=False*)

Performs event-related ECG analysis on epochs.

Parameters

- **epochs** (*Union[dict, pd.DataFrame]*) – A dict containing one *DataFrame* per event/trial, usually obtained via `epochs_create()`, or a *DataFrame* containing all epochs, usually obtained via `epochs_to_df()`.
- **silent** (*bool*) – If True, silence possible warnings.

Returns

DataFrame – A dataframe containing the analyzed ECG features for each epoch, with each epoch indicated by the *Label* column (if not present, by the *Index* column). The analyzed features consist of the following:

- “ECG_Rate_Max”: the maximum heart rate after stimulus onset.
- “ECG_Rate_Min”: the minimum heart rate after stimulus onset.
- “ECG_Rate_Mean”: the mean heart rate after stimulus onset.
- “ECG_Rate_Max_Time”: the time at which maximum heart rate occurs.
- “ECG_Rate_Min_Time”: the time at which minimum heart rate occurs.

- *"ECG_Phase_Atrial"*: indication of whether the onset of the event concurs with respiratory systole (1) or diastole (0).
- *"ECG_Phase_Ventricular"*: indication of whether the onset of the event concurs with respiratory systole (1) or diastole (0).
- *"ECG_Phase_Atrial_Completion"*: indication of the stage of the current cardiac (atrial) phase (0 to 1) at the onset of the event.
- *"ECG_Phase_Ventricular_Completion"*: indication of the stage of the current cardiac (ventricular) phase (0 to 1) at the onset of the event.

We also include the following *experimental* features related to the parameters of a quadratic model:

- *"ECG_Rate_Trend_Linear"*: The parameter corresponding to the linear trend.
- *"ECG_Rate_Trend_Quadratic"*: The parameter corresponding to the curvature.
- *"ECG_Rate_Trend_R2"*: the quality of the quadratic model. If too low, the parameters might not be reliable or meaningful.

See also:

`events_find()`, `epochs_create()`, `bio_process()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example with simulated data
>>> ecg, info = nk.ecg_process(nk.ecg_simulate(duration=20))
>>>
>>> # Process the data
>>> epochs = nk.epochs_create(ecg, events=[5000, 10000, 15000],
...                           epochs_start=-0.1, epochs_end=1.9)
>>> nk.ecg_eventrelated(epochs)
Label  Event_Onset  ...  ECG_Phase_Completion_Ventricular  ECG_Quality_Mean
1      1           ...  ...
2      2           ...  ...
3      3           ...  ...
```

```
[3 rows x 16 columns] >>> >>> # Example with real data >>> data = nk.data("bio_eventrelated_100hz")
>>> >>> # Process the data >>> df, info = nk.bio_process(ecg=data["ECG"], sampling_rate=100) >>>
events = nk.events_find(data["Photosensor"], ... threshold_keep='below', ... event_conditions=["Negative",
"Neutral", ... "Neutral", "Negative"]) >>> epochs = nk.epochs_create(df, events, sampling_rate=100, ...
epochs_start=-0.1, epochs_end=1.9) >>> nk.ecg_eventrelated(epochs) #doctest: +ELLIPSIS
```

Label Condition ... ECG_Phase_Completion_Ventricular ECG_Quality_Mean

```
1 1 Negative ... .. 2 2 Neutral ... .. 3 3 Neutral ... .. 4 4 Negative ... ..
```

[4 rows x 17 columns]

ecg_findpeaks (*ecg_cleaned*, *sampling_rate=1000*, *method='neurokit'*, *show=False*)

Find R-peaks in an ECG signal.

Low-level function used by *ecg_peaks()* to identify R-peaks in an ECG signal using a different set of algorithms. See *ecg_peaks()* for details.

Parameters

- **ecg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned ECG channel as returned by `ecg_clean()`.
- **sampling_rate** (*int*) – The sampling frequency of `ecg_signal` (in Hz, i.e., samples/second). Defaults to 1000.
- **method** (*string*) – The algorithm to be used for R-peak detection. Can be one of ‘neurokit’ (default), ‘pantompkins1985’, ‘hamilton2002’, ‘christov2004’, ‘gamboa2008’, ‘elgendi2010’, ‘engzeemod2012’, ‘kalidas2017’, ‘martinez2003’, ‘rodrigues2020’ or ‘promac’.
- **show** (*bool*) – If True, will return a plot to visualizing the thresholds used in the algorithm. Useful for debugging.

Returns **info** (*dict*) – A dictionary containing additional information, in this case the samples at which R-peaks occur, accessible with the key “ECG_R_Peaks”.

See also:

`ecg_clean()`, `signal_fixpeaks()`, `ecg_peaks()`, `ecg_rate()`, `ecg_process()`, `ecg_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=10, sampling_rate=1000)
>>> cleaned = nk.ecg_clean(ecg, sampling_rate=1000)
>>> info = nk.ecg_findpeaks(cleaned)
>>> nk.events_plot(info["ECG_R_Peaks"], cleaned)
<Figure ...>
```

```
>>>
>>> # Different methods
>>> neurokit = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="neurokit"), method=
↳ "neurokit")
>>> pantompkins1985 = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="pantompkins1985
↳ "), method="pantompkins1985")
>>> hamilton2002 = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="hamilton2002"),
↳ method="hamilton2002")
>>> martinez2003 = nk.ecg_findpeaks(cleaned, method="martinez2003")
>>> christov2004 = nk.ecg_findpeaks(cleaned, method="christov2004")
>>> gamboa2008 = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="gamboa2008"), method=
↳ "gamboa2008")
>>> elgendi2010 = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="elgendi2010"),
↳ method="elgendi2010")
>>> engzeemod2012 = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="engzeemod2012"),
↳ method="engzeemod2012")
>>> kalidas2017 = nk.ecg_findpeaks(nk.ecg_clean(ecg, method="kalidas2017"),
↳ method="kalidas2017")
>>> rodrigues2020 = nk.ecg_findpeaks(cleaned, method="rodrigues2020")
>>>
>>> # Visualize
>>> nk.events_plot([neurokit["ECG_R_Peaks"],
...                  pantompkins1985["ECG_R_Peaks"],
...                  hamilton2002["ECG_R_Peaks"],
...                  christov2004["ECG_R_Peaks"],
...                  gamboa2008["ECG_R_Peaks"],
```

(continues on next page)

(continued from previous page)

```

...         elgendi2010["ECG_R_Peaks"],
...         engzeemod2012["ECG_R_Peaks"],
...         kalidas2017["ECG_R_Peaks"],
...         martinez2003["ECG_R_Peaks"],
...         rodrigues2020["ECG_R_Peaks"]], cleaned)
<Figure ...>
>>>
>>> # Method-agreement
>>> ecg = nk.ecg_simulate(duration=10, sampling_rate=500)
>>> ecg = nk.signal_distort(ecg,
...                          sampling_rate=500,
...                          noise_amplitude=0.2, noise_frequency=[25, 50],
...                          artifacts_amplitude=0.2, artifacts_frequency=50)
>>> nk.ecg_findpeaks(ecg, sampling_rate=1000, method="promac", show=True)
{'ECG_R_Peaks': array(...)}

```

References

- Gamboa, H. (2008). Multi-modal behavioral biometrics based on hci and electrophysiology. PhD Thesis Universidade.
- Zong, W., Heldt, T., Moody, G. B., & Mark, R. G. (2003, September). An open-source algorithm to detect onset of arterial blood pressure pulses. In *Computers in Cardiology, 2003* (pp. 259-262). IEEE.
- Hamilton, Open Source ECG Analysis Software Documentation, E.P.Limited, 2002.
- Pan, J., & Tompkins, W. J. (1985). A real-time QRS detection algorithm. *IEEE transactions on biomedical engineering*, (3), 230-236.
- Engelse, W. A. H., & Zeelenberg, C. (1979). A single scan algorithm for QRS detection and feature extraction *IEEE Comput Cardiol*. Long Beach: IEEE Computer Society.
- Lourenço, A., Silva, H., Leite, P., Lourenço, R., & Fred, A. L. (2012, February). Real Time Electrocardiogram Segmentation for Finger based ECG Biometrics. In *Biosignals* (pp. 49-54).

ecg_intervalrelated (*data*, *sampling_rate=1000*)

Performs ECG analysis on longer periods of data (typically > 10 seconds), such as resting-state data.

Parameters

- **data** (*Union[dict, pd.DataFrame]*) – A DataFrame containing the different processed signal(s) as different columns, typically generated by *ecg_process()* or *bio_process()*. Can also take a dict containing sets of separately processed DataFrames.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).

Returns

DataFrame – A dataframe containing the analyzed ECG features. The analyzed features consist of the following:

- *"ECG_Rate_Mean"*: the mean heart rate.
- *"ECG_HRV"*: the different heart rate variability metrics.

See *hrv_summary()* docstrings for details.

See also:

bio_process(), *ecg_eventrelated()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Process the data
>>> df, info = nk.ecg_process(data["ECG"], sampling_rate=100)
>>>
>>> # Single dataframe is passed
>>> nk.ecg_intervalrelated(df, sampling_rate=100)
    ECG_Rate_Mean  HRV_RMSSD  ...
0               ...
```

```
[1 rows x 55 columns] >>> >>> epochs = nk.epochs_create(df, events=[0, 15000], sampling_rate=100, ...
epochs_end=150) >>> nk.ecg_intervalrelated(epochs) #doctest: +ELLIPSIS
```

```
    ECG_Rate_Mean HRV_RMSSD ...
```

```
1 ...
```

```
[2 rows x 55 columns]
```

ecg_peaks (*ecg_cleaned*, *sampling_rate=1000*, *method='neurokit'*, *correct_artifacts=False*)

Find R-peaks in an ECG signal.

Find R-peaks in an ECG signal using the specified method.

Parameters

- **ecg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned ECG channel as returned by *ecg_clean()*.
- **sampling_rate** (*int*) – The sampling frequency of *ecg_signal* (in Hz, i.e., samples/second). Defaults to 1000.
- **method** (*string*) – The algorithm to be used for R-peak detection. Can be one of ‘neurokit’ (default), ‘pamtompkins1985’, ‘hamilton2002’, ‘christov2004’, ‘gamboa2008’, ‘elgendi2010’, ‘engzeemod2012’ or ‘kalidas2017’.
- **correct_artifacts** (*bool*) – Whether or not to identify artifacts as defined by Jukka A. Lipponen & Mika P. Tarvainen (2019): A robust algorithm for heart rate variability time series artefact correction using novel beat classification, Journal of Medical Engineering & Technology, DOI: 10.1080/03091902.2019.1640306.

Returns

- **signals** (*DataFrame*) – A DataFrame of same length as the input signal in which occurrences of R-peaks marked as “1” in a list of zeros with the same length as *ecg_cleaned*. Accessible with the keys “ECG_R_Peaks”.
- **info** (*dict*) – A dictionary containing additional information, in this case the samples at which R-peaks occur, accessible with the key “ECG_R_Peaks”.

See also:

```
ecg_clean(), ecg_findpeaks(), ecg_process(), ecg_plot(), signal_rate(),
signal_fixpeaks()
```

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=10, sampling_rate=1000)
>>> cleaned = nk.ecg_clean(ecg, sampling_rate=1000)
>>> signals, info = nk.ecg_peaks(cleaned, correct_artifacts=True)
>>> nk.events_plot(info["ECG_R_Peaks"], cleaned)
<Figure ...>
```

References

- Gamboa, H. (2008). Multi-modal behavioral biometrics based on hci and electrophysiology. PhD Thesis Universidade.
- W. Zong, T. Heldt, G.B. Moody, and R.G. Mark. An open-source algorithm to detect onset of arterial blood pressure pulses. In Computers in Cardiology, 2003, pages 259–262, 2003.
- Hamilton, Open Source ECG Analysis Software Documentation, E.P.Limited, 2002.
- Jiapu Pan and Willis J. Tompkins. A Real-Time QRS Detection Algorithm. In: IEEE Transactions on Biomedical Engineering BME-32.3 (1985), pp. 230–236.
- C. Zeelenberg, A single scan algorithm for QRS detection and feature extraction, IEEE Comp. in Cardiology, vol. 6, pp. 37-42, 1979
- A. Lourenco, H. Silva, P. Leite, R. Lourenco and A. Fred, “Real Time Electrocardiogram Segmentation for Finger Based ECG Biometrics”, BIOSIGNALS 2012, pp. 49-54, 2012.

ecg_phase (*ecg_cleaned*, *rpeaks=None*, *delineate_info=None*, *sampling_rate=None*)
 Compute cardiac phase (for both atrial and ventricular).

Finds the cardiac phase, labelled as 1 for systole and 0 for diastole.

Parameters

- **ecg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned ECG channel as returned by *ecg_clean()*.
- **rpeaks** (*list or array or DataFrame or Series or dict*) – The samples at which the different ECG peaks occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with *ecg_findpeaks()* or *ecg_peaks()*.
- **delineate_info** (*dict*) – A dictionary containing additional information of ecg delineation and can be obtained with *ecg_delineate()*.
- **sampling_rate** (*int*) – The sampling frequency of *ecg_signal* (in Hz, i.e., samples/second). Defaults to None.

Returns

signals (*DataFrame*) – A DataFrame of same length as *ecg_signal* containing the following columns:

- *“ECG_Phase_Atrial”*: cardiac phase, marked by “1” for systole and “0” for diastole.
- *“ECG_Phase_Completion_Atrial”*: cardiac phase (atrial) completion, expressed in percentage (from 0 to 1), representing the stage of the current cardiac phase.
- *“ECG_Phase_Ventricular”*: cardiac phase, marked by “1” for systole and “0” for diastole.

- *"ECG_Phase_Completion_Ventricular"*: cardiac phase (ventricular) completion, expressed in percentage (from 0 to 1), representing the stage of the current cardiac phase.

See also:

`ecg_clean()`, `ecg_peaks()`, `ecg_process()`, `ecg_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=10, sampling_rate=1000)
>>> cleaned = nk.ecg_clean(ecg, sampling_rate=1000)
>>> _, rpeaks = nk.ecg_peaks(cleaned)
>>> signals, waves = nk.ecg_delineate(cleaned, rpeaks, sampling_rate=1000)
>>>
>>> cardiac_phase = nk.ecg_phase(ecg_cleaned=cleaned, rpeaks=rpeaks,
...                             delineate_info=waves, sampling_rate=1000)
>>> nk.signal_plot([cleaned, cardiac_phase], standardize=True)
```

ecg_plot (*ecg_signals*, *rpeaks*=None, *sampling_rate*=None, *show_type*='default')

Visualize ECG data.

Parameters

- **ecg_signals** (*DataFrame*) – DataFrame obtained from `ecg_process()`.
- **rpeaks** (*dict*) – The samples at which the R-peak occur. Dict returned by `ecg_process()`. Defaults to None.
- **sampling_rate** (*int*) – The sampling frequency of the ECG (in Hz, i.e., samples/second). Needs to be supplied if the data should be plotted over time in seconds. Otherwise the data is plotted over samples. Defaults to None. Must be specified to plot artifacts.
- **show_type** (*str*) – Visualize the ECG data with 'default' or visualize artifacts thresholds with 'artifacts' produced by `ecg_fixpeaks()`, or 'full' to visualize both.

Returns *fig* – Figure representing a plot of the processed ecg signals (and peak artifacts).

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=15, sampling_rate=1000, heart_rate=80)
>>> signals, info = nk.ecg_process(ecg, sampling_rate=1000)
>>> nk.ecg_plot(signals, sampling_rate=1000, show_type='default')
<Figure ...>
```

See also:

`ecg_process()`

ecg_process (*ecg_signal*, *sampling_rate*=1000, *method*='neurokit')

Process an ECG signal.

Convenience function that automatically processes an ECG signal.

Parameters

- **ecg_signal** (*Union[list, np.array, pd.Series]*) – The raw ECG channel.

- **sampling_rate** (*int*) – The sampling frequency of *ecg_signal* (in Hz, i.e., samples/second). Defaults to 1000.
- **method** (*str*) – The processing pipeline to apply. Defaults to “neurokit”.

Returns

- **signals** (*DataFrame*) – A DataFrame of the same length as the *ecg_signal* containing the following columns:
 - “*ECG_Raw*”: the raw signal.
 - “*ECG_Clean*”: the cleaned signal.
 - “*ECG_R_Peaks*”: the R-peaks marked as “1” in a list of zeros.
 - “*ECG_Rate*”: heart rate interpolated between R-peaks.
 - “*ECG_P_Peaks*”: the P-peaks marked as “1” in a list of zeros
 - “*ECG_Q_Peaks*”: the Q-peaks marked as “1” in a list of zeros .
 - “*ECG_S_Peaks*”: the S-peaks marked as “1” in a list of zeros.
 - “*ECG_T_Peaks*”: the T-peaks marked as “1” in a list of zeros.
 - “*ECG_P_Onsets*”: the P-onsets marked as “1” in a list of zeros.
 - “*ECG_P_Offsets*”: the P-offsets marked as “1” in a list of zeros (only when method in *ecg_delineate* is wavelet).
 - “*ECG_T_Onsets*”: the T-onsets marked as “1” in a list of zeros (only when method in *ecg_delineate* is wavelet).
 - “*ECG_T_Offsets*”: the T-offsets marked as “1” in a list of zeros.
 - “*ECG_R_Onsets*”: the R-onsets marked as “1” in a list of zeros (only when method in *ecg_delineate* is wavelet).
 - “*ECG_R_Offsets*”: the R-offsets marked as “1” in a list of zeros (only when method in *ecg_delineate* is wavelet).
 - “*ECG_Phase_Atrial*”: cardiac phase, marked by “1” for systole and “0” for diastole.
 - “*ECG_Phase_Ventricular*”: cardiac phase, marked by “1” for systole and “0” for diastole.
 - “*ECG_Atrial_PhaseCompletion*”: cardiac phase (atrial) completion, expressed in percentage (from 0 to 1), representing the stage of the current cardiac phase.
 - “*ECG_Ventricular_PhaseCompletion*”: cardiac phase (ventricular) completion, expressed in percentage (from 0 to 1), representing the stage of the current cardiac phase.
- **info** (*dict*) – A dictionary containing the samples at which the R-peaks occur, accessible with the key “ECG_Peaks”.

See also:

`ecg_clean()`, `ecg_findpeaks()`, `ecg_plot()`, `signal_rate()`, `signal_fixpeaks()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=15, sampling_rate=1000, heart_rate=80)
>>> signals, info = nk.ecg_process(ecg, sampling_rate=1000)
>>> nk.ecg_plot(signals)
<Figure ...>
```

ecg_quality (*ecg_cleaned*, *rpeaks=None*, *sampling_rate=1000*)
Quality of ECG Signal.

Compute a continuous index of quality of the ECG signal, by interpolating the distance of each QRS segment from the average QRS segment present in the data. This index is therefore relative, and 1 corresponds to heartbeats that are the closest to the average sample and 0 corresponds to the most distance heartbeat, from that average sample.

Returns *array* – Vector containing the quality index ranging from 0 to 1.

See also:

`ecg_segment()`, `ecg_delineate()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=30, sampling_rate=300, noise=0.2)
>>> ecg_cleaned = nk.ecg_clean(ecg, sampling_rate=300)
>>> quality = nk.ecg_quality(ecg_cleaned, sampling_rate=300)
>>>
>>> nk.signal_plot([ecg_cleaned, quality], standardize=True)
```

ecg_rate (*peaks*, *sampling_rate=1000*, *desired_length=None*, *interpolation_method='monotone_cubic'*)
Calculate signal rate from a series of peaks.

This function can also be called either via `ecg_rate()`, `ppg_rate()` or `rsp_rate()` (aliases provided for consistency).

Parameters

- **peaks** (*Union[list, np.array, pd.DataFrame, pd.Series, dict]*) – The samples at which the peaks occur. If an array is passed in, it is assumed that it was obtained with `signal_findpeaks()`. If a DataFrame is passed in, it is assumed it is of the same length as the input signal in which occurrences of R-peaks are marked as “1”, with such containers obtained with e.g., `ecg_findpeaks()` or `rsp_findpeaks()`.
- **sampling_rate** (*int*) – The sampling frequency of the signal that contains peaks (in Hz, i.e., samples/second). Defaults to 1000.
- **desired_length** (*int*) – If left at the default None, the returned rate will have the same number of elements as peaks. If set to a value larger than the sample at which the last peak occurs in the signal (i.e., `peaks[-1]`), the returned rate will be interpolated between peaks over *desired_length* samples. To interpolate the rate over the entire duration of the signal, set *desired_length* to the number of samples in the signal. Cannot be smaller than or equal to the sample at which the last peak occurs in the signal. Defaults to None.
- **interpolation_method** (*str*) – Method used to interpolate the rate between peaks. See `signal_interpolate()`. ‘monotone_cubic’ is chosen as the default interpolation method

since it ensures monotone interpolation between data points (i.e., it prevents physiologically implausible “overshoots” or “undershoots” in the y-direction). In contrast, the widely used cubic spline interpolation does not ensure monotonicity.

Returns *array* – A vector containing the rate.

See also:

`signal_findpeaks()`, `signal_fixpeaks()`, `signal_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, sampling_rate=1000, frequency=1)
>>> info = nk.signal_findpeaks(signal)
>>>
>>> rate = nk.signal_rate(peaks=info["Peaks"], desired_length=len(signal))
>>> fig = nk.signal_plot(rate)
>>> fig
```

ecg_rsa (*ecg_signals*, *rsp_signals=None*, *rpeaks=None*, *sampling_rate=1000*, *continuous=False*)

Respiratory Sinus Arrhythmia (RSA)

Respiratory sinus arrhythmia (RSA), also referred to as ‘cardiac coherence’, is the naturally occurring variation in heart rate during the breathing cycle. Metrics to quantify it are often used as a measure of parasympathetic nervous system activity. Neurophysiology informs us that the functional output of the myelinated vagus originating from the nucleus ambiguus has a respiratory rhythm. Thus, there would a temporal relation between the respiratory rhythm being expressed in the firing of these efferent pathways and the functional effect on the heart rate rhythm manifested as RSA. Importantly, several methods exist to quantify RSA:

- The *Peak-to-trough (P2T)* algorithm measures the statistical range in milliseconds of the heart

period oscillation associated with synchronous respiration. Operationally, subtracting the shortest heart period during inspiration from the longest heart period during a breath cycle produces an estimate of RSA during each breath. The peak-to-trough method makes no statistical assumption or correction (e.g., adaptive filtering) regarding other sources of variance in the heart period time series that may confound, distort, or interact with the metric such as slower periodicities and baseline trend. Although it has been proposed that the P2T method “acts as a time-domain filter dynamically centered at the exact ongoing respiratory frequency” (Grossman, 1992), the method does not transform the time series in any way, as a filtering process would. Instead the method uses knowledge of the ongoing respiratory cycle to associate segments of the heart period time series with either inhalation or exhalation (Lewis, 2012).

- The *Porges-Bohrer (PB)* algorithm assumes that heart period time series reflect the sum of several

component time series. Each of these component time series may be mediated by different neural mechanisms and may have different statistical features. The Porges-Bohrer method applies an algorithm that selectively extracts RSA, even when the periodic process representing RSA is superimposed on a complex baseline that may include aperiodic and slow periodic processes. Since the method is designed to remove sources of variance in the heart period time series other than the variance within the frequency band of spontaneous breathing, the method is capable of accurately quantifying RSA when the signal to noise ratio is low.

Parameters

- **ecg_signals** (*DataFrame*) – *DataFrame* obtained from `ecg_process()`. Should contain columns *ECG_Rate* and *ECG_R_Peaks*. Can also take a *DataFrame* comprising of both ECG and RSP signals, generated by `bio_process()`.

- **rsp_signals** (*DataFrame*) – DataFrame obtained from *rsp_process()*. Should contain columns *RSP_Phase* and *RSP_PhaseCompletion*. No impact when a DataFrame comprising of both the ECG and RSP signals are passed as *ecg_signals*. Defaults to None.
- **rpeaks** (*dict*) – The samples at which the R-peaks of the ECG signal occur. Dict returned by *ecg_peaks()*, *ecg_process()*, or *bio_process()*. Defaults to None.
- **sampling_rate** (*int*) – The sampling frequency of signals (in Hz, i.e., samples/second).
- **continuous** (*bool*) – If False, will return RSA properties computed from the data (one value per index). If True, will return continuous estimations of RSA of the same length as the signal. See below for more details.

Returns

rsa (*dict*) – A dictionary containing the RSA features, which includes:

- **"RSA_P2T_Values"**: the estimate of RSA during each breath cycle, produced by subtracting the shortest heart period (or RR interval) from the longest heart period in ms.
- **"RSA_P2T_Mean"**: the mean peak-to-trough across all cycles in ms
- **"RSA_P2T_Mean_log"**: the logarithm of the mean of RSA estimates.
- **"RSA_P2T_SD"**: the standard deviation of all RSA estimates.
- **"RSA_P2T_NoRSA"**: the number of breath cycles from which RSA could not be calculated.
- **"RSA_PorgesBohrer"**: the Porges-Bohrer estimate of RSA, optimal when the signal to noise ratio is low, in $\ln(\text{ms}^2)$.

Example

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data
>>> ecg_signals, info = nk.ecg_process(data["ECG"], sampling_rate=100)
>>> rsp_signals, _ = nk.rsp_process(data["RSP"], sampling_rate=100)
>>>
>>> # Get RSA features
>>> nk.ecg_rsa(ecg_signals, rsp_signals, info, sampling_rate=100,
↳continuous=False)
{'RSA_P2T_Mean': ...,
 'RSA_P2T_Mean_log': ...,
 'RSA_P2T_SD': ...,
 'RSA_P2T_NoRSA': ...,
 'RSA_PorgesBohrer': ...}
>>>
>>> # Get RSA as a continuous signal
>>> rsa = nk.ecg_rsa(ecg_signals, rsp_signals, info, sampling_rate=100,
↳continuous=True)
>>> rsa
      RSA_P2T
0      0.09
1      0.09
```

(continues on next page)

(continued from previous page)

```
2      0.09
...    ...
```

```
[15000 rows x 1 columns] >>> nk.signal_plot([ecg_signals["ECG_Rate"], rsp_signals["RSP_Rate"], rsa], stan-
standardize=True)
```

References

- Servant, D., Logier, R., Mouster, Y., & Goudemand, M. (2009). La variabilité de la fréquence cardiaque. Intérêts en psychiatrie. L'Encéphale, 35(5), 423–428. doi:10.1016/j.encep.2008.06.016
- Lewis, G. F., Furman, S. A., McCool, M. F., & Porges, S. W. (2012). Statistical strategies to quantify respiratory sinus arrhythmia: Are commonly used metrics equivalent?. Biological psychology, 89(2), 349-364.
- Zohar, A. H., Cloninger, C. R., & McCraty, R. (2013). Personality and heart rate variability: exploring pathways from personality to cardiac coherence and health. Open Journal of Social Sciences, 1(06), 32.

ecg_rsp (*ecg_rate*, *sampling_rate*=1000, *method*='vangent2019')

Extract ECG Derived Respiration (EDR).

This implementation is far from being complete, as the information in the related papers prevents me from getting a full understanding of the procedure. Help is required!

Parameters

- **ecg_rate** (*array*) – The heart rate signal as obtained via *ecg_rate()*.
- **sampling_rate** (*int*) – The sampling frequency of the signal that contains the R-peaks (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **method** (*str*) – Can be one of 'vangent2019' (default), 'soni2019', 'charlton2016' or 'sarkar2015'.

Returns *array* – A Numpy array containing the heart rate.

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> # Get heart rate
>>> data = nk.data("bio_eventrelated_100hz")
>>> rpeaks, info = nk.ecg_peaks(data["ECG"], sampling_rate=100)
>>> ecg_rate = nk.signal_rate(rpeaks, sampling_rate=100, desired_
->length=len(rpeaks))
>>>
>>>
>>> # Get ECG Derived Respiration (EDR)
>>>edr = nk.ecg_rsp(ecg_rate, sampling_rate=100)
>>> nk.standardize(pd.DataFrame({"EDR": edr, "RSP": data["RSP"]})).plot()
<matplotlib.axes._subplots.AxesSubplot object at ...>
>>>
>>> # Method comparison (the closer to 0 the better)
>>> nk.standardize(pd.DataFrame({"True RSP": data["RSP"],
...                               "vangent2019": nk.ecg_rsp(ecg_rate, sampling_
->rate=100, method="vangent2019")
```

(continues on next page)

(continued from previous page)

```

...                               "sarkar2015": nk.ecg_rsp(ecg_rate, sampling_
→rate=100, method="sarkar2015"),
...                               "charlton2016": nk.ecg_rsp(ecg_rate, sampling_
→rate=100, method="charlton2016"),
...                               "soni2019": nk.ecg_rsp(ecg_rate, sampling_
→rate=100,
...                                                       method="soni2019")))).
→plot()
<matplotlib.axes._subplots.AxesSubplot object at ...>

```

References

- van Gent, P., Farah, H., van Nes, N., & van Arem, B. (2019). HeartPy: A novel heart rate algorithm for the analysis of noisy signals. *Transportation research part F: traffic psychology and behaviour*, 66, 368-378.
- Sarkar, S., Bhattacharjee, S., & Pal, S. (2015). Extraction of respiration signal from ECG for respiratory rate estimation.
- Charlton, P. H., Bonnici, T., Tarassenko, L., Clifton, D. A., Beale, R., & Watkinson, P. J. (2016). An assessment of algorithms to estimate respiratory rate from the electrocardiogram and photoplethysmogram. *Physiological measurement*, 37(4), 610.
- Soni, R., & Muniyandi, M. (2019). Breath rate variability: a novel measure to study the meditation effects. *International Journal of Yoga*, 12(1), 45.

ecg_segment (*ecg_cleaned*, *rpeaks=None*, *sampling_rate=1000*, *show=False*)

Segment an ECG signal into single heartbeats.

Parameters

- **ecg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned ECG channel as returned by *ecg_clean()*.
- **rpeaks** (*dict*) – The samples at which the R-peaks occur. Dict returned by *ecg_peaks()*. Defaults to None.
- **sampling_rate** (*int*) – The sampling frequency of *ecg_signal* (in Hz, i.e., samples/second). Defaults to 1000.
- **show** (*bool*) – If True, will return a plot of heartbeats. Defaults to False.

Returns *dict* – A dict containing DataFrames for all segmented heartbeats.

See also:

ecg_clean(), *ecg_plot()*

Examples

```

>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=15, sampling_rate=1000, heart_rate=80)
>>> ecg_cleaned = nk.ecg_clean(ecg, sampling_rate=1000)
>>> nk.ecg_segment(ecg_cleaned, rpeaks=None, sampling_rate=1000, show=True)
{'1':      Signal  Index  Label
...
'2':      Signal  Index  Label

```

(continues on next page)

(continued from previous page)

```
...
'19':          Signal  Index  Label
...}
```

ecg_simulate (*duration=10, length=None, sampling_rate=1000, noise=0.01, heart_rate=70, method='ecgsyn', random_state=None*)
 Simulate an ECG/EKG signal.

Generate an artificial (synthetic) ECG signal of a given duration and sampling rate using either the ECGSYN dynamical model (McSharry et al., 2003) or a simpler model based on Daubechies wavelets to roughly approximate cardiac cycles.

Parameters

- **duration** (*int*) – Desired recording length in seconds.
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second).
- **length** (*int*) – The desired length of the signal (in samples).
- **noise** (*float*) – Noise level (amplitude of the laplace noise).
- **heart_rate** (*int*) – Desired simulated heart rate (in beats per minute).
- **method** (*str*) – The model used to generate the signal. Can be ‘simple’ for a simulation based on Daubechies wavelets that roughly approximates a single cardiac cycle. If ‘ecgsyn’ (default), will use an advanced model described [McSharry et al. \(2003\)](#).
- **random_state** (*int*) – Seed for the random number generator.

Returns *array* – Vector containing the ECG signal.

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> ecg1 = nk.ecg_simulate(duration=10, method="simple")
>>> ecg2 = nk.ecg_simulate(duration=10, method="ecgsyn")
>>> pd.DataFrame({"ECG_Simple": ecg1,
...              "ECG_Complex": ecg2}).plot(subplots=True)
array([<matplotlib.axes._subplots.AxesSubplot object at ...>,
       <matplotlib.axes._subplots.AxesSubplot object at ...>],
      dtype=object)
```

See also:

`rsp_simulate()`, `eda_simulate()`, `ppg_simulate()`, `emg_simulate()`

References

- McSharry, P. E., Clifford, G. D., Tarassenko, L., & Smith, L. A. (2003). A dynamical model for generating synthetic electrocardiogram signals. IEEE transactions on biomedical engineering, 50(3), 289-294. - https://github.com/diarmaidocualain/ecg_simulation

7.2 PPG

Submodule for NeuroKit.

ppg_clean (*ppg_signal*, *sampling_rate=1000*, *method='elgendi'*)

Clean a photoplethysmogram (PPG) signal.

Prepare a raw PPG signal for systolic peak detection.

Parameters

- **ppg_signal** (*Union[list, np.array, pd.Series]*) – The raw PPG channel.
- **sampling_rate** (*int*) – The sampling frequency of the PPG (in Hz, i.e., samples/second). The default is 1000.
- **method** (*str*) – The processing pipeline to apply. Can be one of “elgendi”. The default is “elgendi”.

Returns **clean** (*array*) – A vector containing the cleaned PPG.

See also:

ppg_simulate(), *ppg_findpeaks()*

Examples

```
>>> import neurokit2 as nk
>>> import matplotlib.pyplot as plt
>>>
>>> ppg = nk.ppg_simulate(heart_rate=75, duration=30)
>>> ppg_clean = nk.ppg_clean(ppg)
>>>
>>> plt.plot(ppg, label="raw PPG")
>>> plt.plot(ppg_clean, label="clean PPG")
>>> plt.legend()
```

ppg_findpeaks (*ppg_cleaned*, *sampling_rate=1000*, *method='elgendi'*, *show=False*)

Find systolic peaks in a photoplethysmogram (PPG) signal.

Parameters

- **ppg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned PPG channel as returned by *ppg_clean()*.
- **sampling_rate** (*int*) – The sampling frequency of the PPG (in Hz, i.e., samples/second). The default is 1000.
- **method** (*str*) – The processing pipeline to apply. Can be one of “elgendi”. The default is “elgendi”.

- **show** (*bool*) – If True, returns a plot of the thresholds used during peak detection. Useful for debugging. The default is False.

Returns **info** (*dict*) – A dictionary containing additional information, in this case the samples at which systolic peaks occur, accessible with the key “PPG_Peaks”.

See also:

`ppg_simulate()`, `ppg_clean()`

Examples

```
>>> import neurokit2 as nk
>>> import matplotlib.pyplot as plt
>>>
>>> ppg = nk.ppg_simulate(heart_rate=75, duration=30)
>>> ppg_clean = nk.ppg_clean(ppg)
>>> info = nk.ppg_findpeaks(ppg_clean)
>>> peaks = info["PPG_Peaks"]
>>>
>>> plt.plot(ppg, label="raw PPG")
>>> plt.plot(ppg_clean, label="clean PPG")
>>> plt.scatter(peaks, ppg[peaks], c="r", label="systolic peaks")
>>> plt.legend()
```

References

- Elgendi M, Norton I, Brearley M, Abbott D, Schuurmans D (2013) Systolic Peak Detection in

Acceleration Photoplethysmograms Measured from Emergency Responders in Tropical Conditions. PLoS ONE 8(10): e76585. doi:10.1371/journal.pone.0076585.

ppg_plot (*ppg_signals*, *sampling_rate=None*)
Visualize photoplethysmogram (PPG) data.

Parameters

- **ppg_signals** (*DataFrame*) – DataFrame obtained from `ppg_process()`.
- **sampling_rate** (*int*) – The sampling frequency of the PPG (in Hz, i.e., samples/second). Needs to be supplied if the data should be plotted over time in seconds. Otherwise the data is plotted over samples. Defaults to None.

Returns *fig* – Figure representing a plot of the processed PPG signals.

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Simulate data
>>> ppg = nk.ppg_simulate(duration=10, sampling_rate=1000, heart_rate=70)
>>>
>>> # Process signal
>>> signals, info = nk.ppg_process(ppg, sampling_rate=1000)
>>>
>>> # Plot
```

(continues on next page)

(continued from previous page)

```
>>> nk.ppg_plot(signals)
<Figure ...>
```

See also:

`ppg_process()`

ppg_process (*ppg_signal*, *sampling_rate=1000*, ***kwargs*)

Process a photoplethysmogram (PPG) signal.

Convenience function that automatically processes an electromyography signal.

Parameters

- **ppg_signal** (*Union[list, np.array, pd.Series]*) – The raw PPG channel.
- **sampling_rate** (*int*) – The sampling frequency of *emg_signal* (in Hz, i.e., samples/second).

Returns

- **signals** (*DataFrame*) – A DataFrame of same length as *emg_signal* containing the following columns: - “PPG_Raw”: the raw signal. - “PPG_Clean”: the cleaned signal. - “PPG_Rate”: the heart rate as measured based on PPG peaks. - “PPG_Peaks”: the PPG peaks marked as “1” in a list of zeros.
- **info** (*dict*) – A dictionary containing the information of peaks.

See also:

`ppg_clean()`, `ppg_findpeaks()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ppg = nk.ppg_simulate(duration=10, sampling_rate=1000, heart_rate=70)
>>> signals, info = nk.ppg_process(ppg, sampling_rate=1000)
>>> fig = nk.ppg_plot(signals)
>>> fig
```

ppg_rate (*peaks*, *sampling_rate=1000*, *desired_length=None*, *interpolation_method='monotone_cubic'*)

Calculate signal rate from a series of peaks.

This function can also be called either via `ecg_rate()`, `ppg_rate()` or `rsp_rate()` (aliases provided for consistency).

Parameters

- **peaks** (*Union[list, np.array, pd.DataFrame, pd.Series, dict]*) – The samples at which the peaks occur. If an array is passed in, it is assumed that it was obtained with `signal_findpeaks()`. If a DataFrame is passed in, it is assumed it is of the same length as the input signal in which occurrences of R-peaks are marked as “1”, with such containers obtained with e.g., `ecg_findpeaks()` or `rsp_findpeaks()`.
- **sampling_rate** (*int*) – The sampling frequency of the signal that contains peaks (in Hz, i.e., samples/second). Defaults to 1000.
- **desired_length** (*int*) – If left at the default None, the returned rated will have the same number of elements as peaks. If set to a value larger than the sample at which the last

peak occurs in the signal (i.e., `peaks[-1]`), the returned rate will be interpolated between peaks over *desired_length* samples. To interpolate the rate over the entire duration of the signal, set *desired_length* to the number of samples in the signal. Cannot be smaller than or equal to the sample at which the last peak occurs in the signal. Defaults to None.

- **interpolation_method** (*str*) – Method used to interpolate the rate between peaks. See *signal_interpolate()*. ‘monotone_cubic’ is chosen as the default interpolation method since it ensures monotone interpolation between data points (i.e., it prevents physiologically implausible “overshoots” or “undershoots” in the y-direction). In contrast, the widely used cubic spline interpolation does not ensure monotonicity.

Returns *array* – A vector containing the rate.

See also:

`signal_findpeaks()`, `signal_fixpeaks()`, `signal_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, sampling_rate=1000, frequency=1)
>>> info = nk.signal_findpeaks(signal)
>>>
>>> rate = nk.signal_rate(peaks=info["Peaks"], desired_length=len(signal))
>>> fig = nk.signal_plot(rate)
>>> fig
```

ppg_simulate (*duration=120*, *sampling_rate=1000*, *heart_rate=70*, *frequency_modulation=0.3*, *ibi_randomness=0.1*, *drift=0*, *motion_amplitude=0.1*, *powerline_amplitude=0.01*, *burst_number=0*, *burst_amplitude=1*, *random_state=None*, *show=False*)

Simulate a photoplethysmogram (PPG) signal.

Phenomenological approximation of PPG. The PPG wave is described with four landmarks: wave onset, location of the systolic peak, location of the diastolic notch and location of the diastolic peaks. These landmarks are defined as x and y coordinates (in a time series). These coordinates are then interpolated at the desired sampling rate to obtain the PPG signal.

Parameters

- **duration** (*int*) – Desired recording length in seconds. The default is 120.
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second). The default is 1000.
- **heart_rate** (*int*) – Desired simulated heart rate (in beats per minute). The default is 70.
- **frequency_modulation** (*float*) – Float between 0 and 1. Determines how pronounced respiratory sinus arrhythmia (RSA) is (0 corresponds to absence of RSA). The default is 0.3.
- **ibi_randomness** (*float*) – Float between 0 and 1. Determines how much random noise there is in the duration of each PPG wave (0 corresponds to absence of variation). The default is 0.1.
- **drift** (*float*) – Float between 0 and 1. Determines how pronounced the baseline drift (.05 Hz) is (0 corresponds to absence of baseline drift). The default is 1.

- **motion_amplitude** (*float*) – Float between 0 and 1. Determines how pronounced the motion artifact (0.5 Hz) is (0 corresponds to absence of motion artifact). The default is 0.1.
- **powerline_amplitude** (*float*) – Float between 0 and 1. Determines how pronounced the powerline artifact (50 Hz) is (0 corresponds to absence of powerline artifact). Note that `powerline_amplitude > 0` is only possible if 'sampling_rate' is ≥ 500 . The default is 0.1.
- **burst_amplitude** (*float*) – Float between 0 and 1. Determines how pronounced high frequency burst artifacts are (0 corresponds to absence of bursts). The default is 1.
- **burst_number** (*int*) – Determines how many high frequency burst artifacts occur. The default is 0.
- **show** (*bool*) – If true, returns a plot of the landmarks and interpolated PPG. Useful for debugging.
- **random_state** (*int*) – Seed for the random number generator. Keep it fixed for reproducible results.

Returns `ppg` (*array*) – A vector containing the PPG.

See also:

`ecg_simulate()`, `rsp_simulate()`, `eda_simulate()`, `emg_simulate()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> ppg = nk.ppg_simulate(duration=40, sampling_rate=500, heart_rate=75, random_
    ↳ state=42)
```

7.3 HRV

hrv (*peaks*, *sampling_rate=1000*, *show=False*)

Computes indices of Heart Rate Variability (HRV).

Computes HRV indices in the time-, frequency-, and nonlinear domain. Note that a minimum duration of the signal containing the peaks is recommended for some HRV indices to be meaningful. For instance, 1, 2 and 5 minutes of high quality signal are the recommended minima for HF, LF and LF/HF, respectively. See references for details.

Parameters

- **peaks** (*dict*) – Samples at which cardiac extrema (i.e., R-peaks, systolic peaks) occur. Dictionary returned by `ecg_findpeaks`, `ecg_peaks`, `ppg_findpeaks`, or `ppg_peaks`.
- **sampling_rate** (*int*, *optional*) – Sampling rate (Hz) of the continuous cardiac signal in which the peaks occur. Should be at least twice as high as the highest frequency in vhf. By default 1000.
- **show** (*bool*, *optional*) – If True, returns the plots that are generated for each of the domains.

Returns `DataFrame` – Contains HRV metrics from three domains: - frequency (see `hrv_frequency`) - time (see `hrv_time`) - non-linear (see ``hrv_nonlinear``)
<[https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.hrv.hrv_nonlinear`](https://neurokit2.readthedocs.io/en/latest/functions.html#neurokit2.hrv.hrv_nonlinear)>

See also:

`ecg_peaks()`, `ppg_peaks()`, `hrv_time()`, `hrv_frequency()`, `hrv_nonlinear()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Find peaks
>>> peaks, info = nk.ecg_peaks(data["ECG"], sampling_rate=100)
>>>
>>> # Compute HRV indices
>>> hrv_indices = nk.hrv(peaks, sampling_rate=100, show=True)
>>> hrv_indices
```

References

- Stein, P. K. (2002). Assessing heart rate variability from real-world Holter reports. Cardiac

electrophysiology review, 6(3), 239-244.

- Shaffer, F., & Ginsberg, J. P. (2017). An overview of heart rate variability metrics and norms.

Frontiers in public health, 5, 258.

hrv_frequency (*peaks*, *sampling_rate*=1000, *ulf*=0, 0.0033, *vlf*=0.0033, 0.04, *lf*=0.04, 0.15, *hf*=0.15, 0.4, *vhf*=0.4, 0.5, *psd_method*='welch', *show*=False, *silent*=True, ***kwargs*)
Computes frequency-domain indices of Heart Rate Variability (HRV).

Note that a minimum duration of the signal containing the peaks is recommended for some HRV indices to be meaningful. For instance, 1, 2 and 5 minutes of high quality signal are the recommended minima for HF, LF and LF/HF, respectively. See references for details.

Parameters

- **peaks** (*dict*) – Samples at which cardiac extrema (i.e., R-peaks, systolic peaks) occur. Dictionary returned by `ecg_findpeaks`, `ecg_peaks`, `ppg_findpeaks`, or `ppg_peaks`.
- **sampling_rate** (*int*, *optional*) – Sampling rate (Hz) of the continuous cardiac signal in which the peaks occur. Should be at least twice as high as the highest frequency in vhf. By default 1000.
- **ulf** (*tuple*, *optional*) – Upper and lower limit of the ultra-low frequency band. By default (0, 0.0033).
- **vlf** (*tuple*, *optional*) – Upper and lower limit of the very-low frequency band. By default (0.0033, 0.04).
- **lf** (*tuple*, *optional*) – Upper and lower limit of the low frequency band. By default (0.04, 0.15).
- **hf** (*tuple*, *optional*) – Upper and lower limit of the high frequency band. By default (0.15, 0.4).
- **vhf** (*tuple*, *optional*) – Upper and lower limit of the very-high frequency band. By default (0.4, 0.5).

- **psd_method** (*str*) – Method used for spectral density estimation. For details see `signal.signal_power`. By default “welch”.
- **silent** (*bool*) – If False, warnings will be printed. Default to True.
- **show** (*bool*) – If True, will plot the power in the different frequency bands.
- ****kwargs** (*optional*) – Other arguments.

Returns *DataFrame* – Contains frequency domain HRV metrics: - **ULF**: The spectral power density pertaining to ultra low frequency band i.e., .0 to .0033 Hz by default. - **VLF**: The spectral power density pertaining to very low frequency band i.e., .0033 to .04 Hz by default. - **LF**: The spectral power density pertaining to low frequency band i.e., .04 to .15 Hz by default. - **HF**: The spectral power density pertaining to high frequency band i.e., .15 to .4 Hz by default. - **VHF**: The variability, or signal power, in very high frequency i.e., .4 to .5 Hz by default. - **LFn**: The normalized low frequency, obtained by dividing the low frequency power by the total power. - **HFn**: The normalized high frequency, obtained by dividing the low frequency power by the total power. - **LnHF**: The log transformed HF.

See also:

`ecg_peaks()`, `ppg_peaks()`, `hrv_summary()`, `hrv_time()`, `hrv_nonlinear()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Find peaks
>>> peaks, info = nk.ecg_peaks(data["ECG"], sampling_rate=100)
>>>
>>> # Compute HRV indices
>>> hrv = nk.hrv_frequency(peaks, sampling_rate=100, show=True)
```

References

- Stein, P. K. (2002). Assessing heart rate variability from real-world Holter reports. *Cardiac electrophysiology review*, 6(3), 239-244.
- Shaffer, F., & Ginsberg, J. P. (2017). An overview of heart rate variability metrics and norms. *Frontiers in public health*, 5, 258.

hrv_nonlinear (*peaks, sampling_rate=1000, show=False*)
Computes nonlinear indices of Heart Rate Variability (HRV).

See references for details.

Parameters

- **peaks** (*dict*) – Samples at which cardiac extrema (i.e., R-peaks, systolic peaks) occur. Dictionary returned by `ecg_findpeaks`, `ecg_peaks`, `ppg_findpeaks`, or `ppg_peaks`.
- **sampling_rate** (*int, optional*) – Sampling rate (Hz) of the continuous cardiac signal in which the peaks occur. Should be at least twice as high as the highest frequency in vhf. By default 1000.

- **show** (*bool, optional*) – If True, will return a Poincaré plot, a scattergram, which plots each RR interval against the next successive one. The ellipse centers around the average RR interval. By default False.

Returns

DataFrame – Contains non-linear HRV metrics:

- **Characteristics of the Poincaré Plot Geometry:**
 - **SD1:** SD1 is a measure of the spread of RR intervals on the Poincaré plot perpendicular to the line of identity. It is an index of short-term RR interval fluctuations, i.e., beat-to-beat variability. It is equivalent (although on another scale) to RMSSD, and therefore it is redundant to report correlations with both (Ciccone, 2017).
 - **SD2:** SD2 is a measure of the spread of RR intervals on the Poincaré plot along the line of identity. It is an index of long-term RR interval fluctuations.
 - **SD1SD2:** the ratio between short and long term fluctuations of the RR intervals (SD1 divided by SD2).
 - **S:** Area of ellipse described by SD1 and SD2 ($\pi * SD1 * SD2$). It is proportional to $SD1SD2$.
 - **CSI:** The Cardiac Sympathetic Index (Toichi, 1997), calculated by dividing the longitudinal variability of the Poincaré plot ($4 * SD2$) by its transverse variability ($4 * SD1$).
 - **CVI:** The Cardiac Vagal Index (Toichi, 1997), equal to the logarithm of the product of longitudinal ($4 * SD2$) and transverse variability ($4 * SD1$).
 - **CSI_Modified:** The modified CSI (Jeppesen, 2014) obtained by dividing the square of the longitudinal variability by its transverse variability.
- **Indices of Heart Rate Asymmetry (HRA), i.e., asymmetry of the Poincaré plot** (Yan, 2017):
 - **GI:** Guzik's Index, defined as the distance of points above line of identity (LI) to LI divided by the distance of all points in Poincaré plot to LI except those that are located on LI.
 - **SI:** Slope Index, defined as the phase angle of points above LI divided by the phase angle of all points in Poincaré plot except those that are located on LI.
 - **AI:** Area Index, defined as the cumulative area of the sectors corresponding to the points that are located above LI divided by the cumulative area of sectors corresponding to all points in the Poincaré plot except those that are located on LI.

- **PI**: Porta's Index, defined as the number of points below LI divided by the total number of points in Poincaré plot except those that are located on LI.
- **SD1d** and **SD1a**: short-term variance of contributions of decelerations (prolongations of RR intervals) and accelerations (shortenings of RR intervals), respectively (Piskorski, 2011).
- **C1d** and **C1a**: the contributions of heart rate decelerations and accelerations to short-term HRV, respectively (Piskorski, 2011).
- **SD2d** and **SD2a**: long-term variance of contributions of decelerations (prolongations of RR intervals) and accelerations (shortenings of RR intervals), respectively (Piskorski, 2011).
- **C2d** and **C2a**: the contributions of heart rate decelerations and accelerations to long-term HRV, respectively (Piskorski, 2011).
- **SDNNd** and **SDNNa**: total variance of contributions of decelerations (prolongations of RR intervals) and accelerations (shortenings of RR intervals), respectively (Piskorski, 2011).
- **Cd** and **Ca**: the total contributions of heart rate decelerations and accelerations to HRV.
- **Indices of Heart Rate Fragmentation** (Costa, 2017):
 - **PIP**: Percentage of inflection points of the RR intervals series.
 - **IALS**: Inverse of the average length of the acceleration/deceleration segments.
 - **PSS**: Percentage of short segments.
 - **PAS**: Percentage of NN intervals in alternation segments.
- **Indices of Complexity**:
 - **ApEn**: The approximate entropy measure of HRV, calculated by *entropy_approximate()*.
 - **SampEn**: The sample entropy measure of HRV, calculated by *entropy_sample()*.

See also:

`ecg_peaks()`, `ppg_peaks()`, `hrv_frequency()`, `hrv_time()`, `hrv_summary()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Find peaks
>>> peaks, info = nk.ecg_peaks(data["ECG"], sampling_rate=100)
>>>
>>> # Compute HRV indices
```

(continues on next page)

(continued from previous page)

```
>>> hrv = nk.hrv_nonlinear(peaks, sampling_rate=100, show=True)
>>> hrv
```

References

- Yan, C., Li, P., Ji, L., Yao, L., Karmakar, C., & Liu, C. (2017). Area asymmetry of heart rate variability signal. *Biomedical engineering online*, 16(1), 112.
- Ciccone, A. B., Siedlik, J. A., Wecht, J. M., Deckert, J. A., Nguyen, N. D., & Weir, J. P. (2017). Reminder: RMSSD and SD1 are identical heart rate variability metrics. *Muscle & nerve*, 56(4), 674-678.
- Shaffer, F., & Ginsberg, J. P. (2017). An overview of heart rate variability metrics and norms. *Frontiers in public health*, 5, 258.
- Costa, M. D., Davis, R. B., & Goldberger, A. L. (2017). Heart rate fragmentation: a new approach to the analysis of cardiac interbeat interval dynamics. *Front. Physiol.* 8, 255 (2017).
- Jeppesen, J., Beniczky, S., Johansen, P., Sidenius, P., & Fuglsang-Frederiksen, A. (2014). Using Lorenz plot and Cardiac Sympathetic Index of heart rate variability for detecting seizures for patients with epilepsy. In 2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (pp. 4563-4566). IEEE.
- Piskorski, J., & Guzik, P. (2011). Asymmetric properties of long-term and total heart rate variability. *Medical & biological engineering & computing*, 49(11), 1289-1297.
- Stein, P. K. (2002). Assessing heart rate variability from real-world Holter reports. *Cardiac electrophysiology review*, 6(3), 239-244.
- Brennan, M. et al. (2001). Do Existing Measures of Poincaré Plot Geometry Reflect Nonlinear Features of Heart Rate Variability?. *IEEE Transactions on Biomedical Engineering*, 48(11), 1342-1347.
- Toichi, M., Sugiura, T., Murai, T., & Sengoku, A. (1997). A new method of assessing cardiac autonomic function and its comparison with spectral analysis and coefficient of variation of R-R interval. *Journal of the autonomic nervous system*, 62(1-2), 79-84.

hrv_time (*peaks*, *sampling_rate*=1000, *show*=False)

Computes time-domain indices of Heart Rate Variability (HRV).

See references for details.

Parameters

- **peaks** (*dict*) – Samples at which cardiac extrema (i.e., R-peaks, systolic peaks) occur. Dictionary returned by `ecg_findpeaks`, `ecg_peaks`, `ppg_findpeaks`, or `ppg_peaks`.
- **sampling_rate** (*int*, *optional*) – Sampling rate (Hz) of the continuous cardiac signal in which the peaks occur. Should be at least twice as high as the highest frequency in vhf. By default 1000.
- **show** (*bool*) – If True, will plot the distribution of R-R intervals.

Returns *DataFrame* – Contains time domain HRV metrics: - **RMSSD**: The square root of the mean of the sum of successive differences between adjacent RR intervals. It is equivalent (although on another scale) to SD1, and therefore it is redundant to report correlations with both (Ciccone, 2017). - **MeanNN**: The mean of the RR intervals. - **SDNN**: The standard deviation of the RR intervals. - **SDSD**: The standard deviation of the successive differences between RR intervals. - **CVNN**: The standard deviation of the RR intervals (SDNN) divided by the mean of the RR intervals (MeanNN). - **CVSD**: The root mean square of the sum of successive differences (RMSSD) divided by the mean of the RR intervals (MeanNN). - **MedianNN**: The median of the absolute values of the successive differences between RR intervals. - **MadNN**: The median absolute deviation of the RR intervals. - **HCVNN**: The median absolute deviation of the RR intervals (MadNN) divided by the median of the absolute differences of their successive differences (MedianNN). - **IQRNN**: The interquartile range (IQR) of the RR intervals. - **pNN50**: The proportion of RR intervals greater than 50ms, out of the total number of RR intervals. - **pNN20**: The proportion of RR intervals greater than 20ms, out of the total number of RR intervals. - **TINN**: A geometrical parameter of the HRV, or more specifically, the baseline width of the RR intervals distribution obtained by triangular interpolation, where the error of least squares determines the triangle. It is an approximation of the RR interval distribution. - **HTI**: The HRV triangular index, measuring the total number of RR intervals divided by the height of the RR intervals histogram.

See also:

`ecg_peaks()`, `ppg_peaks()`, `hrv_frequency()`, `hrv_summary()`, `hrv_nonlinear()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Find peaks
>>> peaks, info = nk.ecg_peaks(data["ECG"], sampling_rate=100)
>>>
>>> # Compute HRV indices
>>> hrv = nk.hrv_time(peaks, sampling_rate=100, show=True)
```

References

- Ciccone, A. B., Siedlik, J. A., Wecht, J. M., Deckert, J. A., Nguyen, N. D., & Weir, J. P. (2017). Reminder: RMSSD and SD1 are identical heart rate variability metrics. *Muscle & nerve*, 56(4), 674-678.
- Stein, P. K. (2002). Assessing heart rate variability from real-world Holter reports. *Cardiac electrophysiology review*, 6(3), 239-244.
- Shaffer, F., & Ginsberg, J. P. (2017). An overview of heart rate variability metrics and norms. *Frontiers in public health*, 5, 258.

7.4 RSP

Submodule for NeuroKit.

rsp_amplitude (*rsp_cleaned*, *peaks*, *troughs*=None, *interpolation_method*='monotone_cubic')

Compute respiratory amplitude.

Compute respiratory amplitude given the raw respiration signal and its extrema.

Parameters

- **rsp_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned respiration channel as returned by *rsp_clean()*.
- **peaks** (*list or array or DataFrame or Series or dict*) – The samples at which the inhalation peaks occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with *rsp_findpeaks()*.
- **troughs** (*list or array or DataFrame or Series or dict*) – The samples at which the inhalation troughs occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with *rsp_findpeaks()*.
- **interpolation_method** (*str*) – Method used to interpolate the amplitude between peaks. See *signal_interpolate()*. ‘monotone_cubic’ is chosen as the default interpolation method since it ensures monotone interpolation between data points (i.e., it prevents physiologically implausible “overshoots” or “undershoots” in the y-direction). In contrast, the widely used cubic spline interpolation does not ensure monotonicity.

Returns *array* – A vector containing the respiratory amplitude.

See also:

rsp_clean(), *rsp_peaks()*, *signal_rate()*, *rsp_process()*, *rsp_plot()*

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> rsp = nk.rsp_simulate(duration=90, respiratory_rate=15)
>>> cleaned = nk.rsp_clean(rsp, sampling_rate=1000)
>>> info, signals = nk.rsp_peaks(cleaned)
>>>
>>> amplitude = nk.rsp_amplitude(cleaned, signals)
>>> fig = nk.signal_plot(pd.DataFrame({"RSP": rsp, "Amplitude": amplitude}),
>>>                        subplots=True)
>>> fig
```

rsp_analyze (*data*, *sampling_rate*=1000, *method*='auto')

Performs RSP analysis on either epochs (event-related analysis) or on longer periods of data such as resting-state data.

Parameters

- **data** (*dict or DataFrame*) – A dictionary of epochs, containing one DataFrame per epoch, usually obtained via *epochs_create()*, or a DataFrame containing all epochs, usually obtained via *epochs_to_df()*. Can also take a DataFrame of processed signals from a longer period of data, typically generated by *rsp_process()* or *bio_process()*. Can also take a dict containing sets of separate periods of data.

- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **method** (*str*) – Can be one of ‘event-related’ for event-related analysis on epochs, or ‘interval-related’ for analysis on longer periods of data. Defaults to ‘auto’ where the right method will be chosen based on the mean duration of the data (‘event-related’ for duration under 10s).

Returns *DataFrame* – A dataframe containing the analyzed RSP features. If event-related analysis is conducted, each epoch is indicated by the *Label* column. See *rsp_eventrelated()* and *rsp_intervalrelated()* docstrings for details.

See also:

`bio_process()`, `rsp_process()`, `epochs_create()`, `rsp_eventrelated()`,
`rsp_intervalrelated()`

Examples

```
>>> import neurokit2 as nk

>>> # Example 1: Download the data for event-related analysis
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data for event-related analysis
>>> df, info = nk.bio_process(rsp=data["RSP"], sampling_rate=100)
>>> events = nk.events_find(data["Photosensor"], threshold_keep='below',
...                         event_conditions=["Negative", "Neutral", "Neutral",
-> "Negative"])
>>> epochs = nk.epochs_create(df, events, sampling_rate=100, epochs_start=-0.1,
-> epochs_end=1.9)
>>>
>>> # Analyze
>>> nk.rsp_analyze(epochs, sampling_rate=100)
>>>
>>> # Example 2: Download the resting-state data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Process the data
>>> df, info = nk.rsp_process(data["RSP"], sampling_rate=100)
>>>
>>> # Analyze
>>> nk.rsp_analyze(df, sampling_rate=100)
```

rsp_clean (*rsp_signal*, *sampling_rate*=1000, *method*='khodadad2018')
Preprocess a respiration (RSP) signal.

Clean a respiration signal using different sets of parameters, such as ‘khodadad2018’ (linear detrending followed by a fifth order 2Hz low-pass IIR Butterworth filter) or [BioSPPy](#) (second order 0.1 - 0.35 Hz bandpass Butterworth filter followed by a constant detrending).

Parameters

- **rsp_signal** (*Union[list, np.array, pd.Series]*) – The raw respiration channel (as measured, for instance, by a respiration belt).
- **sampling_rate** (*int*) – The sampling frequency of *rsp_signal* (in Hz, i.e., samples/second).

- **method** (*str*) – The processing pipeline to apply. Can be one of “khodadad2018” (default) or “biosppy”.

Returns *array* – Vector containing the cleaned respiratory signal.

See also:

`rsp_findpeaks()`, `signal_rate()`, `rsp_amplitude()`, `rsp_process()`, `rsp_plot()`

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=30, sampling_rate=50, noise=0.01)
>>> signals = pd.DataFrame({ "RSP_Raw": rsp,
...                          "RSP_Khodadad2018": nk.rsp_clean(rsp, sampling_
→rate=50, method="khodadad2018"),
...                          "RSP_BioSPPy": nk.rsp_clean(rsp, sampling_rate=50,
→method="biosppy")})
>>> fig = signals.plot()
>>> fig
```

References

- Khodadad et al. (2018)

rsp_eventrelated (*epochs*, *silent=False*)

Performs event-related RSP analysis on epochs.

Parameters

- **epochs** (*Union[dict, pd.DataFrame]*) – A dict containing one DataFrame per event/trial, usually obtained via `epochs_create()`, or a DataFrame containing all epochs, usually obtained via `epochs_to_df()`.
- **silent** (*bool*) – If True, silence possible warnings.

Returns *DataFrame* – A dataframe containing the analyzed RSP features for each epoch, with each epoch indicated by the *Label* column (if not present, by the *Index* column). The analyzed features consist of the following: - “*RSP_Rate_Max*”: the maximum respiratory rate after stimulus onset. - “*RSP_Rate_Min*”: the minimum respiratory rate after stimulus onset. - “*RSP_Rate_Mean*”: the mean respiratory rate after stimulus onset. - “*RSP_Rate_Max_Time*”: the time at which maximum respiratory rate occurs. - “*RSP_Rate_Min_Time*”: the time at which minimum respiratory rate occurs. - “*RSP_Amplitude_Max*”: the maximum respiratory amplitude after stimulus onset. - “*RSP_Amplitude_Min*”: the minimum respiratory amplitude after stimulus onset. - “*RSP_Amplitude_Mean*”: the mean respiratory amplitude after stimulus onset. - “*RSP_Phase*”: indication of whether the onset of the event concurs with respiratory inspiration (1) or expiration (0). - “*RSP_PhaseCompletion*”: indication of the stage of the current respiration phase (0 to 1) at the onset of the event.

See also:

`events_find()`, `epochs_create()`, `bio_process()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example with simulated data
>>> rsp, info = nk.rsp_process(nk.rsp_simulate(duration=120))
>>> epochs = nk.epochs_create(rsp, events=[5000, 10000, 15000], epochs_start=-0.1,
→ epochs_end=1.9)
>>>
>>> # Analyze
>>> rsp1 = nk.rsp_eventrelated(epochs)
>>> rsp1
>>>
>>> # Example with real data
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data
>>> df, info = nk.bio_process(rsp=data["RSP"], sampling_rate=100)
>>> events = nk.events_find(data["Photosensor"], threshold_keep='below',
...                          event_conditions=["Negative", "Neutral", "Neutral",
→ "Negative"])
>>> epochs = nk.epochs_create(df, events, sampling_rate=100, epochs_start=-0.1,
→ epochs_end=2.9)
>>>
>>> # Analyze
>>> rsp2 = nk.rsp_eventrelated(epochs)
>>> rsp2
```

rsp_findpeaks (*rsp_cleaned*, *sampling_rate*=1000, *method*='khodadad2018', *amplitude_min*=0.3)

Extract extrema in a respiration (RSP) signal.

Low-level function used by *rsp_peaks()* to identify inhalation peaks and exhalation troughs in a preprocessed respiration signal using different sets of parameters. See *rsp_peaks()* for details.

Parameters

- **rsp_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned respiration channel as returned by *rsp_clean()*.
- **sampling_rate** (*int*) – The sampling frequency of ‘rsp_cleaned’ (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of “khodadad2018” (default) or “biosppy”.
- **amplitude_min** (*float*) – Only applies if method is “khodadad2018”. Extrema that have a vertical distance smaller than (*outlier_threshold* * average vertical distance) to any direct neighbour are removed as false positive outliers. I.e., *outlier_threshold* should be a float with positive sign (the default is 0.3). Larger values of *outlier_threshold* correspond to more conservative thresholds (i.e., more extrema removed as outliers).

Returns **info** (*dict*) – A dictionary containing additional information, in this case the samples at which inhalation peaks and exhalation troughs occur, accessible with the keys “RSP_Peaks”, and “RSP_Troughs”, respectively.

See also:

rsp_clean(), *rsp_fixpeaks()*, *rsp_peaks()*, *signal_rate()*, *rsp_amplitude()*, *rsp_process()*, *rsp_plot()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=30, respiratory_rate=15)
>>> cleaned = nk.rsp_clean(rsp, sampling_rate=1000)
>>> info = nk.rsp_findpeaks(cleaned)
>>> fig = nk.events_plot([info["RSP_Peaks"], info["RSP_Troughs"]], cleaned)
>>> fig
```

rsp_fixpeaks (*peaks*, *troughs*=None)

Correct RSP peaks.

Low-level function used by *rsp_peaks()* to correct the peaks found by *rsp_findpeaks()*. Doesn't do anything for now for RSP. See *rsp_peaks()* for details.

Parameters

- **peaks** (*list or array or DataFrame or Series or dict*) – The samples at which the inhalation peaks occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with *rsp_findpeaks()*.
- **troughs** (*list or array or DataFrame or Series or dict*) – The samples at which the inhalation troughs occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with *rsp_findpeaks()*.

Returns **info** (*dict*) – A dictionary containing additional information, in this case the samples at which inhalation peaks and exhalation troughs occur, accessible with the keys “RSP_Peaks”, and “RSP_Troughs”, respectively.

See also:

rsp_clean(), *rsp_findpeaks()*, *rsp_peaks()*, *rsp_amplitude()*, *rsp_process()*, *rsp_plot()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=30, respiratory_rate=15)
>>> cleaned = nk.rsp_clean(rsp, sampling_rate=1000)
>>> info = nk.rsp_findpeaks(cleaned)
>>> info = nk.rsp_fixpeaks(info)
>>> fig = nk.events_plot([info["RSP_Peaks"], info["RSP_Troughs"]], cleaned)
>>> fig
```

rsp_intervalrelated (*data*, *sampling_rate*=1000)

Performs RSP analysis on longer periods of data (typically > 10 seconds), such as resting-state data.

Parameters

- **data** (*DataFrame or dict*) – A DataFrame containing the different processed signal(s) as different columns, typically generated by *rsp_process()* or *bio_process()*. Can also take a dict containing sets of separately processed DataFrames.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).

Returns *DataFrame* – A dataframe containing the analyzed RSP features. The analyzed features consist of the following: - “RSP_Rate_Mean”: the mean heart rate. -

“*RSP_Amplitude_Mean*”: the mean respiratory amplitude. - “*RSP_RRV*”: the different respiratory rate variability metrics. See *rsp_rrv()* docstrings for details.

See also:

`bio_process()`, `rsp_eventrelated()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Process the data
>>> df, info = nk.rsp_process(data["RSP"], sampling_rate=100)

>>> # Single dataframe is passed
>>> nk.rsp_intervalrelated(df)
>>>
>>> epochs = nk.epochs_create(df, events=[0, 15000], sampling_rate=100, epochs_
->end=150)
>>> nk.rsp_intervalrelated(epochs)
```

rsp_peaks (*rsp_cleaned*, *sampling_rate*=1000, *method*='khodadad2018', *amplitude_min*=0.3)

Identify extrema in a respiration (RSP) signal.

This function *rsp_findpeaks()* and *rsp_fixpeaks* to identify and process inhalation peaks and exhalation troughs in a preprocessed respiration signal using different sets of parameters, such as:

- `Khodadad et al. (2018)

<<https://iopscience.iop.org/article/10.1088/1361-6579/aad7e6/meta>>`_

- `BioSPPy

<<https://github.com/PIA-Group/BioSPPy/blob/master/biosppy/signals/resp.py>>`_

Parameters

- **rsp_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned respiration channel as returned by *rsp_clean()*.
- **sampling_rate** (*int*) – The sampling frequency of ‘rsp_cleaned’ (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of “khodadad2018” (default) or “biosppy”.
- **amplitude_min** (*float*) – Only applies if method is “khodadad2018”. Extrema that have a vertical distance smaller than (*outlier_threshold* * average vertical distance) to any direct neighbour are removed as false positive outliers. i.e., *outlier_threshold* should be a float with positive sign (the default is 0.3). Larger values of *outlier_threshold* correspond to more conservative thresholds (i.e., more extrema removed as outliers).

Returns

- **info** (*dict*) – A dictionary containing additional information, in this case the samples at which inhalation peaks and exhalation troughs occur, accessible with the keys “RSP_Peaks”, and “RSP_Troughs”, respectively.

- **peak_signal** (*DataFrame*) – A *DataFrame* of same length as the input signal in which occurrences of inhalation peaks and exhalation troughs are marked as “1” in lists of zeros with the same length as *rsp_cleaned*. Accessible with the keys “RSP_Peaks” and “RSP_Troughs” respectively.

See also:

`rsp_clean()`, `signal_rate()`, `rsp_findpeaks()`, `rsp_fixpeaks()`, `rsp_amplitude()`, `rsp_process()`, `rsp_plot()`

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> rsp = nk.rsp_simulate(duration=30, respiratory_rate=15)
>>> cleaned = nk.rsp_clean(rsp, sampling_rate=1000)
>>> peak_signal, info = nk.rsp_peaks(cleaned, sampling_rate=1000)
>>>
>>> data = pd.concat([pd.DataFrame({"RSP": rsp}), peak_signal], axis=1)
>>> fig = nk.signal_plot(data)
>>> fig
```

rsp_phase (*peaks, troughs=None, desired_length=None*)

Compute respiratory phase (inspiration and expiration).

Finds the respiratory phase, labelled as 1 for inspiration and 0 for expiration.

Parameters

- **peaks** (*list or array or DataFrame or Series or dict*) – The samples at which the inhalation peaks occur. If a dict or a *DataFrame* is passed, it is assumed that these containers were obtained with `rsp_findpeaks()`.
- **troughs** (*list or array or DataFrame or Series or dict*) – The samples at which the inhalation troughs occur. If a dict or a *DataFrame* is passed, it is assumed that these containers were obtained with `rsp_findpeaks()`.
- **desired_length** (*int*) – By default, the returned respiration rate has the same number of elements as *peaks*. If set to an integer, the returned rate will be interpolated between *peaks* over *desired_length* samples. Has no effect if a *DataFrame* is passed in as the *peaks* argument.

Returns signals (*DataFrame*) – A *DataFrame* of same length as *rsp_signal* containing the following columns: - “*RSP_Inspiration*”: breathing phase, marked by “1” for inspiration and “0” for expiration. - “*RSP_Phase_Completion*”: breathing phase completion, expressed in percentage (from 0 to 1), representing the stage of the current respiratory phase.

See also:

`rsp_clean()`, `rsp_peaks()`, `rsp_amplitude()`, `rsp_process()`, `rsp_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=30, respiratory_rate=15)
>>> cleaned = nk.rsp_clean(rsp, sampling_rate=1000)
>>> peak_signal, info = nk.rsp_peaks(cleaned)
>>>
>>> phase = nk.rsp_phase(peak_signal, desired_length=len(cleaned))
>>> fig = nk.signal_plot([rsp, phase], standardize=True)
>>> fig
```

rsp_plot (*rsp_signals*, *sampling_rate=None*)

Visualize respiration (RSP) data.

Parameters

- **rsp_signals** (*DataFrame*) – DataFrame obtained from *rsp_process()*.
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second).

Examples

```
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=90, respiratory_rate=15)
>>> rsp_signals, info = nk.rsp_process(rsp, sampling_rate=1000)
>>> fig = nk.rsp_plot(rsp_signals)
>>> fig
```

Returns *fig* – Figure representing a plot of the processed rsp signals.

See also:

rsp_process()

rsp_process (*rsp_signal*, *sampling_rate=1000*, *method='khodadad2018'*)

Process a respiration (RSP) signal.

Convenience function that automatically processes a respiration signal with one of the following methods:

- ‘Khodadad et al. (2018)

<<https://iopscience.iop.org/article/10.1088/1361-6579/aad7e6/meta>>`_

- ‘BioSPPy

<<https://github.com/PIA-Group/BioSPPy/blob/master/biosppy/signals/resp.py>>`_

Parameters

- **rsp_signal** (*Union[list, np.array, pd.Series]*) – The raw respiration channel (as measured, for instance, by a respiration belt).
- **sampling_rate** (*int*) – The sampling frequency of *rsp_signal* (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of “khodadad2018” (default) or “biosppy”.

Returns

- **signals** (*DataFrame*) – A *DataFrame* of same length as *rsp_signal* containing the following columns: - “*RSP_Raw*”: the raw signal. - “*RSP_Clean*”: the cleaned signal. - “*RSP_Peaks*”: the inhalation peaks marked as “1” in a list of zeros. - “*RSP_Troughs*”: the exhalation troughs marked as “1” in a list of zeros. - “*RSP_Rate*”: breathing rate interpolated between inhalation peaks. - “*RSP_Amplitude*”: breathing amplitude interpolated between inhalation peaks. - “*RSP_Phase*”: breathing phase, marked by “1” for inspiration and “0” for expiration. - “*RSP_PhaseCompletion*”: breathing phase completion, expressed in percentage (from 0 to 1), representing the stage of the current respiratory phase.
- **info** (*dict*) – A dictionary containing the samples at which inhalation peaks and exhalation troughs occur, accessible with the keys “*RSP_Peaks*”, and “*RSP_Troughs*”, respectively.

See also:

`rsp_clean()`, `rsp_findpeaks()`, `signal_rate()`, `rsp_amplitude()`, `rsp_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=90, respiratory_rate=15)
>>> signals, info = nk.rsp_process(rsp, sampling_rate=1000)
>>> fig = nk.rsp_plot(signals)
>>> fig
```

rsp_rate (*peaks*, *sampling_rate*=1000, *desired_length*=None, *interpolation_method*='monotone_cubic')

Calculate signal rate from a series of peaks.

This function can also be called either via `ecg_rate()`, `ppg_rate()` or `rsp_rate()` (aliases provided for consistency).

Parameters

- **peaks** (*Union[list, np.array, pd.DataFrame, pd.Series, dict]*) – The samples at which the peaks occur. If an array is passed in, it is assumed that it was obtained with `signal_findpeaks()`. If a *DataFrame* is passed in, it is assumed it is of the same length as the input signal in which occurrences of R-peaks are marked as “1”, with such containers obtained with e.g., `ecg_findpeaks()` or `rsp_findpeaks()`.
- **sampling_rate** (*int*) – The sampling frequency of the signal that contains peaks (in Hz, i.e., samples/second). Defaults to 1000.
- **desired_length** (*int*) – If left at the default None, the returned rate will have the same number of elements as peaks. If set to a value larger than the sample at which the last peak occurs in the signal (i.e., `peaks[-1]`), the returned rate will be interpolated between peaks over *desired_length* samples. To interpolate the rate over the entire duration of the signal, set *desired_length* to the number of samples in the signal. Cannot be smaller than or equal to the sample at which the last peak occurs in the signal. Defaults to None.
- **interpolation_method** (*str*) – Method used to interpolate the rate between peaks. See `signal_interpolate()`. ‘monotone_cubic’ is chosen as the default interpolation method since it ensures monotone interpolation between data points (i.e., it prevents physiologically implausible “overshoots” or “undershoots” in the y-direction). In contrast, the widely used cubic spline interpolation does not ensure monotonicity.

Returns *array* – A vector containing the rate.

See also:

`signal_findpeaks()`, `signal_fixpeaks()`, `signal_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, sampling_rate=1000, frequency=1)
>>> info = nk.signal_findpeaks(signal)
>>>
>>> rate = nk.signal_rate(peaks=info["Peaks"], desired_length=len(signal))
>>> fig = nk.signal_plot(rate)
>>> fig
```

rsp_rrv (*rsp_rate*, *peaks=None*, *sampling_rate=1000*, *show=False*, *silent=True*)

Computes time domain and frequency domain features for Respiratory Rate Variability (RRV) analysis.

Parameters

- **rsp_rate** (*array*) – Array containing the respiratory rate, produced by `signal_rate()`.
- **peaks** (*dict*) – The samples at which the inhalation peaks occur. Dict returned by `rsp_peaks()`. Defaults to None.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).
- **show** (*bool*) – If True, will return a Poincaré plot, a scattergram, which plots each breath-to-breath interval against the next successive one. The ellipse centers around the average breath-to-breath interval. Defaults to False.
- **silent** (*bool*) – If False, warnings will be printed. Default to True.

Returns *DataFrame* – DataFrame consisting of the computed RRV metrics, which includes: - “RRV_SDBB”: the standard deviation of the breath-to-breath intervals. - “RRV_RMSSD”: the root mean square of successive differences of the breath-to-breath intervals. - “RRV_SDSD”: the standard deviation of the successive differences between adjacent breath-to-breath intervals. - “RRV_BBx”: the number of successive interval differences that are greater than x seconds. - “RRV-pBBx”: the proportion of breath-to-breath intervals that are greater than x seconds, out of the total number of intervals. - “RRV_VLF”: spectral power density pertaining to very low frequency band i.e., 0 to .04 Hz by default. - “RRV_LF”: spectral power density pertaining to low frequency band i.e., .04 to .15 Hz by default. - “RRV_HF”: spectral power density pertaining to high frequency band i.e., .15 to .4 Hz by default. - “RRV_LFHF”: the ratio of low frequency power to high frequency power. - “RRV_LFn”: the normalized low frequency, obtained by dividing the low frequency power by the total power. - “RRV_HFn”: the normalized high frequency, obtained by dividing the low frequency power by total power. - “RRV_SD1”: SD1 is a measure of the spread of breath-to-breath intervals on the Poincaré plot perpendicular to the line of identity. It is an index of short-term variability. - “RRV_SD2”: SD2 is a measure of the spread of breath-to-breath intervals on the Poincaré plot along the line of identity. It is an index of long-term variability. - “RRV_SD2SD1”: the ratio between short and long term fluctuations of the breath-to-breath intervals (SD2 divided by SD1). - “RRV_ApEn”: the approximate entropy of RRV, calculated by `entropy_approximate()`. - “RRV_SampEn”: the sample entropy of RRV, calculated by `entropy_sample()`. - “RRV_DFA_1”: the “short-term” fluctuation value generated from Detrended Fluctuation Analysis i.e. the root mean square deviation from the fitted trend of the breath-to-breath intervals. Will only be computed if more than 160 breath cycles in the signal. - “RRV_DFA_2”: the long-term fluctuation value. Will only be computed if more than 640 breath cycles in the signal.

See also:

`signal_rate()`, `rsp_peaks()`, `signal_power()`, `entropy_sample()`,
`entropy_approximate()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> rsp = nk.rsp_simulate(duration=90, respiratory_rate=15)
>>> rsp, info = nk.rsp_process(rsp)
>>> rrv = nk.rsp_rrv(rsp, show=True)
```

References

- Soni, R., & Muniyandi, M. (2019). Breath rate variability: a novel measure to study the meditation effects. *International Journal of Yoga*, 12(1), 45.

rsp_simulate (*duration=10, length=None, sampling_rate=1000, noise=0.01, respiratory_rate=15, method='breathmetrics', random_state=None*)
 Simulate a respiratory signal.

Generate an artificial (synthetic) respiratory signal of a given duration and rate.

Parameters

- **duration** (*int*) – Desired length of duration (s).
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second).
- **length** (*int*) – The desired length of the signal (in samples).
- **noise** (*float*) – Noise level (amplitude of the laplace noise).
- **respiratory_rate** (*float*) – Desired number of breath cycles in one minute.
- **method** (*str*) – The model used to generate the signal. Can be ‘sinusoidal’ for a simulation based on a trigonometric sine wave that roughly approximates a single respiratory cycle. If ‘breathmetrics’ (default), will use an advanced model described [Noto, et al. \(2018\)](#).
- **random_state** (*int*) – Seed for the random number generator.

Returns *array* – Vector containing the respiratory signal.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> import neurokit2 as nk
>>>
>>> rsp1 = nk.rsp_simulate(duration=30, method="sinusoidal")
>>> rsp2 = nk.rsp_simulate(duration=30, method="breathmetrics")
>>> fig = pd.DataFrame({"RSP_Simple": rsp1, "RSP_Complex": rsp2}).
  ↳plot(subplots=True)
>>> fig
```

References

Noto, T., Zhou, G., Schuele, S., Templer, J., & Zelano, C. (2018). Automated analysis of breathing waveforms using BreathMetrics: A respiratory signal processing toolbox. *Chemical Senses*, 43(8), 583–597. <https://doi.org/10.1093/chemse/bjy045>

See also:

`rsp_clean()`, `rsp_findpeaks()`, `signal_rate()`, `rsp_process()`, `rsp_plot()`

7.5 EDA

Submodule for NeuroKit.

eda_analyze (*data*, *sampling_rate=1000*, *method='auto'*)

Performs EDA analysis on either epochs (event-related analysis) or on longer periods of data such as resting-state data.

Parameters

- **data** (*Union[dict, pd.DataFrame]*) – A dictionary of epochs, containing one DataFrame per epoch, usually obtained via `epochs_create()`, or a DataFrame containing all epochs, usually obtained via `epochs_to_df()`. Can also take a DataFrame of processed signals from a longer period of data, typically generated by `eda_process()` or `bio_process()`. Can also take a dict containing sets of separate periods of data.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **method** (*str*) – Can be one of ‘event-related’ for event-related analysis on epochs, or ‘interval-related’ for analysis on longer periods of data. Defaults to ‘auto’ where the right method will be chosen based on the mean duration of the data (‘event-related’ for duration under 10s).

Returns *DataFrame* – A dataframe containing the analyzed EDA features. If event-related analysis is conducted, each epoch is indicated by the *Label* column. See `eda_eventrelated()` and `eda_intervalrelated()` docstrings for details.

See also:

`bio_process()`, `eda_process()`, `epochs_create()`, `eda_eventrelated()`, `eda_intervalrelated()`

Examples

```
>>> import neurokit2 as nk

>>> # Example 1: Download the data for event-related analysis
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data for event-related analysis
>>> df, info = nk.bio_process(eda=data["EDA"], sampling_rate=100)
>>> events = nk.events_find(data["Photosensor"], threshold_keep='below',
...                         event_conditions=["Negative", "Neutral", "Neutral",
...     "Negative"])
>>> epochs = nk.epochs_create(df, events, sampling_rate=100, epochs_start=-0.1,
...     epochs_end=1.9)
```

(continues on next page)

(continued from previous page)

```

>>>
>>> # Analyze
>>> nk.eda_analyze(epochs, sampling_rate=100)
>>>
>>> # Example 2: Download the resting-state data
>>> data = nk.data("bio_resting_8min_100hz")
>>>
>>> # Process the data
>>> df, info = nk.eda_process(data["EDA"], sampling_rate=100)
>>>
>>> # Analyze
>>> nk.eda_analyze(df, sampling_rate=100)

```

eda_autocor (*eda_cleaned*, *sampling_rate=1000*, *lag=4*)

Computes autocorrelation measure of raw EDA signal i.e., the correlation between the time series data and a specified time-lagged version of itself.

Parameters

- **eda_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned EDA signal.
- **sampling_rate** (*int*) – The sampling frequency of raw EDA signal (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **lag** (*int*) – Time lag in seconds. Defaults to 4 seconds to avoid autoregressive correlations approaching 1, as recommended by Halem et al. (2020).

Returns *float* – Autocorrelation index of the eda signal.

See also:

`eda_simulate()`, `eda_clean()`

Examples

```

>>> import neurokit2 as nk
>>>
>>> # Simulate EDA signal
>>> eda_signal = nk.eda_simulate(duration=5, scr_number=5, drift=0.1)
>>> eda_cleaned = nk.eda_clean(eda_signal)
>>> cor = nk.eda_autocor(eda_cleaned)
>>> cor

```

References

- Halem, S., van Roekel, E., Kroencke, L., Kuper, N., & Denissen, J. (2020). Moments That Matter? On the Complexity of Using Triggers Based on Skin Conductance to Sample Arousing Events Within an Experience Sampling Framework. *European Journal of Personality*.

eda_changepoints (*eda_cleaned*)

Calculate the number of change points using of the skin conductance signal in terms of mean and variance. Defaults to an algorithm penalty of 10000, as recommended by Halem et al. (2020).

Parameters **eda_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned EDA signal.

Returns *float* – Number of changepoints in the

See also:

`eda_simulate()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Simulate EDA signal
>>> eda_signal = nk.eda_simulate(duration=5, scr_number=5, drift=0.1)
>>> eda_cleaned = nk.eda_clean(eda_signal)
>>> changepoints = nk.eda_changepoints(eda_cleaned)
>>> changepoints
```

References

- Halem, S., van Roekel, E., Kroencke, L., Kuper, N., & Denissen, J. (2020). Moments That Matter? On the Complexity of Using Triggers Based on Skin Conductance to Sample Arousing Events Within an Experience Sampling Framework. *European Journal of Personality*.

eda_clean (*eda_signal*, *sampling_rate*=1000, *method*='neurokit')

Preprocess Electrodermal Activity (EDA) signal.

Parameters

- **eda_signal** (*Union[list, np.array, pd.Series]*) – The raw EDA signal.
- **sampling_rate** (*int*) – The sampling frequency of *rsp_signal* (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of 'neurokit' (default) or 'biosppy'.

Returns *array* – Vector containing the cleaned EDA signal.

See also:

`eda_simulate()`, `eda_findpeaks()`, `eda_process()`, `eda_plot()`

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> eda = nk.eda_simulate(duration=30, sampling_rate=100, scr_number=10, noise=0.
↳ 01, drift=0.02)
>>> signals = pd.DataFrame({"EDA_Raw": eda,
...                         "EDA_BioSPPy": nk.eda_clean(eda, sampling_rate=100,
↳ method='biosppy'),
...                         "EDA_NeuroKit": nk.eda_clean(eda, sampling_rate=100,
↳ method='neurokit')})
>>> fig = signals.plot()
>>> fig
```

eda_eventrelated (*epochs*, *silent*=False)

Performs event-related EDA analysis on epochs.

Parameters

- **epochs** (*Union[dict, pd.DataFrame]*) – A dict containing one DataFrame per event/trial, usually obtained via `epochs_create()`, or a DataFrame containing all epochs, usually obtained via `epochs_to_df()`.
- **silent** (*bool*) – If True, silence possible warnings.

Returns

DataFrame – A dataframe containing the analyzed EDA features for each epoch, with each epoch indicated by the *Label* column (if not present, by the *Index* column). The analyzed features consist the following:

- *"EDA_SCR"*: indication of whether Skin Conductance Response (SCR) occurs following the event (1 if an SCR onset is present and 0 if absent) and if so, its corresponding peak amplitude, time of peak, rise and recovery time. If there is no occurrence of SCR, nans are displayed for the below features.
- *"EDA_Peak_Amplitude"*: the maximum amplitude of the phasic component of the signal.
- *"SCR_Peak_Amplitude"*: the peak amplitude of the first SCR in each epoch.
- *"SCR_Peak_Amplitude_Time"*: the timepoint of each first SCR peak amplitude.
- *"SCR_RiseTime"*: the risetime of each first SCR i.e., the time it takes for SCR to reach peak amplitude from onset.
- *"SCR_RecoveryTime"*: the half-recovery time of each first SCR i.e., the time it takes for SCR to decrease to half amplitude.

See also:

`events_find()`, `epochs_create()`, `bio_process()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example with simulated data
>>> eda = nk.eda_simulate(duration=15, scr_number=3)
>>>
>>> # Process data
>>> eda_signals, info = nk.eda_process(eda, sampling_rate=1000)
>>> epochs = nk.epochs_create(eda_signals, events=[5000, 10000, 15000], sampling_
↳ rate=1000,
...                               epochs_start=-0.1, epochs_end=1.9)
>>>
>>> # Analyze
>>> nk.eda_eventrelated(epochs)
>>>
>>> # Example with real data
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data
>>> df, info = nk.bio_process(eda=data["EDA"], sampling_rate=100)
>>> events = nk.events_find(data["Photosensor"], threshold_keep='below',
...                           event_conditions=["Negative", "Neutral", "Neutral",
↳ "Negative"])
>>> epochs = nk.epochs_create(df, events, sampling_rate=100, epochs_start=-0.1,
↳ epochs_end=6.9)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Analyze
>>> nk.eda_eventrelated(epochs)
```

eda_findpeaks (*eda_phasic*, *sampling_rate*=1000, *method*='neurokit', *amplitude_min*=0.1)

Identify Skin Conductance Responses (SCR) in Electrodermal Activity (EDA).

Low-level function used by *eda_peaks()* to identify Skin Conductance Responses (SCR) peaks in the phasic component of Electrodermal Activity (EDA) with different possible methods. See *eda_peaks()* for details.

Parameters

- **eda_phasic** (*Union[list, np.array, pd.Series]*) – The phasic component of the EDA signal (from *eda_phasic()*).
- **sampling_rate** (*int*) – The sampling frequency of the EDA signal (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of “neurokit” (default), “gamboa2008”, “kim2004” (the default in BioSPPy) or “vanhalem2020”.
- **amplitude_min** (*float*) – Only used if ‘method’ is ‘neurokit’ or ‘kim2004’. Minimum threshold by which to exclude SCRs (peaks) as relative to the largest amplitude in the signal.

Returns **info** (*dict*) – A dictionary containing additional information, in this case the amplitude of the SCR, the samples at which the SCR onset and the SCR peaks occur. Accessible with the keys “SCR_Amplitude”, “SCR_Onsets”, and “SCR_Peaks” respectively.

See also:

eda_simulate(), *eda_clean()*, *eda_phasic()*, *eda_fixpeaks()*, *eda_peaks()*, *eda_process()*, *eda_plot()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Get phasic component
>>> eda_signal = nk.eda_simulate(duration=30, scr_number=5, drift=0.1, noise=0)
>>> eda_cleaned = nk.eda_clean(eda_signal)
>>> eda = nk.eda_phasic(eda_cleaned)
>>> eda_phasic = eda["EDA_Phasic"].values
>>>
>>> # Find peaks
>>> gamboa2008 = nk.eda_findpeaks(eda_phasic, method="gamboa2008")
>>> kim2004 = nk.eda_findpeaks(eda_phasic, method="kim2004")
>>> neurokit = nk.eda_findpeaks(eda_phasic, method="neurokit")
>>> vanhalem2020 = nk.eda_findpeaks(eda_phasic, method="vanhalem2020")
>>> fig = nk.events_plot([gamboa2008["SCR_Peaks"], kim2004["SCR_Peaks"],
>>>                       vanhalem2020["SCR_Peaks"],
>>>                       neurokit["SCR_Peaks"]], eda_phasic)
>>> fig
```

References

- Gamboa, H. (2008). Multi-modal behavioral biometrics based on hci and electrophysiology. PhD Thesis Universidade.
- Kim, K. H., Bang, S. W., & Kim, S. R. (2004). Emotion recognition system using short-term monitoring of physiological signals. Medical and biological engineering and computing, 42(3), 419-427.
- van Halem, S., Van Roekel, E., Kroencke, L., Kuper, N., & Denissen, J. (2020). Moments That Matter? On the Complexity of Using Triggers Based on Skin Conductance to Sample Arousing Events Within an Experience Sampling Framework. European Journal of Personality.

eda_fixpeaks (*peaks, onsets=None, height=None*)

Correct Skin Conductance Responses (SCR) peaks.

Low-level function used by `eda_peaks()` to correct the peaks found by `eda_findpeaks()`. Doesn't do anything for now for EDA. See `eda_peaks()` for details.

Parameters

- **peaks** (*list or array or DataFrame or Series or dict*) – The samples at which the SCR peaks occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with `eda_findpeaks()`.
- **onsets** (*list or array or DataFrame or Series or dict*) – The samples at which the SCR onsets occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with `eda_findpeaks()`. Defaults to None.
- **height** (*list or array or DataFrame or Series or dict*) – The samples at which the amplitude of the SCR peaks occur. If a dict or a DataFrame is passed, it is assumed that these containers were obtained with `eda_findpeaks()`. Defaults to None.

Returns **info** (*dict*) – A dictionary containing additional information, in this case the amplitude of the SCR, the samples at which the SCR onset and the SCR peaks occur. Accessible with the keys “SCR_Amplitude”, “SCR_Onsets”, and “SCR_Peaks” respectively.

See also:

`eda_simulate()`, `eda_clean()`, `eda_phasic()`, `eda_findpeaks()`, `eda_peaks()`, `eda_process()`, `eda_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Get phasic component
>>> eda_signal = nk.eda_simulate(duration=30, scr_number=5, drift=0.1, noise=0)
>>> eda_cleaned = nk.eda_clean(eda_signal)
>>> eda = nk.eda_phasic(eda_cleaned)
>>> eda_phasic = eda["EDA_Phasic"].values
>>>
>>> # Find and fix peaks
>>> info = nk.eda_findpeaks(eda_phasic)
>>> info = nk.eda_fixpeaks(info)
>>>
>>> fig = nk.events_plot(info["SCR_Peaks"], eda_phasic)
>>> fig
```

eda_intervalrelated (*data*)

Performs EDA analysis on longer periods of data (typically > 10 seconds), such as resting-state data.

Parameters *data* (*Union[dict, pd.DataFrame]*) – A DataFrame containing the different processed signal(s) as different columns, typically generated by *eda_process()* or *bio_process()*. Can also take a dict containing sets of separately processed DataFrames.

Returns *DataFrame* – A dataframe containing the analyzed EDA features. The analyzed features consist of the following: - “*SCR_Peaks_N*”: the number of occurrences of Skin Conductance Response (SCR). - “*SCR_Peaks_Amplitude_Mean*”: the mean amplitude of the SCR peak occurrences.

See also:

bio_process(), *eda_eventrelated()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Download data
>>> data = nk.data("bio_resting_8min_100hz")
>>>
>>> # Process the data
>>> df, info = nk.eda_process(data["EDA"], sampling_rate=100)
>>>
>>> # Single dataframe is passed
>>> nk.eda_intervalrelated(df)
>>>
>>> epochs = nk.epochs_create(df, events=[0, 25300], sampling_rate=100, epochs_
->end=20)
>>> nk.eda_intervalrelated(epochs)
```

eda_peaks (*eda_phasic*, *sampling_rate=1000*, *method='neurokit'*, *amplitude_min=0.1*)

Identify Skin Conductance Responses (SCR) in Electrodermal Activity (EDA).

Identify Skin Conductance Responses (SCR) peaks in the phasic component of Electrodermal Activity (EDA) with different possible methods, such as:

- Gamboa, H. (2008)

<<http://www.lx.it.pt/~afred/pub/thesisHugoGamboa.pdf>> - Kim et al. (2004)

Parameters

- **eda_phasic** (*Union[list, np.array, pd.Series]*) – The phasic component of the EDA signal (from *eda_phasic()*).
- **sampling_rate** (*int*) – The sampling frequency of the EDA signal (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of “neurokit” (default), “gamboa2008” or “kim2004” (the default in BioSPPy).
- **amplitude_min** (*float*) – Only used if ‘method’ is ‘neurokit’ or ‘kim2004’. Minimum threshold by which to exclude SCRs (peaks) as relative to the largest amplitude in the signal.

Returns

- **info** (*dict*) – A dictionary containing additional information, in this case the amplitude of the SCR, the samples at which the SCR onset and the SCR peaks occur. Accessible with the keys “SCR_Amplitude”, “SCR_Onsets”, and “SCR_Peaks” respectively.
- **signals** (*DataFrame*) – A DataFrame of same length as the input signal in which occurrences of SCR peaks are marked as “1” in lists of zeros with the same length as *eda_cleaned*. Accessible with the keys “SCR_Peaks”.

See also:

`eda_simulate()`, `eda_clean()`, `eda_phasic()`, `eda_process()`, `eda_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Get phasic component
>>> eda_signal = nk.eda_simulate(duration=30, scr_number=5, drift=0.1, noise=0,
↳sampling_rate=100)
>>> eda_cleaned = nk.eda_clean(eda_signal, sampling_rate=100)
>>> eda = nk.eda_phasic(eda_cleaned, sampling_rate=100)
>>> eda_phasic = eda["EDA_Phasic"].values
>>>
>>> # Find peaks
>>> _, gamboa2008 = nk.eda_peaks(eda_phasic, method="gamboa2008")
>>> _, kim2004 = nk.eda_peaks(eda_phasic, method="kim2004")
>>> _, neurokit = nk.eda_peaks(eda_phasic, method="neurokit")
>>> nk.events_plot([gamboa2008["SCR_Peaks"], kim2004["SCR_Peaks"], neurokit["SCR_
↳Peaks"]], eda_phasic)
<Figure ...>
```

References

- Gamboa, H. (2008). Multi-modal behavioral biometrics based on hci and electrophysiology. PhD Thesis Universidade.
- Kim, K. H., Bang, S. W., & Kim, S. R. (2004). Emotion recognition system using short-term monitoring of physiological signals. Medical and biological engineering and computing, 42(3), 419-427.

eda_phasic (*eda_signal*, *sampling_rate=1000*, *method='highpass'*)

Decompose Electrodermal Activity (EDA) into Phasic and Tonic components.

Decompose the Electrodermal Activity (EDA) into two components, namely Phasic and Tonic, using different methods including cvxEDA (Greco, 2016) or Biopac’s Acqknowledge algorithms.

Parameters

- **eda_signal** (*Union[list, np.array, pd.Series]*) – The raw EDA signal.
- **sampling_rate** (*int*) – The sampling frequency of raw EDA signal (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **method** (*str*) – The processing pipeline to apply. Can be one of “cvxEDA”, “median”, “smoothmedian”, “highpass”, “biopac”, or “acqknowledge”.

Returns *DataFrame* – DataFrame containing the ‘Tonic’ and the ‘Phasic’ components as columns.

See also:

`eda_simulate()`, `eda_clean()`, `eda_peaks()`, `eda_process()`, `eda_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Simulate EDA signal
>>> eda_signal = nk.eda_simulate(duration=30, scr_number=5, drift=0.1)
>>>
>>> # Decompose using different algorithms
>>> cvxEDA = nk.eda_phasic(nk.standardize(eda_signal), method='cvxeda')
>>> smoothMedian = nk.eda_phasic(nk.standardize(eda_signal), method='smoothmedian')
>>> highpass = nk.eda_phasic(nk.standardize(eda_signal), method='highpass')
>>>
>>> data = pd.concat([cvxEDA.add_suffix('_cvxEDA'), smoothMedian.add_suffix('_SmoothMedian'),
...                  highpass.add_suffix('_Highpass')], axis=1)
>>> data["EDA_Raw"] = eda_signal
>>> fig = data.plot()
>>> fig
>>>
>>> eda_signal = nk.data("bio_eventrelated_100hz")["EDA"]
>>> data = nk.eda_phasic(nk.standardize(eda_signal), sampling_rate=100)
>>> data["EDA_Raw"] = eda_signal
>>> fig = nk.signal_plot(data, standardize=True)
>>> fig
```

References

- cvxEDA: <https://github.com/lciti/cvxEDA>.
- Greco, A., Valenza, G., & Scilingo, E. P. (2016). Evaluation of CDA and CvxEDA Models. In Advances in Electrodermal Activity Processing with Applications for Mental Health (pp. 35-43). Springer International Publishing.
- Greco, A., Valenza, G., Lanata, A., Scilingo, E. P., & Citi, L. (2016). cvxEDA: A convex optimization approach to electrodermal activity processing. IEEE Transactions on Biomedical Engineering, 63(4), 797-804.

eda_plot (*eda_signals*, *sampling_rate=None*)
Visualize electrodermal activity (EDA) data.

Parameters

- **eda_signals** (*DataFrame*) – DataFrame obtained from `eda_process()`.
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second). Defaults to None.

Returns *fig* – Figure representing a plot of the processed EDA signals.

Examples

```
>>> import neurokit2 as nk
>>>
>>> eda_signal = nk.eda_simulate(duration=30, scr_number=5, drift=0.1, noise=0,
→sampling_rate=250)
>>> eda_signals, info = nk.eda_process(eda_signal, sampling_rate=250)
>>> fig = nk.eda_plot(eda_signals)
>>> fig
```

See also:

`eda_process()`

eda_process (*eda_signal*, *sampling_rate*=1000, *method*='neurokit')
Process Electrodermal Activity (EDA).

Convenience function that automatically processes electrodermal activity (EDA) signal.

Parameters

- **eda_signal** (*Union[list, np.array, pd.Series]*) – The raw EDA signal.
- **sampling_rate** (*int*) – The sampling frequency of *rsp_signal* (in Hz, i.e., samples/second).
- **method** (*str*) – The processing pipeline to apply. Can be one of “biosppy” or “neurokit” (default).

Returns

- **signals** (*DataFrame*) – A DataFrame of same length as *eda_signal* containing the following columns:
 - “*EDA_Raw*”: the raw signal.
 - “*EDA_Clean*”: the cleaned signal.
 - “*EDA_Tonic*”: the tonic component of the signal, or the Tonic Skin Conductance Level (SCL).
 - “*EDA_Phasic*”: the phasic component of the signal, or the Phasic Skin Conductance Response (SCR).
 - “*SCR_Onsets*”: the samples at which the onsets of the peaks occur, marked as “1” in a list of zeros.
 - “*SCR_Peaks*”: the samples at which the peaks occur, marked as “1” in a list of zeros.
 - “*SCR_Height*”: the SCR amplitude of the signal including the Tonic component. Note that cumulative effects of close- occurring SCRs might lead to an underestimation of the amplitude.
 - “*SCR_Amplitude*”: the SCR amplitude of the signal excluding the Tonic component.
 - “*SCR_RiseTime*”: the time taken for SCR onset to reach peak amplitude within the SCR.
 - “*SCR_Recovery*”: the samples at which SCR peaks recover (decline) to half amplitude, marked as “1” in a list of zeros.
- **info** (*dict*) – A dictionary containing the information of each SCR peak (see `eda_findpeaks()`).

See also:

`eda_simulate()`, `eda_clean()`, `eda_phasic()`, `eda_findpeaks()`, `eda_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> eda_signal = nk.eda_simulate(duration=30, scr_number=5, drift=0.1, noise=0)
>>> signals, info = nk.eda_process(eda_signal, sampling_rate=1000)
>>> fig = nk.eda_plot(signals)
>>> fig
```

eda_simulate (*duration=10, length=None, sampling_rate=1000, noise=0.01, scr_number=1, drift=- 0.01, random_state=None*)

Simulate Electrodermal Activity (EDA) signal.

Generate an artificial (synthetic) EDA signal of a given duration and sampling rate.

Parameters

- **duration** (*int*) – Desired recording length in seconds.
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **length** (*int*) – The desired length of the signal (in samples). Defaults to None.
- **noise** (*float*) – Noise level (amplitude of the laplace noise). Defaults to 0.01.
- **scr_number** (*int*) – Desired number of skin conductance responses (SCRs), i.e., peaks. Defaults to 1.
- **drift** (*float or list*) – The slope of a linear drift of the signal. Defaults to -0.01.
- **random_state** (*int*) – Seed for the random number generator. Defaults to None.

Returns *array* – Vector containing the EDA signal.

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> eda = nk.eda_simulate(duration=10, scr_number=3)
>>> fig = nk.signal_plot(eda)
>>> fig
```

See also:

`ecg_simulate()`, `rsp_simulate()`, `emg_simulate()`, `ppg_simulate()`

References

- Bach, D. R., Flandin, G., Friston, K. J., & Dolan, R. J. (2010). Modelling event-related skin conductance responses. *International Journal of Psychophysiology*, 75(3), 349-356.

7.6 EMG

Submodule for NeuroKit.

emg_activation (*emg_amplitude=None, emg_cleaned=None, sampling_rate=1000, method='threshold', threshold='default', duration_min='default', **kwargs*)

Detects onset in EMG signal based on the amplitude threshold.

Parameters

- **emg_amplitude** (*array*) – At least one EMG-related signal. Either the amplitude of the EMG signal, obtained from `emg_amplitude()` for methods like ‘threshold’ or ‘mixture’), and / or the cleaned EMG signal (for methods like ‘pelt’).
- **emg_cleaned** (*array*) – At least one EMG-related signal. Either the amplitude of the EMG signal, obtained from `emg_amplitude()` for methods like ‘threshold’ or ‘mixture’), and / or the cleaned EMG signal (for methods like ‘pelt’).
- **sampling_rate** (*int*) – The sampling frequency of `emg_signal` (in Hz, i.e., samples/second).
- **method** (*str*) – The algorithm used to discriminate between activity and baseline. Can be one of ‘mixture’ (default) or ‘threshold’. If ‘mixture’, will use a Gaussian Mixture Model to categorize between the two states. If ‘threshold’, will consider as activated all points which amplitude is superior to the threshold.
- **threshold** (*float*) – If `method` is ‘mixture’, then it corresponds to the minimum probability required to be considered as activated (default to 0.33). If `method` is ‘threshold’, then it corresponds to the minimum amplitude to detect as onset. Defaults to one tenth of the standard deviation of `emg_amplitude`.
- **duration_min** (*float*) – The minimum duration of a period of activity or non-activity in seconds. If ‘default’, will be set to 0.05 (50 ms).
- **kwargs** (*optional*) – Other arguments.

Returns

- **info** (*dict*) – A dictionary containing additional information, in this case the samples at which the onsets, offsets, and periods of activations of the EMG signal occur, accessible with the key “EMG_Onsets”, “EMG_Offsets”, and “EMG_Activity” respectively.
- **activity_signal** (*DataFrame*) – A DataFrame of same length as the input signal in which occurrences of onsets, offsets, and activity (above the threshold) of the EMG signal are marked as “1” in lists of zeros with the same length as `emg_amplitude`. Accessible with the keys “EMG_Onsets”, “EMG_Offsets”, and “EMG_Activity” respectively.

See also:

`emg_simulate()`, `emg_clean()`, `emg_amplitude()`, `emg_process()`, `emg_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Simulate signal and obtain amplitude
>>> emg = nk.emg_simulate(duration=10, burst_number=3)
>>> emg_cleaned = nk.emg_clean(emg)
>>> emg_amplitude = nk.emg_amplitude(emg_cleaned)
>>>
>>> # Threshold method
>>> activity, info = nk.emg_activation(emg_amplitude=emg_amplitude, method=
→ "threshold")
>>> fig = nk.events_plot([info["EMG_Offsets"], info["EMG_Onsets"]], emg_cleaned)
>>> fig
>>>
>>> # Threshold method
>>> activity, info = nk.emg_activation(emg_cleaned=emg_cleaned, method="pelt")
>>> nk.signal_plot([emg_cleaned, activity["EMG_Activity"]])
>>> fig
```

References

- BioSPPy: <https://github.com/PIA-Group/BioSPPy/blob/master/biosppy/signals/emg.py>

emg_amplitude (*emg_cleaned*)

Compute electromyography (EMG) amplitude.

Compute electromyography amplitude given the cleaned respiration signal, done by calculating the linear envelope of the signal.

Parameters **emg_cleaned** (*Union[list, np.array, pd.Series]*) – The cleaned electromyography channel as returned by *emg_clean()*.

Returns *array* – A vector containing the electromyography amplitude.

See also:

emg_clean(), *emg_rate()*, *emg_process()*, *emg_plot()*

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> emg = nk.emg_simulate(duration=10, sampling_rate=1000, burst_number=3)
>>> cleaned = nk.emg_clean(emg, sampling_rate=1000)
>>>
>>> amplitude = nk.emg_amplitude(cleaned)
>>> fig = pd.DataFrame({"EMG": emg, "Amplitude": amplitude}).plot(subplots=True)
>>> fig
```

emg_analyze (*data, sampling_rate=1000, method='auto'*)

Performs EMG analysis on either epochs (event-related analysis) or on longer periods of data such as resting-state data.

Parameters

- **data** (*Union[dict, pd.DataFrame]*) – A dictionary of epochs, containing one DataFrame per epoch, usually obtained via `epochs_create()`, or a DataFrame containing all epochs, usually obtained via `epochs_to_df()`. Can also take a DataFrame of processed signals from a longer period of data, typically generated by `emg_process()` or `bio_process()`. Can also take a dict containing sets of separate periods of data.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second). Defaults to 1000Hz.
- **method** (*str*) – Can be one of ‘event-related’ for event-related analysis on epochs, or ‘interval-related’ for analysis on longer periods of data. Defaults to ‘auto’ where the right method will be chosen based on the mean duration of the data (‘event-related’ for duration under 10s).

Returns *DataFrame* – A dataframe containing the analyzed EMG features. If event-related analysis is conducted, each epoch is indicated by the *Label* column. See `emg_eventrelated()` and `emg_intervalrelated()` docstrings for details.

See also:

`bio_process()`, `emg_process()`, `epochs_create()`, `emg_eventrelated()`,
`emg_intervalrelated()`

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd

>>> # Example with simulated data
>>> emg = nk.emg_simulate(duration=20, sampling_rate=1000, burst_number=3)
>>> emg_signals, info = nk.emg_process(emg, sampling_rate=1000)
>>> epochs = nk.epochs_create(emg_signals, events=[3000, 6000, 9000], sampling_
→rate=1000,
...                           epochs_start=-0.1, epochs_end=1.9)
>>>
>>> # Event-related analysis
>>> nk.emg_analyze(epochs, method="event-related")
>>>
>>> # Interval-related analysis
>>> nk.emg_analyze(emg_signals, method="interval-related")
```

emg_clean (*emg_signal*, *sampling_rate=1000*)

Preprocess an electromyography (emg) signal.

Clean an EMG signal using a set of parameters, such as: in ``BioSPPy <https://github.com/PIA-Group/BioSPPy/blob/e65da30f6379852ecb98f8e2e0c9b4b5175416c3/biosppy/signals/emg.py>``: fourth order 100 Hz highpass Butterworth filter followed by a constant detrending.

Parameters

- **emg_signal** (*Union[list, np.array, pd.Series]*) – The raw EMG channel.
- **sampling_rate** (*int*) – The sampling frequency of *emg_signal* (in Hz, i.e., samples/second). Defaults to 1000.

Returns *array* – Vector containing the cleaned EMG signal.

See also:

`emg_amplitude()`, `emg_process()`, `emg_plot()`

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> emg = nk.emg_simulate(duration=10, sampling_rate=1000)
>>> signals = pd.DataFrame({"EMG_Raw": emg, "EMG_Cleaned":nk.emg_clean(emg,
→sampling_rate=1000)})
>>> fig = signals.plot()
>>> fig
```

emg_eventrelated(*epochs*, *silent=False*)

Performs event-related EMG analysis on epochs.

Parameters

- **epochs** (*Union[dict, pd.DataFrame]*) – A dict containing one DataFrame per event/trial, usually obtained via *epochs_create()*, or a DataFrame containing all epochs, usually obtained via *epochs_to_df()*.
- **silent** (*bool*) – If True, silence possible warnings.

Returns

DataFrame – A dataframe containing the analyzed EMG features for each epoch, with each epoch indicated by the *Label* column (if not present, by the *Index* column). The analyzed features consist of the following:

- *"EMG_Activation"*: indication of whether there is muscular activation following the onset

of the event (1 if present, 0 if absent) and if so, its corresponding amplitude features and the number of activations in each epoch. If there is no activation, nans are displayed for the below features. - *"EMG_Amplitude_Mean"*: the mean amplitude of the activity. - *"EMG_Amplitude_Max"*: the maximum amplitude of the activity. - *"EMG_Amplitude_Max_Time"*: the time of maximum amplitude. - *"EMG_Bursts"*: the number of activations, or bursts of activity, within each epoch.

See also:

emg_simulate(), *emg_process()*, *events_find()*, *epochs_create()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example with simulated data
>>> emg = nk.emg_simulate(duration=20, sampling_rate=1000, burst_number=3)
>>> emg_signals, info = nk.emg_process(emg, sampling_rate=1000)
>>> epochs = nk.epochs_create(emg_signals, events=[3000, 6000, 9000], sampling_
→rate=1000,
...                               epochs_start=-0.1, epochs_end=1.9)
>>> nk.emg_eventrelated(epochs)
```

emg_intervalrelated(*data*)

Performs EMG analysis on longer periods of data (typically > 10 seconds), such as resting-state data.

Parameters *data* (*Union[dict, pd.DataFrame]*) – A DataFrame containing the different processed signal(s) as different columns, typically generated by *emg_process()* or *bio_process()*. Can also take a dict containing sets of separately processed DataFrames.

Returns *DataFrame* – A dataframe containing the analyzed EMG features. The analyzed features consist of the following: - “*EMG_Activation_N*”: the number of bursts of muscular activity. - “*EMG_Amplitude_Mean*”: the mean amplitude of the muscular activity.

See also:

`bio_process()`, `emg_eventrelated()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example with simulated data
>>> emg = nk.emg_simulate(duration=40, sampling_rate=1000, burst_number=3)
>>> emg_signals, info = nk.emg_process(emg, sampling_rate=1000)
>>>
>>> # Single dataframe is passed
>>> nk.emg_intervalrelated(emg_signals)
>>>
>>> epochs = nk.epochs_create(emg_signals, events=[0, 20000], sampling_rate=1000,
→ epochs_end=20)
>>> nk.emg_intervalrelated(epochs)
```

emg_plot (*emg_signals*, *sampling_rate=None*)

Visualize electromyography (EMG) data.

Parameters

- **emg_signals** (*DataFrame*) – DataFrame obtained from *emg_process()*.
- **sampling_rate** (*int*) – The sampling frequency of the EMG (in Hz, i.e., samples/second). Needs to be supplied if the data should be plotted over time in seconds. Otherwise the data is plotted over samples. Defaults to None.

Returns *fig* – Figure representing a plot of the processed emg signals.

Examples

```
>>> import neurokit2 as nk
>>>
>>> emg = nk.emg_simulate(duration=10, sampling_rate=1000, burst_number=3)
>>> emg_signals, _ = nk.emg_process(emg, sampling_rate=1000)
>>> fig = nk.emg_plot(emg_signals)
>>> fig
```

See also:

`ecg_process()`

emg_process (*emg_signal*, *sampling_rate=1000*)

Process a electromyography (EMG) signal.

Convenience function that automatically processes an electromyography signal.

Parameters

- **emg_signal** (*Union[list, np.array, pd.Series]*) – The raw electromyography channel.
- **sampling_rate** (*int*) – The sampling frequency of *emg_signal* (in Hz, i.e., samples/second).

Returns

- **signals** (*DataFrame*) – A DataFrame of same length as *emg_signal* containing the following columns: - “*EMG_Raw*”: the raw signal. - “*EMG_Clean*”: the cleaned signal. - “*EMG_Amplitude*”: the signal amplitude, or the activation level of the signal. - “*EMG_Activity*”: the activity of the signal for which amplitude exceeds the threshold specified, marked as “1” in a list of zeros. - “*EMG_Onsets*”: the onsets of the amplitude, marked as “1” in a list of zeros. - “*EMG_Offsets*”: the offsets of the amplitude, marked as “1” in a list of zeros.
- **info** (*dict*) – A dictionary containing the information of each amplitude onset, offset, and peak activity (see *emg_activation()*).

See also:

emg_clean(), *emg_amplitude()*, *emg_plot()*

Examples

```
>>> import neurokit2 as nk
>>>
>>> emg = nk.emg_simulate(duration=10, sampling_rate=1000, burst_number=3)
>>> signals, info = nk.emg_process(emg, sampling_rate=1000)
>>> fig = nk.emg_plot(signals)
>>> fig
```

emg_simulate (*duration=10, length=None, sampling_rate=1000, noise=0.01, burst_number=1, burst_duration=1.0, random_state=42*)
Simulate an EMG signal.

Generate an artificial (synthetic) EMG signal of a given duration and sampling rate.

Parameters

- **duration** (*int*) – Desired recording length in seconds.
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second).
- **length** (*int*) – The desired length of the signal (in samples).
- **noise** (*float*) – Noise level (gaussian noise).
- **burst_number** (*int*) – Desired number of bursts of activity (active muscle periods).
- **burst_duration** (*float or list*) – Duration of the bursts. Can be a float (each burst will have the same duration) or a list of durations for each bursts.
- **random_state** (*int*) – Seed for the random number generator.

Returns *array* – Vector containing the EMG signal.

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> emg = nk.emg_simulate(duration=10, burst_number=3)
>>> fig = nk.signal_plot(emg)
>>> fig
```

See also:

`ecg_simulate()`, `rsp_simulate()`, `eda_simulate()`, `ppg_simulate()`

References

This function is based on [this script](#).

7.7 EEG

Submodule for NeuroKit.

mne_channel_add(*raw*, *channel*, *channel_type*=None, *channel_name*=None, *sync_index_raw*=0, *sync_index_channel*=0)

Add channel as array to MNE.

Add a channel to a mne's Raw m/eeg file. It will basically synchronize the channel to the eeg data following a particular index and add it.

Parameters

- **raw** (*mne.io.Raw*) – Raw EEG data from MNE.
- **channel** (*list or array*) – The signal to be added.
- **channel_type** (*str*) – Channel type. Currently supported fields are 'ecg', 'bio', 'stim', 'eog', 'misc', 'seeg', 'ecog', 'mag', 'eeg', 'ref_meg', 'grad', 'emg', 'hbr' or 'hbo'.
- **channel_name** (*str*) – Desired channel name.
- **sync_index_raw** (*int or list*) – An index (e.g., the onset of the same event marked in the same signal), in the raw data, by which to align the two inputs. This can be used in case the EEG data and the channel to add do not have the same onsets and must be aligned through some common event.
- **sync_index_channel** (*int or list*) – An index (e.g., the onset of the same event marked in the same signal), in the channel to add, by which to align the two inputs. This can be used in case the EEG data and the channel to add do not have the same onsets and must be aligned through some common event.

Returns *mne.io.Raw* – Raw data in FIF format.

Example

```
>>> import neurokit2 as nk
>>> import mne
>>>
>>> # Let's say that the 42nd sample point in the EEG correspond to the 333rd_
    ↳point in the ECG
>>> event_index_in_eeg = 42
>>> event_index_in_ecg = 333
>>>
>>> raw = mne.io.read_raw_fif(mne.datasets.sample.data_path() + '/MEG/sample/
    ↳sample_audvis_raw.fif',
    ...                       preload=True)
>>> ecg = nk.ecg_simulate(length=170000)
>>>
>>> raw = nk.mne_channel_add(raw, ecg, sync_index_raw=event_index_in_eeg,
    ...                       sync_index_channel=event_index_in_ecg, channel_type=
    ↳"ecg")
```

mne_channel_extract (*raw, what, name=None*)

Channel array extraction from MNE.

Select one or several channels by name and returns them in a dataframe.

Parameters

- **raw** (*mne.io.Raw*) – Raw EEG data.
- **what** (*str or list*) – Can be ‘MEG’, which will extract all MEG channels, ‘EEG’, which will extract all EEG channels, or ‘EOG’, which will extract all EOG channels (that is, if channel names are named with prefixes of their type e.g., ‘EEG 001’ etc. or ‘EOG 061’). Provide exact a single or a list of channel’s name(s) if not (e.g., [‘124’, ‘125’]).
- **name** (*str or list*) – Useful only when extracting one channel. Can also take a list of names for renaming multiple channels, Otherwise, defaults to None.

Returns *DataFrame* – A DataFrame or Series containing the channel(s).

Example

```
>>> import neurokit2 as nk
>>> import mne
>>>
>>> raw = mne.io.read_raw_fif(mne.datasets.sample.data_path() +
    ...                       '/MEG/sample/sample_audvis_raw.fif', preload=True)
>>>
>>> raw_channel = nk.mne_channel_extract(raw, what=["EEG 060", "EEG 055"], name=[
    ↳'060', '055'])
>>> eeg_channels = nk.mne_channel_extract(raw, "EEG")
>>> eog_channels = nk.mne_channel_extract(raw, what='EOG', name='EOG')
```

7.8 Signal Processing

Submodule for NeuroKit.

signal_autocor (*signal*, *lag=None*, *normalize=True*)

Auto-correlation of a 1-dimensional sequences.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – Vector of values.
- **normalize** (*bool*) – Normalize the autocorrelation output.
- **lag** (*int*) – Time lag. If specified, one value of autocorrelation between signal with its lag self will be returned.

Returns *r* – The cross-correlation of the signal with itself at different time lags. Minimum time lag is 0, maximum time lag is the length of the signal. Or a correlation value at a specific lag if lag is not None.

Examples

```
>>> import neurokit2 as nk
>>>
>>> x = [1, 2, 3, 4, 5]
>>> autocor = nk.signal_autocor(x)
>>> autocor
```

signal_binarize (*signal*, *method='threshold'*, *threshold='auto'*)

Binarize a continuous signal.

Convert a continuous signal into zeros and ones depending on a given threshold.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **method** (*str*) – The algorithm used to discriminate between the two states. Can be one of 'mixture' (default) or 'threshold'. If 'mixture', will use a Gaussian Mixture Model to categorize between the two states. If 'threshold', will consider as activated all points which value is superior to the threshold.
- **threshold** (*float*) – If *method* is 'mixture', then it corresponds to the minimum probability required to be considered as activated (if 'auto', then 0.5). If *method* is 'threshold', then it corresponds to the minimum amplitude to detect as onset. If "auto", takes the value between the max and the min.

Returns *list* – A list or array depending on the type passed.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = np.cos(np.linspace(start=0, stop=20, num=1000))
>>> binary = nk.signal_binarize(signal)
>>> fig = pd.DataFrame({"Raw": signal, "Binary": binary}).plot()
>>> fig
```

signal_changepoints (*signal*, *change*='meanvar', *penalty*=None, *show*=False)
Change Point Detection.

Only the PELT method is implemented for now.

Parameters

- **signal** (*Union*[list, np.array, pd.Series]) – Vector of values.
- **change** (*str*) – Can be one of “meanvar” (default), “mean” or “var”.
- **penalty** (*float*) – The algorithm penalty. Default to `np.log(len(signal))`.
- **show** (*bool*) – Defaults to False.

Returns

- *Array* – Values indicating the samples at which the changepoints occur.
- *Fig* – Figure of plot of signal with markers of changepoints.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.emg_simulate(burst_number=3)
>>> fig = nk.signal_changepoints(signal, change="var", show=True)
>>> fig
```

References

- Killick, R., Fearnhead, P., & Eckley, I. A. (2012). Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500), 1590-1598.

signal_decompose (*signal*, *method*='emd', *n_components*=None, ***kwargs*)
Decompose a signal.

Signal decomposition into different sources using different methods, such as Empirical Mode Decomposition (EMD) or Singular spectrum analysis (SSA)-based signal separation method.

The extracted components can then be recombined into meaningful sources using `signal_recompose()`.

Parameters

- **signal** (*Union*[list, np.array, pd.Series]) – Vector of values.
- **method** (*str*) – The decomposition method. Can be one of ‘emd’ or ‘ssa’.

- **n_components** (*int*) – Number of components to extract. Only used for ‘ssa’ method. If None, will default to 50.
- ****kwargs** – Other arguments passed to other functions.

Returns *Array* – Components of the decomposed signal.

See also:

`signal_recompose()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Create complex signal
>>> signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01) # High freq
>>> signal += 3 * nk.signal_simulate(duration=10, frequency=3, noise=0.01) # Higher freq
>>> signal += 3 * np.linspace(0, 2, len(signal)) # Add baseline and trend
>>> signal += 2 * nk.signal_simulate(duration=10, frequency=0.1, noise=0)
>>>
>>> nk.signal_plot(signal)
>>>
>>> # EMD method
>>> components = nk.signal_decompose(signal, method="emd")
>>> fig = nk.signal_plot(components) # Visualize components
>>> fig
>>>
>>> # SSA method
>>> components = nk.signal_decompose(signal, method="ssa")
>>> fig = nk.signal_plot(components) # Visualize components
>>> fig
```

signal_dettrend (*signal*, *method*='polynomial', *order*=1, *regularization*=500, *alpha*=0.75, *window*=1.5, *stepsize*=0.02)

Polynomial detrending of signal.

Apply a baseline (order = 0), linear (order = 1), or polynomial (order > 1) detrending to the signal (i.e., removing a general trend). One can also use other methods, such as smoothness priors approach described by Tarvainen (2002) or LOESS regression, but these scale badly for long signals.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **method** (*str*) – Can be one of ‘polynomial’ (default; traditional detrending of a given order) or ‘tarvainen2002’ to use the smoothness priors approach described by Tarvainen (2002) (mostly used in HRV analyses as a lowpass filter to remove complex trends), ‘loess’ for LOESS smoothing trend removal or ‘locreg’ for local linear regression (the ‘runline’ algorithm from chronux).
- **order** (*int*) – Only used if *method* is ‘polynomial’. The order of the polynomial. 0, 1 or > 1 for a baseline (‘constant detrend’, i.e., remove only the mean), linear (remove the linear trend) or polynomial detrending, respectively. Can also be ‘auto’, in which case it will attempt to find the optimal order to minimize the RMSE.
- **regularization** (*int*) – Only used if *method*=‘tarvainen2002’. The regularization parameter (default to 500).

- **alpha** (*float*) – Only used if *method* is 'loess'. The parameter which controls the degree of smoothing.
- **window** (*float*) – Only used if *method* is 'locreg'. The detrending 'window' should correspond to the desired low frequency band to remove multiplied by the sampling rate (for instance, 1.5×1000 will remove frequencies below 1.5Hz for a signal sampled at 1000Hz).
- **stepsize** (*float*) – Only used if *method* is 'locreg'. Simialrly to 'window', 'stepsize' should also be multiplied by the sampling rate.

Returns *array* – Vector containing the detrended signal.

See also:

`signal_filter()`, `fit_loess()`

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>> import matplotlib.pyplot as plt
>>>
>>> # Simulate signal with low and high frequency
>>> signal = nk.signal_simulate(frequency=[0.1, 2], amplitude=[2, 0.5], sampling_
↳rate=100)
>>> signal = signal + (3 + np.linspace(0, 6, num=len(signal))) # Add baseline_
↳and linear trend
>>>
>>> # Apply detrending algorithms
>>> baseline = nk.signal_detrend(signal, order=0) # Constant detrend (removes_
↳the mean)
>>> linear = nk.signal_detrend(signal, order=1) # Linear detrend
>>> quadratic = nk.signal_detrend(signal, order=2) # Quadratic detrend
>>> cubic = nk.signal_detrend(signal, order=3) # Cubic detrend
>>> poly10 = nk.signal_detrend(signal, order=10) # Linear detrend (10th order)
>>> tarvainen = nk.signal_detrend(signal, method='tarvainen2002') # Tarvainen_
↳(2002) method
>>> loess = nk.signal_detrend(signal, method='loess') # LOESS detrend (smooth_
↳removal)
>>> locreg = nk.signal_detrend(signal, method='locreg',
...                             window=1.5*100, stepsize=0.02*100) # Local_
↳regression (100Hz)
>>>
>>> # Visualize different methods
>>> axes = pd.DataFrame({"Original signal": signal,
...                      "Baseline": baseline,
...                      "Linear": linear,
...                      "Quadratic": quadratic,
...                      "Cubic": cubic,
...                      "Polynomial (10th)": poly10,
...                      "Tarvainen": tarvainen,
...                      "LOESS": loess,
...                      "Local Regression": locreg}).plot(subplots=True)
>>> # Plot horizontal lines to better visualize the detrending
>>> for subplot in axes:
...     subplot.axhline(y=0, color='k', linestyle='--')
```


References

- Tarvainen, M. P., Ranta-Aho, P. O., & Karjalainen, P. A. (2002). An advanced detrending method

with application to HRV analysis. IEEE Transactions on Biomedical Engineering, 49(2), 172-175. <<https://ieeexplore.ieee.org/document/979357>>`_

```
signal_distort (signal,      sampling_rate=1000,      noise_shape='laplace',      noise_amplitude=0,
                  noise_frequency=100, powerline_amplitude=0, powerline_frequency=50, arti-
                  facts_amplitude=0, artifacts_frequency=100, artifacts_number=5, linear_drift=False,
                  random_state=None, silent=False)
```

Signal distortion.

Add noise of a given frequency, amplitude and shape to a signal.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).
- **noise_shape** (*str*) – The shape of the noise. Can be one of ‘laplace’ (default) or ‘gaussian’.
- **noise_amplitude** (*float*) – The amplitude of the noise (the scale of the random function, relative to the standard deviation of the signal).
- **noise_frequency** (*float*) – The frequency of the noise (in Hz, i.e., samples/second).
- **powerline_amplitude** (*float*) – The amplitude of the powerline noise (relative to the standard deviation of the signal).
- **powerline_frequency** (*float*) – The frequency of the powerline noise (in Hz, i.e., samples/second).
- **artifacts_amplitude** (*float*) – The amplitude of the artifacts (relative to the standard deviation of the signal).
- **artifacts_frequency** (*int*) – The frequency of the artifacts (in Hz, i.e., samples/second).
- **artifacts_number** (*int*) – The number of artifact bursts. The bursts have a random duration between 1 and 10% of the signal duration.
- **linear_drift** (*bool*) – Whether or not to add linear drift to the signal.
- **random_state** (*int*) – Seed for the random number generator. Keep it fixed for reproducible results.
- **silent** (*bool*) – Whether or not to display warning messages.

Returns *array* – Vector containing the distorted signal.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, frequency=0.5)
>>>
>>> # Noise
>>> noise = pd.DataFrame({"Freq100": nk.signal_distort(signal, noise_
    ↳frequency=200),
...                       "Freq50": nk.signal_distort(signal, noise_frequency=50),
...                       "Freq10": nk.signal_distort(signal, noise_frequency=10),
...                       "Freq5": nk.signal_distort(signal, noise_frequency=5),
...                       "Raw": signal}).plot()
>>> noise
>>>
>>> # Artifacts
>>> artifacts = pd.DataFrame({"1Hz": nk.signal_distort(signal, noise_amplitude=0,
    ↳artifacts_frequency=1,
    ↳artifacts_amplitude=0.5),
...                          "5Hz": nk.signal_distort(signal, noise_amplitude=0,
    ↳artifacts_frequency=5,
    ↳artifacts_amplitude=0.2),
...                          "Raw": signal}).plot()
>>> artifacts
```

signal_filter (*signal*, *sampling_rate*=1000, *lowcut*=None, *highcut*=None, *method*='butterworth', *order*=2, *window_size*='default', *powerline*=50)

Filter a signal using 'butterworth', 'fir' or 'savgol' filters.

Apply a lowpass (if 'highcut' frequency is provided), highpass (if 'lowcut' frequency is provided) or bandpass (if both are provided) filter to the signal.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values. or “bandstop”.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).
- **lowcut** (*float*) – Lower cutoff frequency in Hz. The default is None.
- **highcut** (*float*) – Upper cutoff frequency in Hz. The default is None.
- **method** (*str*) – Can be one of 'butterworth', 'fir', 'bessel' or 'savgol'. Note that for Butterworth, the function uses the SOS method from *scipy.signal.sosfiltfilt*, recommended for general purpose filtering. One can also specify “butterworth_ba” for a more traditional and legacy method (often implemented in other software).
- **order** (*int*) – Only used if method is 'butterworth' or 'savgol'. Order of the filter (default is 2).
- **window_size** (*int*) – Only used if method is 'savgol'. The length of the filter window (i.e. the number of coefficients). Must be an odd integer. If 'default', will be set to the sampling rate divided by 10 (101 if the sampling rate is 1000 Hz).
- **powerline** (*int*) – Only used if method is 'powerline'. The powerline frequency (normally 50 Hz or 60 Hz).

See also:

`signal_detrend()`, `signal_psd()`

Returns *array* – Vector containing the filtered signal.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, frequency=0.5) # Low freq
>>> signal += nk.signal_simulate(duration=10, frequency=5) # High freq
>>>
>>> # Lowpass
>>> fig1 = pd.DataFrame({"Raw": signal,
...                      "Butter_2": nk.signal_filter(signal, highcut=3, method=
↳ 'butterworth', order=2),
...                      "Butter_2_BA": nk.signal_filter(signal, highcut=3,
↳ method='butterworth_ba', order=2),
...                      "Butter_5": nk.signal_filter(signal, highcut=3, method=
↳ 'butterworth', order=5),
...                      "Butter_5_BA": nk.signal_filter(signal, highcut=3,
↳ method='butterworth_ba', order=5),
...                      "Bessel_2": nk.signal_filter(signal, highcut=3, method=
↳ 'bessel', order=2),
...                      "Bessel_5": nk.signal_filter(signal, highcut=3, method=
↳ 'bessel', order=5),
...                      "FIR": nk.signal_filter(signal, highcut=3, method='fir')})
↳ .plot(subplots=True)
>>> fig1
```

```
>>> # Highpass
>>> fig2 = pd.DataFrame({"Raw": signal,
...                      "Butter_2": nk.signal_filter(signal, lowcut=2, method=
↳ 'butterworth', order=2),
...                      "Butter_2_ba": nk.signal_filter(signal, lowcut=2, method=
↳ 'butterworth_ba', order=2),
...                      "Butter_5": nk.signal_filter(signal, lowcut=2, method=
↳ 'butterworth', order=5),
...                      "Butter_5_BA": nk.signal_filter(signal, lowcut=2, method=
↳ 'butterworth_ba', order=5),
...                      "Bessel_2": nk.signal_filter(signal, lowcut=2, method=
↳ 'bessel', order=2),
...                      "Bessel_5": nk.signal_filter(signal, lowcut=2, method=
↳ 'bessel', order=5),
...                      "FIR": nk.signal_filter(signal, lowcut=2, method='fir')})
↳ .plot(subplots=True)
>>> fig2
```

```
>>> # Bandpass in real-life scenarios
>>> original = nk.rsp_simulate(duration=30, method="breathmetrics", noise=0)
>>> signal = nk.signal_distort(original, noise_frequency=[0.1, 2, 10, 100], noise_
↳ amplitude=1,
...                      powerline_amplitude=1)
>>>
>>> # Bandpass between 10 and 30 breaths per minute (respiratory rate range)
```

(continues on next page)

(continued from previous page)

```

>>> fig3 = pd.DataFrame({"Raw": signal,
...                      "Butter_2": nk.signal_filter(signal, lowcut=10/60,
↳highcut=30/60,
...                      method='butterworth',
↳order=2),
...                      "Butter_2_BA": nk.signal_filter(signal, lowcut=10/60,
↳highcut=30/60,
...                      method='butterworth_ba',
↳order=2),
...                      "Butter_5": nk.signal_filter(signal, lowcut=10/60,
↳highcut=30/60,
...                      method='butterworth',
↳order=5),
...                      "Butter_5_BA": nk.signal_filter(signal, lowcut=10/60,
↳highcut=30/60,
...                      method='butterworth_ba',
↳order=5),
...                      "Bessel_2": nk.signal_filter(signal, lowcut=10/60,
↳highcut=30/60,
...                      method='bessel', order=2),
...                      "Bessel_5": nk.signal_filter(signal, lowcut=10/60,
↳highcut=30/60,
...                      method='bessel', order=5),
...                      "FIR": nk.signal_filter(signal, lowcut=10/60, highcut=30/
↳60,
...                      method='fir'),
...                      "Savgol": nk.signal_filter(signal, method='savgol')}).
↳plot(subplots=True)
>>> fig3

```

signal_findpeaks (*signal*, *height_min=None*, *height_max=None*, *relative_height_min=None*, *relative_height_max=None*, *relative_mean=True*, *relative_median=False*, *relative_max=False*)

Find peaks in a signal.

Locate peaks (local maxima) in a signal and their related characteristics, such as height (prominence), width and distance with other peaks.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **height_min** (*float*) – The minimum height (i.e., amplitude in terms of absolute values). For example, ``height_min=20`` will remove all peaks which height is smaller or equal to 20 (in the provided signal's values).
- **height_max** (*float*) – The maximum height (i.e., amplitude in terms of absolute values).
- **relative_height_min** (*float*) – The minimum height (i.e., amplitude) relative to the sample (see below). For example, *relative_height_min=-2.96* will remove all peaks which height lies below 2.96 standard deviations from the mean of the heights.
- **relative_height_max** (*float*) – The maximum height (i.e., amplitude) relative to the sample (see below).
- **relative_mean** (*bool*) – If a relative threshold is specified, how should it be computed (i.e., relative to what?). *relative_mean=True* will use Z-scores.

- **relative_median** (*bool*) – If a relative threshold is specified, how should it be computed (i.e., relative to what?). Relative to median uses a more robust form of standardization (see `standardize()`).
- **relative_max** (*bool*) – If a relative threshold is specified, how should it be computed (i.e., relative to what?). Relative to max will consider the maximum height as the reference.

Returns *dict* – Returns a dict itself containing 5 arrays: - ‘Peaks’ contains the peaks indices (as relative to the given signal). For instance, the value 3 means that the third data point of the signal is a peak. - ‘Distance’ contains, for each peak, the closest distance with another peak. Note that these values will be recomputed after filtering to match the selected peaks. - ‘Height’ contains the prominence of each peak. See `scipy.signal.peak_prominences()`. - ‘Width’ contains the width of each peak. See `scipy.signal.peak_widths()`. - ‘Onset’ contains the onset, start (or left trough), of each peak. - ‘Offset’ contains the offset, end (or right trough), of each peak.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>> import scipy.misc
>>>
>>> signal = nk.signal_simulate(duration=5)
>>> info = nk.signal_findpeaks(signal)
>>> fig1 = nk.events_plot([info["Onsets"], info["Peaks"]], signal)
>>> fig1
>>>
>>> signal = nk.signal_distort(signal)
>>> info = nk.signal_findpeaks(signal, height_min=1)
>>> fig2 = nk.events_plot(info["Peaks"], signal)
>>> fig2
>>>
>>> # Filter peaks
>>> ecg = scipy.misc.electrocardiogram()
>>> signal = ecg[0:1000]
>>> info1 = nk.signal_findpeaks(signal, relative_height_min=0)
>>> info2 = nk.signal_findpeaks(signal, relative_height_min=1)
>>> fig3 = nk.events_plot([info1["Peaks"], info2["Peaks"]], signal)
>>> fig3
```

See also:

`scipy.signal.find_peaks()`, `scipy.signal.peak_widths()`, `peak_prominences`, `signal.peak_widths()`, `eda_findpeaks()`, `ecg_findpeaks()`, `rsp_findpeaks()`, `signal_fixpeaks()`

signal_fixpeaks (*peaks*, *sampling_rate=1000*, *iterative=True*, *show=False*, *interval_min=None*, *interval_max=None*, *relative_interval_min=None*, *relative_interval_max=None*, *robust=False*, *method='Kubios'*)

Correct erroneous peak placements.

Identify and correct erroneous peak placements based on outliers in peak-to-peak differences (period).

Parameters

- **peaks** (*list or array or DataFrame or Series or dict*) – The samples at which the peaks occur. If an array is passed in, it is assumed that it was obtained with `signal_findpeaks()`.

If a DataFrame is passed in, it is assumed to be obtained with `ecg_findpeaks()` or `ppg_findpeaks()` and to be of the same length as the input signal.

- **sampling_rate** (*int*) – The sampling frequency of the signal that contains the peaks (in Hz, i.e., samples/second).
- **iterative** (*bool*) – Whether or not to apply the artifact correction repeatedly (results in superior artifact correction).
- **show** (*bool*) – Whether or not to visualize artifacts and artifact thresholds.
- **interval_min** (*float*) – The minimum interval between the peaks.
- **interval_max** (*float*) – The maximum interval between the peaks.
- **relative_interval_min** (*float*) – The minimum interval between the peaks as relative to the sample (expressed in standard deviation from the mean).
- **relative_interval_max** (*float*) – The maximum interval between the peaks as relative to the sample (expressed in standard deviation from the mean).
- **robust** (*bool*) – Use a robust method of standardization (see `standardize()`) for the relative thresholds.
- **method** (*str*) – Either “Kubios” or “Neurokit”. “Kubios” uses the artifact detection and correction described in Lipponen, J. A., & Tarvainen, M. P. (2019). Note that “Kubios” is only meant for peaks in ECG or PPG. “neurokit” can be used with peaks in ECG, PPG, or respiratory data.

Returns

- **peaks_clean** (*array*) – The corrected peak locations.
- **artifacts** (*dict*) – Only if method=”Kubios”. A dictionary containing the indices of artifacts, accessible with the keys “ectopic”, “missed”, “extra”, and “longshort”.

See also:

`signal_findpeaks()`, `ecg_findpeaks()`, `ecg_peaks()`, `ppg_findpeaks()`, `ppg_peaks()`

Examples

```
>>> import neurokit2 as nk
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> # Kubios
>>> ecg = nk.ecg_simulate(duration=240, noise=0.25, heart_rate=70, random_
↳state=42)
>>> rpeaks_uncorrected = nk.ecg_findpeaks(ecg)
>>> artifacts, rpeaks_corrected = nk.signal_fixpeaks(rpeaks_uncorrected,
↳iterative=True,
...                                                show=True, method="Kubios")
>>> rate_corrected = nk.signal_rate(rpeaks_corrected, desired_length=len(ecg))
>>> rate_uncorrected = nk.signal_rate(rpeaks_uncorrected, desired_length=len(ecg))
>>>
>>> fig, ax = plt.subplots()
>>> ax.plot(rate_uncorrected, label="heart rate without artifact correction")
>>> ax.plot(rate_corrected, label="heart rate with artifact correction")
>>> ax.legend(loc="upper right")
>>>
```

(continues on next page)

(continued from previous page)

```

>>> # NeuroKit
>>> signal = nk.signal_simulate(duration=4, sampling_rate=1000, frequency=1)
>>> peaks_true = nk.signal_findpeaks(signal) ["Peaks"]
>>> peaks = np.delete(peaks_true, [1]) # create gaps
>>>
>>> signal = nk.signal_simulate(duration=20, sampling_rate=1000, frequency=1)
>>> peaks_true = nk.signal_findpeaks(signal) ["Peaks"]
>>> peaks = np.delete(peaks_true, [5, 15]) # create gaps
>>> peaks = np.sort(np.append(peaks, [1350, 11350, 18350])) # add artifacts
>>>
>>> peaks_corrected = nk.signal_fixpeaks(peaks=peaks, interval_min=0.5, interval_
->max=1.5, method="neurokit")
>>> # Plot and shift original peaks to the right to see the difference.
>>> fig = nk.events_plot([peaks + 50, peaks_corrected], signal)
>>> fig

```

References

- Lipponen, J. A., & Tarvainen, M. P. (2019). A robust algorithm for heart rate variability time

series artefact correction using novel beat classification. Journal of medical engineering & technology, 43(3), 173-181. 10.1080/03091902.2019.1640306

signal_formatpeaks (*info, desired_length, peak_indices=None*)

Transforms an peak-info dict to a signal of given length.

signal_interpolate (*x_values, y_values, x_new=None, method='quadratic'*)

Interpolate a signal.

Interpolate a signal using different methods.

Parameters

- **x_values** (*Union[list, np.array, pd.Series]*) – The samples corresponding to the values to be interpolated.
- **y_values** (*Union[list, np.array, pd.Series]*) – The values to be interpolated.
- **x_new** (*Union[list, np.array, pd.Series] or int*) – The samples at which to interpolate the y_values. Samples before the first value in x_values or after the last value in x_values will be extrapolated. If an integer is passed, x_new will be considered as the desired length of the interpolated signal between the first and the last values of x_values. No extrapolation will be done for values before or after the first and the last value of x_values.
- **method** (*str*) – Method of interpolation. Can be 'linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', 'next' or 'monotone_cubic'. 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order; 'previous' and 'next' simply return the previous or next value of the point) or as an integer specifying the order of the spline interpolator to use. See <https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html> for details on the 'monotone_cubic' method.

Returns *array* – Vector of interpolated samples.

Examples

```
>>> import numpy as np
>>> import neurokit2 as nk
>>> import matplotlib.pyplot as plt
>>>
>>> # Generate points
>>> signal = nk.signal_simulate(duration=1, sampling_rate=10)
>>>
>>> # List all interpolation methods and interpolation parameters
>>> interpolation_methods = ["zero", "linear", "quadratic", "cubic", 5, "nearest",
→ "monotone_cubic"]
>>> x_values = np.linspace(0, 1, num=10)
>>> x_new = np.linspace(0, 1, num=1000)
>>>
>>> # Visualize all interpolations
>>> fig, ax = plt.subplots()
>>> ax.scatter(x_values, signal, label="original datapoints", zorder=3)
>>> for im in interpolation_methods:
...     signal_interpolated = nk.signal_interpolate(x_values, signal, x_new=x_new,
→ method=im)
...     ax.plot(x_new, signal_interpolated, label=im)
>>> ax.legend(loc="upper left")
```

signal_merge (*signal1*, *signal2*, *time1*=[0, 10], *time2*=[0, 10])

Arbitrary addition of two signals with different time ranges.

Parameters

- **signal1** (*Union[list, np.array, pd.Series]*) – The first signal (i.e., a time series)s in the form of a vector of values.
- **signal2** (*Union[list, np.array, pd.Series]*) – The second signal (i.e., a time series)s in the form of a vector of values.
- **time1** (*list*) – Lists containing two numeric values corresponding to the beginning and end of *signal1*.
- **time2** (*list*) – Same as above, but for *signal2*.

Returns *array* – Vector containing the sum of the two signals.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal1 = np.cos(np.linspace(start=0, stop=10, num=100))
>>> signal2 = np.cos(np.linspace(start=0, stop=20, num=100))
>>>
>>> signal = nk.signal_merge(signal1, signal2, time1=[0, 10], time2=[-5, 5])
>>> nk.signal_plot(signal)
```

signal_period (*peaks*, *sampling_rate*=1000, *desired_length*=None, *interpolation_method*='monotone_cubic')

Calculate signal period from a series of peaks.

Parameters

- **peaks** (*Union[list, np.array, pd.DataFrame, pd.Series, dict]*) – The samples at which the peaks occur. If an array is passed in, it is assumed that it was obtained with `signal_findpeaks()`. If a DataFrame is passed in, it is assumed it is of the same length as the input signal in which occurrences of R-peaks are marked as “1”, with such containers obtained with e.g., `ecg_findpeaks()` or `rsp_findpeaks()`.
- **sampling_rate** (*int*) – The sampling frequency of the signal that contains peaks (in Hz, i.e., samples/second). Defaults to 1000.
- **desired_length** (*int*) – If left at the default None, the returned period will have the same number of elements as peaks. If set to a value larger than the sample at which the last peak occurs in the signal (i.e., `peaks[-1]`), the returned period will be interpolated between peaks over *desired_length* samples. To interpolate the period over the entire duration of the signal, set *desired_length* to the number of samples in the signal. Cannot be smaller than or equal to the sample at which the last peak occurs in the signal. Defaults to None.
- **interpolation_method** (*str*) – Method used to interpolate the rate between peaks. See `signal_interpolate()`. ‘monotone_cubic’ is chosen as the default interpolation method since it ensures monotone interpolation between data points (i.e., it prevents physiologically implausible “overshoots” or “undershoots” in the y-direction). In contrast, the widely used cubic spline interpolation does not ensure monotonicity.

Returns *array* – A vector containing the period.

See also:

`signal_findpeaks()`, `signal_fixpeaks()`, `signal_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, sampling_rate=1000, frequency=1)
>>> info = nk.signal_findpeaks(signal)
>>>
>>> period = nk.signal_period(peaks=info["Peaks"], desired_length=len(signal))
>>> nk.signal_plot(period)
```

signal_phase (*signal, method='radians'*)

Compute the phase of the signal.

The real phase has the property to rotate uniformly, leading to a uniform distribution density. The prophase typically doesn’t fulfill this property. The following functions applies a nonlinear transformation to the phase signal that makes its distribution exactly uniform. If a binary vector is provided (containing 2 unique values), the function will compute the phase of completion of each phase as denoted by each value.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **method** (*str*) – The values in which the phase is expressed. Can be ‘radians’ (default), ‘degrees’ (for values between 0 and 360) or ‘percents’ (for values between 0 and 1).

See also:

`signal_filter()`, `signal_zerocrossings()`, `signal_findpeaks()`

Returns *array* – A vector containing the phase of the signal, between 0 and 2π .

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10)
>>> phase = nk.signal_phase(signal)
>>> nk.signal_plot([signal, phase])
>>>
>>> rsp = nk.rsp_simulate(duration=30)
>>> phase = nk.signal_phase(rsp, method="degrees")
>>> nk.signal_plot([rsp, phase])
>>>
>>> # Percentage of completion of two phases
>>> signal = nk.signal_binarize(nk.signal_simulate(duration=10))
>>> phase = nk.signal_phase(signal, method="percents")
>>> nk.signal_plot([signal, phase])
```

signal_plot (*signal*, *sampling_rate*=None, *subplots*=False, *standardize*=False, *labels*=None, ***kwargs*)
Plot signal with events as vertical lines.

Parameters

- **signal** (*array or DataFrame*) – Signal array (can be a dataframe with many signals).
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second). Needs to be supplied if the data should be plotted over time in seconds. Otherwise the data is plotted over samples. Defaults to None.
- **subplots** (*bool*) – If True, each signal is plotted in a subplot.
- **standardize** (*bool*) – If True, all signals will have the same scale (useful for visualisation).
- **labels** (*str or list*) – Defaults to None.
- ****kwargs** (*optional*) – Arguments passed to matplotlib plotting.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, sampling_rate=1000)
>>> nk.signal_plot(signal, labels='signal1', sampling_rate=1000, color="red")
>>>
>>> data = pd.DataFrame({"Signal2": np.cos(np.linspace(start=0, stop=20,
    ↳ num=1000)),
    ↳ "Signal3": np.sin(np.linspace(start=0, stop=20,
    ↳ num=1000)),
    ↳ "Signal4": nk.signal_binarize(np.cos(np.linspace(start=0,
    ↳ stop=40, num=1000)))})
>>> nk.signal_plot(data, labels=['signal_1', 'signal_2', 'signal_3'],
    ↳ subplots=False)
>>> nk.signal_plot([signal, data], standardize=True)
```

signal_power (*signal*, *frequency_band*, *sampling_rate*=1000, *continuous*=False, *show*=False, ***kwargs*)
Compute the power of a signal in a given frequency band.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **frequency_band** (*tuple or list*) – Tuple or list of tuples indicating the range of frequencies to compute the power in.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).
- **continuous** (*bool*) – Compute instant frequency, or continuous power.
- **show** (*bool*) – If True, will return a Poincaré plot. Defaults to False.
- ****kwargs** – Keyword arguments to be passed to `signal_psd()`.

See also:

`signal_filter()`, `signal_psd()`

Returns `pd.DataFrame` – A DataFrame containing the Power Spectrum values and a plot if `show` is True.

Examples

```
>>> import neurokit2 as nk
>>> import numpy as np
>>>
>>> # Instant power
>>> signal = nk.signal_simulate(frequency=5) + 0.5*nk.signal_
↳simulate(frequency=20)
>>> power_plot = nk.signal_power(signal, frequency_band=[(18, 22), (10, 14)],
↳method="welch", show=True)
>>> power_plot
>>>
>>> # Continuous (simulated signal)
>>> signal = np.concatenate((nk.ecg_simulate(duration=30, heart_rate=75), nk.ecg_
↳simulate(duration=30, heart_rate=85)))
>>> power = nk.signal_power(signal, frequency_band=[(72/60, 78/60), (82/60, 88/
↳60)], continuous=True)
>>> processed, _ = nk.ecg_process(signal)
>>> power["ECG_Rate"] = processed["ECG_Rate"]
>>> nk.signal_plot(power, standardize=True)
>>>
>>> # Continuous (real signal)
>>> signal = nk.data("bio_eventrelated_100hz")["ECG"]
>>> power = nk.signal_power(signal, sampling_rate=100, frequency_band=[(0.12, 0.
↳15), (0.15, 0.4)], continuous=True)
>>> processed, _ = nk.ecg_process(signal, sampling_rate=100)
>>> power["ECG_Rate"] = processed["ECG_Rate"]
>>> nk.signal_plot(power, standardize=True)
```

signal_psd(*signal*, *sampling_rate=1000*, *method='welch'*, *show=True*, *min_frequency=0*, *max_frequency=inf*, *window=None*, *ar_order=15*, *order_criteria='KIC'*, *order_corrected=True*, *burg_norm=True*)
 Compute the Power Spectral Density (PSD).

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).
- **method** (*str*) – Either ‘multitapers’ (default; requires the ‘mne’ package), or ‘welch’ (requires the ‘scipy’ package).
- **show** (*bool*) – If True, will return a plot. If False, will return the density values that can be plotted externally.
- **min_frequency** (*float*) – The minimum frequency.
- **max_frequency** (*float*) – The maximum frequency.
- **window** (*int*) – Length of each window in seconds (for Welch method). If None (default), window will be automatically calculated to capture at least 2 cycles of min_frequency. If the length of recording does not allow the formal, window will be default to half of the length of recording.
- **ar_order** (*int*) – The order of autoregression (for AR methods e.g. Burg).
- **order_criteria** (*str*) – The criteria to automatically select order in parametric PSD (for AR methods e.g. Burg).
- **order_corrected** (*bool*) – Specify for AIC and KIC order_criteria. If unsure which method to use to choose the order, rely on the default of corrected KIC.
- **bug_norm** (*bool*) – Normalization for Burg method.

See also:

`signal_filter()`, `mne.time_frequency.psd_array_multitaper()`, `scipy.signal.welch()`

Returns *pd.DataFrame* – A DataFrame containing the Power Spectrum values and a plot if *show* is True.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(frequency=5) + 0.5*nk.signal_
↳simulate(frequency=20)
>>>
>>> fig1 = nk.signal_psd(signal, method="multitapers")
>>> fig1
>>> fig2 = nk.signal_psd(signal, method="welch", min_frequency=1)
>>> fig2
>>> fig3 = nk.signal_psd(signal, method="burg", min_frequency=1)
>>>
>>> data = nk.signal_psd(signal, method="multitapers", max_frequency=30,
↳show=False)
>>> fig4 = data.plot(x="Frequency", y="Power")
>>> fig4
>>> data = nk.signal_psd(signal, method="welch", max_frequency=30, show=False,
↳min_frequency=1)
>>> fig5 = data.plot(x="Frequency", y="Power")
>>> fig5
```

signal_rate (*peaks*, *sampling_rate=1000*, *desired_length=None*, *interpolation_method='monotone_cubic'*)
Calculate signal rate from a series of peaks.

This function can also be called either via `ecg_rate()`, `ppg_rate()` or `rsp_rate()` (aliases provided for consistency).

Parameters

- **peaks** (*Union[list, np.array, pd.DataFrame, pd.Series, dict]*) – The samples at which the peaks occur. If an array is passed in, it is assumed that it was obtained with `signal_findpeaks()`. If a DataFrame is passed in, it is assumed it is of the same length as the input signal in which occurrences of R-peaks are marked as “1”, with such containers obtained with e.g., `ecg_findpeaks()` or `rsp_findpeaks()`.
- **sampling_rate** (*int*) – The sampling frequency of the signal that contains peaks (in Hz, i.e., samples/second). Defaults to 1000.
- **desired_length** (*int*) – If left at the default None, the returned rate will have the same number of elements as peaks. If set to a value larger than the sample at which the last peak occurs in the signal (i.e., `peaks[-1]`), the returned rate will be interpolated between peaks over *desired_length* samples. To interpolate the rate over the entire duration of the signal, set *desired_length* to the number of samples in the signal. Cannot be smaller than or equal to the sample at which the last peak occurs in the signal. Defaults to None.
- **interpolation_method** (*str*) – Method used to interpolate the rate between peaks. See `signal_interpolate()`. ‘monotone_cubic’ is chosen as the default interpolation method since it ensures monotone interpolation between data points (i.e., it prevents physiologically implausible “overshoots” or “undershoots” in the y-direction). In contrast, the widely used cubic spline interpolation does not ensure monotonicity.

Returns *array* – A vector containing the rate.

See also:

`signal_findpeaks()`, `signal_fixpeaks()`, `signal_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=10, sampling_rate=1000, frequency=1)
>>> info = nk.signal_findpeaks(signal)
>>>
>>> rate = nk.signal_rate(peaks=info["Peaks"], desired_length=len(signal))
>>> fig = nk.signal_plot(rate)
>>> fig
```

signal_recompose (*components*, *method='wcorr'*, *threshold=0.5*, *keep_sd=None*, ***kwargs*)
Combine signal sources after decomposition.

Combine and reconstruct meaningful signal sources after signal decomposition.

Parameters

- **components** (*array*) – Array of components obtained via `signal_decompose()`.
- **method** (*str*) – The decomposition method. Can be one of ‘wcorr’.
- **threshold** (*float*) – The threshold used to group components together.

- **keep_sd** (*float*) – If a float is specified, will only keep the reconstructed components that are superior or equal to that percentage of the max standard deviation (SD) of the components. For instance, `keep_sd=0.01` will remove all components with SD is lower than 1% of the max SD. This can be used to filter out noise.
- ****kwargs** – Other arguments to override for instance `metric='chebyshev'`.

Returns *Array* – Components of the recomposed components.

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Create complex signal
>>> signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01) # High freq
>>> signal += 3 * nk.signal_simulate(duration=10, frequency=3, noise=0.01) # ↵
    ↪Higher freq
>>> signal += 3 * np.linspace(0, 2, len(signal)) # Add baseline and trend
>>> signal += 2 * nk.signal_simulate(duration=10, frequency=0.1, noise=0)
>>>
>>> # Decompose signal
>>> components = nk.signal_decompose(signal, method='emd')
>>>
>>> # Recompose
>>> recomposed = nk.signal_recompose(components, method='wcorr', threshold=0.90)
>>> fig = nk.signal_plot(components) # Visualize components
>>> fig
```

signal_resample (*signal*, *desired_length=None*, *sampling_rate=None*, *desired_sampling_rate=None*, *method='interpolation'*)

Resample a continuous signal to a different length or sampling rate.

Up- or down-sample a signal. The user can specify either a desired length for the vector, or input the original sampling rate and the desired sampling rate. See <https://github.com/neuropsychology/NeuroKit/scripts/resampling.ipynb> for a comparison of the methods.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **desired_length** (*int*) – The desired length of the signal.
- **sampling_rate** (*int*) – The original sampling frequency (in Hz, i.e., samples/second).
- **desired_sampling_rate** (*int*) – The desired (output) sampling frequency (in Hz, i.e., samples/second).
- **method** (*str*) – Can be ‘interpolation’ (see `scipy.ndimage.zoom()`), ‘numpy’ for numpy’s interpolation (see `numpy.interp()`), ‘pandas’ for Pandas’ time series resampling, ‘poly’ (see `scipy.signal.resample_poly()`) or ‘FFT’ (see `scipy.signal.resample()`) for the Fourier method. FFT is the most accurate (if the signal is periodic), but becomes exponentially slower as the signal length increases. In contrast, ‘interpolation’ is the fastest, followed by ‘numpy’, ‘poly’ and ‘pandas’.

Returns *array* – Vector containing resampled signal values.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = np.cos(np.linspace(start=0, stop=20, num=100))
>>>
>>> # Downsample
>>> downsampled_interpolation = nk.signal_resample(signal, method="interpolation",
...                                                sampling_rate=1000, desired_
↳sampling_rate=500)
>>> downsampled_fft = nk.signal_resample(signal, method="FFT",
...                                       sampling_rate=1000, desired_sampling_
↳rate=500)
>>> downsampled_poly = nk.signal_resample(signal, method="poly",
...                                       sampling_rate=1000, desired_sampling_
↳rate=500)
>>> downsampled_numpy = nk.signal_resample(signal, method="numpy",
...                                       sampling_rate=1000, desired_sampling_
↳rate=500)
>>> downsampled_pandas = nk.signal_resample(signal, method="pandas",
...                                       sampling_rate=1000, desired_sampling_
↳rate=500)
>>>
>>> # Upsample
>>> upsampled_interpolation = nk.signal_resample(downsampled_interpolation,
...                                              method="interpolation",
...                                              sampling_rate=500, desired_
↳sampling_rate=1000)
>>> upsampled_fft = nk.signal_resample(downsampled_fft, method="FFT",
...                                    sampling_rate=500, desired_sampling_
↳rate=1000)
>>> upsampled_poly = nk.signal_resample(downsampled_poly, method="poly",
...                                    sampling_rate=500, desired_sampling_
↳rate=1000)
>>> upsampled_numpy = nk.signal_resample(downsampled_numpy, method="numpy",
...                                    sampling_rate=500, desired_sampling_
↳rate=1000)
>>> upsampled_pandas = nk.signal_resample(downsampled_pandas, method="pandas",
...                                    sampling_rate=500, desired_sampling_
↳rate=1000)
>>>
>>> # Compare with original
>>> fig = pd.DataFrame({"Original": signal,
...                    "Interpolation": upsampled_interpolation,
...                    "FFT": upsampled_fft,
...                    "Poly": upsampled_poly,
...                    "Numpy": upsampled_numpy,
...                    "Pandas": upsampled_pandas}).plot(style='.-')
>>> fig
>>>
>>> # Timing benchmarks
>>> %timeit nk.signal_resample(signal, method="interpolation",
...                           sampling_rate=1000, desired_sampling_rate=500)
>>> %timeit nk.signal_resample(signal, method="FFT",
...                             sampling_rate=1000, desired_sampling_rate=500)
```

(continues on next page)

(continued from previous page)

```
>>> %timeit nk.signal_resample(signal, method="poly",
...                             sampling_rate=1000, desired_sampling_rate=500)
>>> %timeit nk.signal_resample(signal, method="numpy",
...                             sampling_rate=1000, desired_sampling_rate=500)
>>> %timeit nk.signal_resample(signal, method="pandas",
...                             sampling_rate=1000, desired_sampling_rate=500)
```

See also:

`scipy.signal.resample_poly()`, `scipy.signal.resample()`, `scipy.ndimage.zoom()`

signal_simulate (*duration=10, sampling_rate=1000, frequency=1, amplitude=0.5, noise=0, silent=False*)
Simulate a continuous signal.

Parameters

- **duration** (*float*) – Desired length of duration (s).
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second).
- **frequency** (*float or list*) – Oscillatory frequency of the signal (in Hz, i.e., oscillations per second).
- **amplitude** (*float or list*) – Amplitude of the oscillations.
- **noise** (*float*) – Noise level (amplitude of the laplace noise).
- **silent** (*bool*) – If False (default), might print warnings if impossible frequencies are queried.

Returns *array* – The simulated signal.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> fig = pd.DataFrame({"1Hz": nk.signal_simulate(duration=5, frequency=1),
...                    "2Hz": nk.signal_simulate(duration=5, frequency=2),
...                    "Multi": nk.signal_simulate(duration=5, frequency=[0.5, 1.5, 2.5],
...                    amplitude=[0.5, 0.2])}).plot()
>>> fig
```

signal_smooth (*signal, method='convolution', kernel='boxzen', size=10, alpha=0.1*)
Signal smoothing.

Signal smoothing can be achieved using either the convolution of a filter kernel with the input signal to compute the smoothed signal (Smith, 1997) or a LOESS regression.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **method** (*str*) – Can be one of ‘convolution’ (default) or ‘loess’.
- **kernel** (*Union[str, np.array]*) – Only used if *method* is ‘convolution’. Type of kernel to use; if array, use directly as the kernel. Can be one of ‘median’, ‘boxzen’, ‘boxcar’, ‘triang’, ‘blackman’, ‘hamming’, ‘hann’, ‘bartlett’, ‘flattop’, ‘parzen’, ‘bohman’,

‘blackmanharris’, ‘nuttall’, ‘barthann’, ‘kaiser’ (needs beta), ‘gaussian’ (needs std), ‘general_gaussian’ (needs power, width), ‘slepian’ (needs width) or ‘chebwin’ (needs attenuation).

- **size** (*int*) – Only used if *method* is ‘convolution’. Size of the kernel; ignored if kernel is an array.
- **alpha** (*float*) – Only used if *method* is ‘loess’. The parameter which controls the degree of smoothing.

Returns *array* – Smoothed signal.

See also:

`fit_loess()`

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = np.cos(np.linspace(start=0, stop=10, num=1000))
>>> distorted = nk.signal_distort(signal, noise_amplitude=[0.3, 0.2, 0.1, 0.05],
↳ noise_frequency=[5, 10, 50, 100])
>>>
>>> size = len(signal)/100
>>> signals = pd.DataFrame({"Raw": distorted,
...                          "Median": nk.signal_smooth(distorted, kernel='median',
↳ size=size-1),
...                          "BoxZen": nk.signal_smooth(distorted, kernel='boxzen',
↳ size=size),
...                          "Triang": nk.signal_smooth(distorted, kernel='triang',
↳ size=size),
...                          "Blackman": nk.signal_smooth(distorted, kernel=
↳ 'blackman', size=size),
...                          "Loess_01": nk.signal_smooth(distorted, method='loess
↳ ', alpha=0.1),
...                          "Loess_02": nk.signal_smooth(distorted, method='loess
↳ ', alpha=0.2),
...                          "Loess_05": nk.signal_smooth(distorted, method='loess
↳ ', alpha=0.5)})
>>> fig = signals.plot()
>>> fig_magnify = signals[50:150].plot() # Magnify
>>> fig_magnify
```

References

- Smith, S. W. (1997). The scientist and engineer’s guide to digital signal processing.

signal_synchrony (*signal1*, *signal2*, *method*='hilbert', *window_size*=50)

Compute the synchrony (coupling) between two signals.

Compute a continuous index of coupling between two signals either using the ‘Hilbert’ method to get the instantaneous phase synchrony, or using rolling window correlation.

The instantaneous phase synchrony measures the phase similarities between signals at each timepoint. The phase refers to the angle of the signal, calculated through the hilbert transform, when it is resonating between $-\pi$ to π degrees. When two signals line up in phase their angular difference becomes zero.

For less clean signals, windowed correlations are widely used because of their simplicity, and can be a good a robust approximation of synchrony between two signals. The limitation is the need to select a window.

Parameters

- **signal1** (*Union[list, np.array, pd.Series]*) – Time series in the form of a vector of values.
- **signal2** (*Union[list, np.array, pd.Series]*) – Time series in the form of a vector of values.
- **method** (*str*) – The method to use. Can be one of ‘hilbert’ or ‘correlation’.
- **window_size** (*int*) – Only used if *method*=‘correlation’. The number of samples to use for rolling correlation.

See also:

`signal_filter()`, `signal_zerocrossings()`, `signal_findpeaks()`

Returns *array* – A vector containing the phase of the signal, between 0 and 2π .

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal1 = nk.signal_simulate(duration=10, frequency=1)
>>> signal2 = nk.signal_simulate(duration=10, frequency=1.5)
>>>
>>> coupling_h = nk.signal_synchrony(signal1, signal2, method="hilbert")
>>> coupling_c = nk.signal_synchrony(signal1, signal2, method="correlation",
->window_size=1000/2)
>>>
>>> fig = nk.signal_plot([signal1, signal2, coupling_h, coupling_c])
>>> fig
```

References

- http://jinhyuncheong.com/jekyll/update/2017/12/10/Timeseries_synchrony_tutorial_and_simulations.html

signal_zerocrossings (*signal, direction='both'*)

Locate the indices where the signal crosses zero.

Note that when the signal crosses zero between two points, the first index is returned.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **direction** (*str*) – Direction in which the signal crosses zero, can be “positive”, “negative” or “both” (default).

Returns *array* – Vector containing the indices of zero crossings.

Examples

```
>>> import numpy as np
>>> import neurokit2 as nk
>>>
>>> signal = np.cos(np.linspace(start=0, stop=15, num=1000))
>>> zeros = nk.signal_zerocrossings(signal)
>>> fig = nk.events_plot(zeros, signal)
>>> fig
>>>
>>> # Only upward or downward zerocrossings
>>> up = nk.signal_zerocrossings(signal, direction='up')
>>> down = nk.signal_zerocrossings(signal, direction='down')
>>> fig = nk.events_plot([up, down], signal)
>>> fig
```

7.9 Events

Submodule for NeuroKit.

events_find (*event_channel*, *threshold*='auto', *threshold_keep*='above', *start_at*=0, *end_at*=None, *duration_min*=1, *duration_max*=None, *inter_min*=0, *discard_first*=0, *discard_last*=0, *event_labels*=None, *event_conditions*=None)

Find and select events in a continuous signal (e.g., from a photosensor).

Parameters

- **event_channel** (*array or list*) – The channel containing the events.
- **threshold** (*str or float*) – The threshold value by which to select the events. If “auto”, takes the value between the max and the min.
- **threshold_keep** (*str*) – “above” or “below”, define the events as above or under the threshold. For photosensors, a white screen corresponds usually to higher values. Therefore, if your events are signaled by a black colour, events values are the lower ones, and you should set the cut to “below”.
- **start_at** (*int*) – Keep events which onset is after a particular time point.
- **end_at** (*int*) – Keep events which onset is before a particular time point.
- **duration_min** (*int*) – The minimum duration of an event to be considered as such (in time points).
- **duration_max** (*int*) – The maximum duration of an event to be considered as such (in time points).
- **inter_min** (*int*) – The minimum duration after an event for the subsequent event to be considered as such (in time points). Useful when spurious consecutive events are created due to very high sampling rate.
- **discard_first** (*int*) – Discard first or last n events. Useful if the experiment starts with some spurious events. If *discard_first*=0, no first event is removed.
- **discard_last** (*int*) – Discard first or last n events. Useful if the experiment ends with some spurious events. If *discard_last*=0, no last event is removed.
- **event_labels** (*list*) – A list containing unique event identifiers. If *None*, will use the event index number.

- **event_conditions** (*list*) – An optional list containing, for each event, for example the trial category, group or experimental conditions.

Returns *dict* – Dict containing 3 or 4 arrays, ‘onset’ for event onsets, ‘duration’ for event durations, ‘label’ for the event identifiers and the optional ‘conditions’ passed to *event_conditions*.

See also:

`events_plot()`, `events_to_mne()`

Example

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=4)
>>> events = nk.events_find(signal)
>>> events
{'onset': array(...),
 'duration': array(...),
 'label': array(...)}
>>>
>>> nk.events_plot(events, signal)
<Figure ...>
```

events_plot (*events*, *signal=None*, *show=True*, *color='red'*, *linestyle='--'*)

Plot events in signal.

Parameters

- **events** (*list or ndarray or dict*) – Events onset location. Can also be a list of lists, in which case it will mark them with different colors. If a dict is passed (e.g., from ‘events_find()’), will select only the ‘onset’ list.
- **signal** (*array or DataFrame*) – Signal array (can be a dataframe with many signals).
- **show** (*bool*) – If True, will return a plot. If False, will return a DataFrame that can be plotted externally.
- **color** (*str*) – Argument passed to matplotlib plotting.
- **linestyle** (*str*) – Argument passed to matplotlib plotting.

Returns *fig* – Figure representing a plot of the signal and the event markers.

See also:

`events_find()`

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> fig = nk.events_plot([1, 3, 5])
>>> fig
>>>
>>> # With signal
>>> signal = nk.signal_simulate(duration=4)
>>> events = nk.events_find(signal)
>>> fig1 = nk.events_plot(events, signal)
>>> fig1
>>>
>>> # Different events
>>> events1 = events["onset"]
>>> events2 = np.linspace(0, len(signal), 8)
>>> fig2 = nk.events_plot([events1, events2], signal)
>>> fig2
>>>
>>> # Conditions
>>> events = nk.events_find(signal, event_conditions=["A", "B", "A", "B"])
>>> fig3 = nk.events_plot(events, signal)
>>> fig3
>>>
>>> # Different colors for all events
>>> signal = nk.signal_simulate(duration=20)
>>> events = nk.events_find(signal)
>>> events = [[i] for i in events['onset']]
>>> fig4 = nk.events_plot(events, signal)
>>> fig4
```

events_to_mne (*events*, *event_conditions=None*)

Create MNE compatible events for integration with M/EEG.

Parameters

- **events** (*list or ndarray or dict*) – Events onset location. Can also be a dict obtained through ‘events_find()’.
- **event_conditions** (*list*) – An optional list containing, for each event, for example the trial category, group or experimental conditions. Defaults to None.

Returns *tuple* – MNE-formatted events and the event id, that can be added via ‘raw.add_events(events), and a dictionary with event’s names.

See also:

`events_find()`

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=4)
>>> events = nk.events_find(signal)
>>> events, event_id = nk.events_to_mne(events)
>>> events
array([[ 1,    0,    0],
       [1001,    0,    0],
       [2001,    0,    0],
       [3001,    0,    0]])
>>> event_id
{'event': 0}
>>>
>>> # Conditions
>>> events = nk.events_find(signal, event_conditions=["A", "B", "A", "B"])
>>> events, event_id = nk.events_to_mne(events)
>>> event_id
{'B': 0, 'A': 1}
```

7.10 Data

Submodule for NeuroKit.

data (*dataset='bio_eventrelated_100hz'*)

Download example datasets.

Download and load available [example datasets](#). Note that an internet connexion is necessary.

Parameters *dataset* (*str*) – The name of the dataset. The list and description is available [here](#).

Returns *DataFrame* – The data.

Examples

```
>>> import neurokit2 as nk
>>>
>>> data = nk.data("bio_eventrelated_100hz")
```

read_acqknowledge (*filename*, *sampling_rate='max'*, *resample_method='interpolation'*, *impute_missing=True*)

Read and format a BIOPAC's AcqKnowledge file into a pandas' dataframe.

The function outputs both the dataframe and the sampling rate (encoded within the AcqKnowledge) file.

Parameters

- **filename** (*str*) – Filename (with or without the extension) of a BIOPAC's AcqKnowledge file.
- **sampling_rate** (*int*) – Sampling rate (in Hz, i.e., samples/second). Since an AcqKnowledge file can contain signals recorded at different rates, harmonization is necessary in order to convert it to a DataFrame. Thus, if *sampling_rate* is set to 'max' (default), will keep the maximum recorded sampling rate and upsample the channels with lower rate if

necessary (using the `signal_resample()` function). If the sampling rate is set to a given value, will resample the signals to the desired value. Note that the value of the sampling rate is outputted along with the data.

- **resample_method** (*str*) – Method of resampling (see `signal_resample()`).
- **impute_missing** (*bool*) – Sometimes, due to connections issues, the signal has some holes (short periods without signal). If ‘impute_missing’ is True, will automatically fill the signal interruptions using padding.

Returns

- **df** (*DataFrame*) – The AcqKnowledge file converted to a dataframe.
- **sampling_rate** (*int*) – The AcqKnowledge file converted to its sampling rate.

See also:

`signal_resample()`

Example

```
>>> import neurokit2 as nk
>>>
>>> data, sampling_rate = nk.read_acqknowledge('file.acq')
```

7.11 Epochs

Submodule for NeuroKit.

epochs_create (*data*, *events=None*, *sampling_rate=1000*, *epochs_start=0*, *epochs_end=1*, *event_labels=None*, *event_conditions=None*, *baseline_correction=False*)
Epoching a dataframe.

Parameters

- **data** (*DataFrame*) – A DataFrame containing the different signal(s) as different columns. If a vector of values is passed, it will be transformed in a DataFrame with a single ‘Signal’ column.
- **events** (*list or ndarray or dict*) – Events onset location. If a dict is passed (e.g., from `events_find()`), will select only the ‘onset’ list. If an integer is passed, will use this number to create an evenly spaced list of events. If None, will chunk the signal into successive blocks of the set duration.
- **sampling_rate** (*int*) – The sampling frequency of the signal (in Hz, i.e., samples/second).
- **epochs_start** (*int*) – Epochs start relative to events_onsets (in seconds). The start can be negative to start epochs before a given event (to have a baseline for instance).
- **epochs_end** (*int*) – Epochs end relative to events_onsets (in seconds).
- **event_labels** (*list*) – A list containing unique event identifiers. If None, will use the event index number.
- **event_conditions** (*list*) – An optional list containing, for each event, for example the trial category, group or experimental conditions.
- **baseline_correction** (*bool*) – Defaults to False.

Returns *dict* – A dict containing DataFrames for all epochs.

See also:

`events_find()`, `events_plot()`, `epochs_to_df()`, `epochs_plot()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Get data
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Find events
>>> events = nk.events_find(data["Photosensor"],
...                         threshold_keep='below',
...                         event_conditions=["Negative", "Neutral", "Neutral",
...                                         ↪ "Negative"])
>>> fig1 = nk.events_plot(events, data)
>>> fig1
>>>
>>> # Create epochs
>>> epochs = nk.epochs_create(data, events, sampling_rate=100, epochs_end=3)
>>> fig2 = nk.epochs_plot(epochs)
>>> fig2
>>>
>>> # Baseline correction
>>> epochs = nk.epochs_create(data, events, sampling_rate=100, epochs_end=3, ↪
...                           ↪baseline_correction=True)
>>> fig3 = nk.epochs_plot(epochs)
>>> fig3
>>>
>>> # Chunk into n blocks of 1 second
>>> epochs = nk.epochs_create(data, sampling_rate=100, epochs_end=1)
```

epochs_plot (*epochs*, *legend=True*, *show=True*)

Plot epochs.

Parameters

- **epochs** (*dict*) – A dict containing one DataFrame per event/trial. Usually obtained via `epochs_create()`.
- **legend** (*bool*) – Display the legend (the key of each epoch).
- **show** (*bool*) – If True, will return a plot. If False, will return a DataFrame that can be plotted externally.

Returns **epochs** (*dict*) – dict containing all epochs.

See also:

`events_find()`, `events_plot()`, `epochs_create()`, `epochs_to_df()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example with data
>>> data = nk.data("bio_eventrelated_100hz")
>>> events = nk.events_find(data["Photosensor"],
...                          threshold_keep='below',
...                          event_conditions=["Negative", "Neutral", "Neutral",
-> "Negative"])
>>> epochs = nk.epochs_create(data, events, sampling_rate=200, epochs_end=1)
>>> fig1 = nk.epochs_plot(epochs)
>>> fig1
>>>
>>> # Example with ECG Peaks
>>> signal = nk.ecg_simulate(duration=10)
>>> events = nk.ecg_findpeaks(signal)
>>> epochs = nk.epochs_create(signal, events=events["ECG_R_Peaks"], epochs_start=-
-> 0.5, epochs_end=0.5)
>>> fig2 = nk.epochs_plot(epochs)
>>> fig2
```

`epochs_to_array(epochs)`

Convert epochs to an array.

TODO: make it work with uneven epochs (not the same length).

Parameters `epochs` (*dict*) – A dict containing one DataFrame per event/trial. Usually obtained via `epochs_create()`.

Returns *array* – An array containing all signals.

See also:

`events_find()`, `events_plot()`, `epochs_create()`, `epochs_plot()`

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> # Get data
>>> signal = nk.signal_simulate(sampling_rate=100)
>>>
>>> # Create epochs
>>> epochs = nk.epochs_create(signal, events=[400, 430, 460], sampling_rate=100,
-> epochs_end=1)
>>> X = nk.epochs_to_array(epochs)
>>> nk.signal_plot(X.T)
```

`epochs_to_df(epochs)`

Convert epochs to a DataFrame.

Parameters `epochs` (*dict*) – A dict containing one DataFrame per event/trial. Usually obtained via `epochs_create()`.

Returns *DataFrame* – A DataFrame containing all epochs identifiable by the ‘Label’ column, which time axis is stored in the ‘Time’ column.

See also:

`events_find()`, `events_plot()`, `epochs_create()`, `epochs_plot()`

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> # Get data
>>> data = pd.read_csv("https://raw.githubusercontent.com/neuropsychology/
↳ NeuroKit/dev/data/bio_eventrelated_100hz.csv")
>>>
>>> # Find events
>>> events = nk.events_find(data["Photosensor"],
...                          threshold_keep='below',
...                          event_conditions=["Negative", "Neutral", "Neutral",
↳ "Negative"])
>>> fig = nk.events_plot(events, data)
>>> fig
>>>
>>> # Create epochs
>>> epochs = nk.epochs_create(data, events, sampling_rate=200, epochs_end=3)
>>> data = nk.epochs_to_df(epochs)
```

7.12 Statistics

Submodule for NeuroKit.

cor (*x*, *y*, *method*='pearson', *show*=False)

Density estimation.

Computes kernel density estimates.

Parameters

- **x** (*Union*[*list*, *np.array*, *pd.Series*]) – Vectors of values.
- **y** (*Union*[*list*, *np.array*, *pd.Series*]) – Vectors of values.
- **method** (*str*) – Correlation method. Can be one of 'pearson', 'spearman', 'kendall'.
- **show** (*bool*) – Draw a scatterplot with a regression line.

Returns *r* – The correlation coefficient.

Examples

```
>>> import neurokit2 as nk
>>>
>>> x = [1, 2, 3, 4, 5]
>>> y = [3, 1, 5, 6, 6]
>>> corr = nk.cor(x, y, method="pearson", show=True)
>>> corr
```

density (*x*, *desired_length*=100, *bandwidth*=1, *show*=False)

Density estimation.

Computes kernel density estimates.

Parameters

- **x** (*Union[list, np.array, pd.Series]*) – A vector of values.
- **desired_length** (*int*) – The amount of values in the returned density estimation.
- **bandwidth** (*float*) – The bandwidth of the kernel. The smaller the values, the smoother the estimation.
- **show** (*bool*) – Display the density plot.

Returns

- *x*, *y* – The x axis of the density estimation.
- *y* – The y axis of the density estimation.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.ecg_simulate(duration=20)
>>> x, y = nk.density(signal, bandwidth=0.5, show=True)
>>>
>>> # Bandwidth comparison
>>> x, y1 = nk.density(signal, bandwidth=0.5)
>>> x, y2 = nk.density(signal, bandwidth=1)
>>> x, y3 = nk.density(signal, bandwidth=2)
>>> pd.DataFrame({"x": x, "y1": y1, "y2": y2, "y3": y3}).plot(x="x")
```

distance (*X*=None, *method*='mahalanobis')

Distance.

Compute distance using different metrics.

Parameters

- **X** (*array or DataFrame*) – A dataframe of values.
- **method** (*str*) – The method to use. One of 'mahalanobis' or 'mean' for the average distance from the mean.

Returns *array* – Vector containing the distance values.

Examples

```
>>> from sklearn import datasets
>>> import neurokit2 as nk
>>>
>>> X = datasets.load_iris().data
>>> vector = distance(X)
>>> vector
```

fit_error (*y*, *y_predicted*, *n_parameters*=2)

Calculate the fit error for a model.

Also specific and direct access functions can be used, such as `fit_mse()`, `fit_rmse()` and `fit_r2()`.

Parameters

- **y** (*Union[list, np.array, pd.Series]*) – The response variable (the y axis).
- **y_predicted** (*Union[list, np.array, pd.Series]*) – The fitted data generated by a model.
- **n_parameters** (*int*) – Number of model parameters (for the degrees of freedom used in R2).

Returns *dict* – A dictionary containing different indices of fit error.

See also:

`fit_mse()`, `fit_rmse()`, `fit_r2()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> y = np.array([-1.0, -0.5, 0, 0.5, 1])
>>> y_predicted = np.array([0.0, 0, 0, 0, 0])
>>>
>>> # Master function
>>> x = nk.fit_error(y, y_predicted)
>>> x
>>>
>>> # Direct access
>>> nk.fit_mse(y, y_predicted)
0.5
>>>
>>> nk.fit_rmse(y, y_predicted)
0.7071067811865476
>>>
>>> nk.fit_r2(y, y_predicted, adjusted=False)
0.7071067811865475
>>>
>>> nk.fit_r2(y, y_predicted, adjusted=True, n_parameters=2)
0.057190958417936755
```

fit_loess (*y, X=None, alpha=0.75, order=2*)

Local Polynomial Regression (LOESS)

Performs a LOWESS (LOcally WEighted Scatter-plot Smoother) regression.

Parameters

- **y** (*Union[list, np.array, pd.Series]*) – The response variable (the y axis).
- **X** (*Union[list, np.array, pd.Series]*) – Explanatory variable (the x axis). If 'None', will treat y as a continuous signal (useful for smoothing).
- **alpha** (*float*) – The parameter which controls the degree of smoothing, which corresponds to the proportion of the samples to include in local regression.
- **order** (*int*) – Degree of the polynomial to fit. Can be 1 or 2 (default).

Returns *array* – Prediction of the LOESS algorithm.

See also:

`signal_smooth()`, `signal_detrend()`, `fit_error()`

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> signal = np.cos(np.linspace(start=0, stop=10, num=1000))
>>> distorted = nk.signal_distort(signal, noise_amplitude=[0.3, 0.2, 0.1], noise_
->frequency=[5, 10, 50])
>>>
>>> pd.DataFrame({ "Raw": distorted, "Loess_1": nk.fit_loess(distorted, order=1),
...                "Loess_2": nk.fit_loess(distorted, order=2)}).plot()
```

References

- <https://simplyor.netlify.com/loess-from-scratch-in-python-animation.en-us/>

fit_mixture (*X=None, n_clusters=2*)

Gaussian Mixture Model.

Performs a polynomial regression of given order.

Parameters

- **X** (*Union[list, np.array, pd.Series]*) – The values to classify.
- **n_clusters** (*int*) – Number of components to look for.

Returns *pd.DataFrame* – DataFrame containing the probability of belonging to each cluster.

See also:

`signal_detrend()`, `fit_error()`

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> x = nk.signal_simulate()
>>> probs = nk.fit_mixture(x, n_clusters=2)
>>> fig = nk.signal_plot([x, probs["Cluster_0"], probs["Cluster_1"]],
->standardize=True)
>>> fig
```

fit_mse (*y, y_predicted*)

Compute Mean Square Error (MSE).

fit_polynomial (*y, X=None, order=2*)

Polynomial Regression.

Performs a polynomial regression of given order.

Parameters

- **y** (*Union[list, np.array, pd.Series]*) – The response variable (the y axis).
- **X** (*Union[list, np.array, pd.Series]*) – Explanatory variable (the x axis). If 'None', will treat y as a continuous signal.

- **order** (*int*) – The order of the polynomial. 0, 1 or > 1 for a baseline, linear or polynomial fit, respectively. Can also be 'auto', in which case it will attempt to find the optimal order to minimize the RMSE.

Returns *array* – Prediction of the regression.

See also:

`signal_detrend()`, `fit_error()`, `fit_polynomial_findorder()`

Examples

```
>>> import pandas as pd
>>> import neurokit2 as nk
>>>
>>> y = np.cos(np.linspace(start=0, stop=10, num=100))
>>>
>>> pd.DataFrame({"y": y, "Poly_0": nk.fit_polynomial(y, order=0),
...              "Poly_1": nk.fit_polynomial(y, order=1),
...              "Poly_2": nk.fit_polynomial(y, order=2),
...              "Poly_3": nk.fit_polynomial(y, order=3), "Poly_5": nk.fit_
→ polynomial(y, order=5),
...              "Poly_auto": nk.fit_polynomial(y, order='auto')}).plot()
```

fit_polynomial_findorder (*y*, *X=None*, *max_order=6*)

Polynomial Regression.

Find the optimal order for polynomial fitting. Currently, the only method implemented is RMSE minimization.

Parameters

- **y** (*Union[list, np.array, pd.Series]*) – The response variable (the y axis).
- **X** (*Union[list, np.array, pd.Series]*) – Explanatory variable (the x axis). If 'None', will treat y as a continuous signal.
- **max_order** (*int*) – The maximum order to test.

Returns *int* – Optimal order.

See also:

`fit_polynomial()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> y = np.cos(np.linspace(start=0, stop=10, num=100))
>>>
>>> nk.fit_polynomial_findorder(y, max_order=10)
9
```

fit_r2 (*y*, *y_predicted*, *adjusted=True*, *n_parameters=2*)

Compute R2.

fit_rmse (*y*, *y_predicted*)

Compute Root Mean Square Error (RMSE).

hdi (*x*, *ci*=0.95, *show*=False, ***kwargs*)
Highest Density Interval (HDI)

Compute the Highest Density Interval (HDI) of a distribution. All points within this interval have a higher probability density than points outside the interval. The HDI can be used in the context of uncertainty characterisation of posterior distributions (in the Bayesian framework) as Credible Interval (CI). Unlike equal-tailed intervals that typically exclude 2.5% from each tail of the distribution and always include the median, the HDI is not equal-tailed and therefore always includes the mode(s) of posterior distributions.

Parameters

- **x** (*Union[list, np.array, pd.Series]*) – A vector of values.
- **ci** (*float*) – Value of probability of the (credible) interval - CI (between 0 and 1) to be estimated. Default to .95 (95%).
- **show** (*bool*) – If True, the function will produce a figure.
- ****kwargs** (*Line2D properties*) – Other arguments to be passed to `density()`.

See also:

`density()`

Returns

- *float(s)* – The HDI low and high limits.
- *fig* – Distribution plot.

Examples

```
>>> import numpy as np
>>> import neurokit2 as nk
>>>
>>> x = np.random.normal(loc=0, scale=1, size=100000)
>>> ci_min, ci_high = nk.hdi(x, ci=0.95, show=True)
```

mad (*x*, *constant*=1.4826)

Median Absolute Deviation: a “robust” version of standard deviation.

Parameters

- **x** (*Union[list, np.array, pd.Series]*) – A vector of values.
- **constant** (*float*) – Scale factor. Use 1.4826 for results similar to default R.

Returns *float* – The MAD.

Examples

```
>>> import neurokit2 as nk
>>> nk.mad([2, 8, 7, 5, 4, 12, 5, 1])
3.7064999999999997
```

References

- https://en.wikipedia.org/wiki/Median_absolute_deviation

mutual_information (*x*, *y*, *method*='varoquaux', *bins*=256, *sigma*=1, *normalized*=True)

Computes the (normalized) mutual information (MI) between two vectors from a joint histogram. The mutual information of two variables is a measure of the mutual dependence between them. More specifically, it quantifies the “amount of information” obtained about one variable by observing the other variable.

Parameters

- **x** (*Union*[*list*, *np.array*, *pd.Series*]) – A vector of values.
- **y** (*Union*[*list*, *np.array*, *pd.Series*]) – A vector of values.
- **method** (*str*) – Method to use. Can either be ‘varoquaux’ or ‘nolitsa’.
- **bins** (*int*) – Number of bins to use while creating the histogram.
- **sigma** (*float*) – Sigma for Gaussian smoothing of the joint histogram. Only used if *method*==‘varoquaux’.
- **normalized** (*bool*) – Compute normalised mutual information. Only used if *method*==‘varoquaux’.

Returns *float* – The computed similarity measure.

Examples

```
>>> import neurokit2 as nk
>>>
>>> x = [3, 3, 5, 1, 6, 3]
>>> y = [5, 3, 1, 3, 4, 5]
>>>
>>> nk.mutual_information(x, y, method="varoquaux")
0.23600751227291816
>>>
>>> nk.mutual_information(x, y, method="nolitsa")
1.4591479170272448
```

References

- Studholme, jhill & jhawkes (1998). “A normalized entropy measure of 3-D medical image alignment”.

in Proc. Medical Imaging 1998, vol. 3338, San Diego, CA, pp. 132-143.

rescale (*data*, *to*=[0, 1])

Rescale data.

Rescale a numeric variable to a new range.

Parameters

- **data** (*Union*[*list*, *np.array*, *pd.Series*]) – Raw data.
- **to** (*list*) – New range of values of the data after rescaling.

Returns *list* – The rescaled values.

Examples

```
>>> import neurokit2 as nk
>>>
>>> nk.rescale(data=[3, 1, 2, 4, 6], to=[0, 1])
[0.4, 0.0, 0.2, 0.6000000000000001, 1.0]
```

standardize (*data*, *robust=False*, *window=None*, ***kwargs*)

Standardization of data.

Performs a standardization of data (Z-scoring), i.e., centering and scaling, so that the data is expressed in terms of standard deviation (i.e., mean = 0, SD = 1) or Median Absolute Deviance (median = 0, MAD = 1).

Parameters

- **data** (*Union[list, np.array, pd.Series]*) – Raw data.
- **robust** (*bool*) – If True, centering is done by subtracting the median from the variables and dividing it by the median absolute deviation (MAD). If False, variables are standardized by subtracting the mean and dividing it by the standard deviation (SD).
- **window** (*int*) – Perform a rolling window standardization, i.e., apply a standardization on a window of the specified number of samples that rolls along the main axis of the signal. Can be used for complex detrending.
- ****kwargs** (*optional*) – Other arguments to be passed to `pandas.rolling()`.

Returns *list* – The standardized values.

Examples

```
>>> import neurokit2 as nk
>>> import pandas as pd
>>>
>>> # Simple example
>>> nk.standardize([3, 1, 2, 4, 6, np.nan])
[...]
>>> nk.standardize([3, 1, 2, 4, 6, np.nan], robust=True)
[...]
>>> nk.standardize(np.array([[1, 2, 3, 4], [5, 6, 7, 8]]).T)
array(...)
>>> nk.standardize(pd.DataFrame({"A": [3, 1, 2, 4, 6, np.nan],
...                               "B": [3, 1, 2, 4, 6, 5]}))
...
      A      B
0     ...    ...
...
>>>
>>> # Rolling standardization of a signal
>>> signal = nk.signal_simulate(frequency=[0.1, 2], sampling_rate=200)
>>> z = nk.standardize(signal, window=200)
>>> nk.signal_plot([signal, z], standardize=True)
```

summary_plot (*x*, ***kwargs*)

Descriptive plot.

Visualize a distribution with density, histogram, boxplot and rugs plots all at once.

Examples

```
>>> import neurokit2 as nk
>>> import numpy as np
>>>
>>> x = np.random.normal(size=100)
>>> fig = nk.summary_plot(x)
>>> fig
```

7.13 Complexity

Submodule for NeuroKit.

complexity_apen (*signal*, *delay=1*, *dimension=2*, *r='default'*, *corrected=False*, ***kwargs*)

Approximate entropy (ApEn)

Python implementations of the approximate entropy (ApEn) and its corrected version (cApEn). Approximate entropy is a technique used to quantify the amount of regularity and the unpredictability of fluctuations over time-series data. The advantages of ApEn include lower computational demand (ApEn can be designed to work for small data samples (< 50 data points) and can be applied in real time) and less sensitive to noise. However, ApEn is heavily dependent on the record length and lacks relative consistency.

This function can be called either via `entropy_approximate()` or `complexity_apen()`, and the corrected version via `complexity_capen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (similarity threshold). It corresponds to the filtering level - max absolute difference between segments. If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **corrected** (*bool*) – If true, will compute corrected ApEn (cApEn), see Porta (2007).
- ****kwargs** – Other arguments.

See also:

`entropy_shannon()`, `entropy_sample()`, `entropy_fuzzy()`

Returns *float* – The approximate entropy as float value.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_approximate(signal)
>>> entropy1
>>> entropy2 = nk.entropy_approximate(signal, corrected=True)
>>> entropy2
```

References

- *EntroPy* <<https://github.com/raphaelvallat/entropy>>`_
- Sabeti, M., Katebi, S., & Boostani, R. (2009). Entropy and complexity measures for EEG signal classification of schizophrenic and control participants. *Artificial intelligence in medicine*, 47(3), 263-274.
- Shi, B., Zhang, Y., Yuan, C., Wang, S., & Li, P. (2017). Entropy analysis of short-term heartbeat interval time series during regular walking. *Entropy*, 19(10), 568.

complexity_capen (*signal*, *delay=1*, *dimension=2*, *r='default'*, *, *corrected=True*, ***kwargs*)
Approximate entropy (ApEn)

Python implementations of the approximate entropy (ApEn) and its corrected version (cApEn). Approximate entropy is a technique used to quantify the amount of regularity and the unpredictability of fluctuations over time-series data. The advantages of ApEn include lower computational demand (ApEn can be designed to work for small data samples (< 50 data points) and can be applied in real time) and less sensitive to noise. However, ApEn is heavily dependent on the record length and lacks relative consistency.

This function can be called either via `entropy_approximate()` or `complexity_apen()`, and the corrected version via `complexity_capen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (similarity threshold). It corresponds to the filtering level - max absolute difference between segments. If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **corrected** (*bool*) – If true, will compute corrected ApEn (cApEn), see Porta (2007).
- ****kwargs** – Other arguments.

See also:

`entropy_shannon()`, `entropy_sample()`, `entropy_fuzzy()`

Returns *float* – The approximate entropy as float value.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_approximate(signal)
>>> entropy1
>>> entropy2 = nk.entropy_approximate(signal, corrected=True)
>>> entropy2
```

References

- EntropyPy <<https://github.com/raphaelvallat/entropy>>`_
- Sabeti, M., Katebi, S., & Boostani, R. (2009). Entropy and complexity measures for EEG signal classification of schizophrenic and control participants. *Artificial intelligence in medicine*, 47(3), 263-274.
- Shi, B., Zhang, Y., Yuan, C., Wang, S., & Li, P. (2017). Entropy analysis of short-term heartbeat interval time series during regular walking. *Entropy*, 19(10), 568.

complexity_cmse (*signal*, *scale*='default', *dimension*=2, *r*='default', *, *composite*=True, *refined*=False, *fuzzy*=False, *show*=False, ***kwargs*)

Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzysrcmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If 'default', will use `range(len(signal) / (dimension + 10))` (see discussion [here](#)). If 'max', will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted 'm' or 'd', sometimes referred to as 'order'). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If 'default', will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the 'refined' composite MSE (RCMSE; Wu, 2014)
- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- `pyEntropy` <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

complexity_d2 (*signal*, *delay=1*, *dimension=2*, *r=64*, *show=False*)

Correlation Dimension.

Python implementation of the Correlation Dimension D2 of a signal.

This function can be called either via `fractal_correlation()` or `complexity_d2()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).

- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*str or int or list*) – The sequence of radiuses to test. If an integer is passed, will get an exponential sequence ranging from 2.5% to 50% of the distance range. Methods implemented in other packages can be used via setting `r='nolds'` or `r='Corr_Dim'`.
- **show** (*bool*) – Plot of correlation dimension if True. Defaults to False.

Returns **D2** (*float*) – The correlation dimension D2.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>>
>>> fractal1 = nk.fractal_correlation(signal, r="nolds", show=True)
>>> fractal1
>>> fractal2 = nk.fractal_correlation(signal, r=32, show=True)
>>> fractal2
>>>
>>> signal = nk.rsp_simulate(duration=120, sampling_rate=50)
>>>
>>> fractal3 = nk.fractal_correlation(signal, r="nolds", show=True)
>>> fractal3
>>> fractal4 = nk.fractal_correlation(signal, r=32, show=True)
>>> fractal4
```

References

- Bolea, J., Laguna, P., Remartínez, J. M., Rovira, E., Navarro, A., & Bailón, R. (2014). Methodological framework for estimating the correlation dimension in HRV signals. Computational and mathematical methods in medicine, 2014.
- Boon, M. Y., Henry, B. I., Suttle, C. M., & Dain, S. J. (2008). The correlation dimension: A useful objective measure of the transient visual evoked potential?. Journal of vision, 8(1), 6-6.

- [nolds](#)

- [Corr_Dim](#)

complexity_delay (*signal, delay_max=100, method='fraser1986', show=False*)

Estimate optimal Time Delay (tau) for time-delay embedding.

The time delay (Tau) is one of the two critical parameters involved in the construction of the time-delay embedding of a signal.

Several authors suggested different methods to guide the choice of Tau:

- Fraser and Swinney (1986) suggest using the first local minimum of the mutual information between the delayed and non-delayed time series, effectively identifying a value of tau for which they share the least information.
- Theiler (1990) suggested to select Tau where the autocorrelation between the signal and its lagged version at Tau first crosses the value $1/e$.

- Casdagli (1991) suggests instead taking the first zero-crossing of the autocorrelation.
- Rosenstein (1993) suggests to the point close to 40% of the slope of the average displacement from the diagonal (ADFD).

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay_max** (*int*) – The maximum time delay (Tau or lag) to test.
- **method** (*str*) – Correlation method. Can be one of ‘fraser1986’, ‘theiler1990’, ‘casdagli1991’, ‘rosenstein1993’.
- **show** (*bool*) – If true, will plot the mutual information values for each value of tau.

Returns *int* – Optimal time delay.

See also:

`complexity_dimension()`, `complexity_embedding()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Artificial example
>>> signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01)
>>> nk.signal_plot(signal)
>>>
>>> delay = nk.complexity_delay(signal, delay_max=1000, show=True, method=
↳ "fraser1986")
>>> delay = nk.complexity_delay(signal, delay_max=1000, show=True, method=
↳ "theiler1990")
>>> delay = nk.complexity_delay(signal, delay_max=1000, show=True, method=
↳ "casdagli1991")
>>> delay = nk.complexity_delay(signal, delay_max=1000, show=True, method=
↳ "rosenstein1993")
>>>
>>> # Realistic example
>>> ecg = nk.ecg_simulate(duration=60*6, sampling_rate=150)
>>> signal = nk.ecg_rate(nk.ecg_peaks(ecg, sampling_rate=150), sampling_rate=150,
↳ desired_length=len(ecg))
>>> nk.signal_plot(signal)
>>>
>>> delay = nk.complexity_delay(signal, delay_max=1000, show=True)
```

References

- Gautama, T., Mandic, D. P., & Van Hulle, M. M. (2003, April). A differential entropy based method for determining the optimal embedding parameters of a signal. In 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). (Vol. 6, pp. VI-29). IEEE.
- Camplani, M., & Cannas, B. (2009). The role of the embedding dimension and time delay in time series forecasting. IFAC Proceedings Volumes, 42(7), 316-320.
- Rosenstein, M. T., Collins, J. J., & De Luca, C. J. (1994). Reconstruction expansion as a geometry-based framework for choosing proper delay times. Physica-Section D, 73(1), 82-98.

complexity_dfa (*signal*, *windows*='default', *overlap*=True, *integrate*=True, *order*=1, *multifractal*=False, *q*=2, *show*=False)

(Multifractal) Detrended Fluctuation Analysis (DFA or MFDFA)

Python implementation of Detrended Fluctuation Analysis (DFA) or Multifractal DFA of a signal. Detrended fluctuation analysis, much like the Hurst exponent, is used to find long-term statistical dependencies in time series.

This function can be called either via `fractal_dfa()` or `complexity_dfa()`, and its multifractal variant can be directly accessed via `fractal_mfdfa()` or `complexity_mfdfa()`

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **windows** (*list*) – A list containing the lengths of the windows (number of data points in each subseries). Also referred to as ‘lag’ or ‘scale’. If ‘default’, will set it to a logarithmic scale (so that each window scale has the same weight) with a minimum of 4 and maximum of a tenth of the length (to have more than 10 windows to calculate the average fluctuation).
- **overlap** (*bool*) – Defaults to True, where the windows will have a 50% overlap with each other, otherwise non-overlapping windows will be used.
- **integrate** (*bool*) – It is common practice to convert the signal to a random walk (i.e., detrend and integrate, which corresponds to the signal ‘profile’). Note that it leads to the flattening of the signal, which can lead to the loss of some details (see Ihlen, 2012 for an explanation). Note that for strongly anticorrelated signals, this transformation should be applied two times (i.e., provide `np.cumsum(signal - np.mean(signal))` instead of `signal`).
- **order** (*int*) – The order of the polynomial trend, 1 for the linear trend.
- **multifractal** (*bool*) – If true, compute Multifractal Detrended Fluctuation Analysis (MFDFA), in which case the argument `q` is taken into account.
- **q** (*list*) – The sequence of fractal exponents when `multifractal=True`. Must be a sequence between -10 and 10 (note that zero will be removed, since the code does not converge there). Setting `q = 2` (default) gives a result close to a standard DFA. For instance, Ihlen (2012) uses `q=[-5, -3, -1, 0, 1, 3, 5]`.
- **show** (*bool*) – Visualise the trend between the window size and the fluctuations.

Returns **dfa** (*float*) – The DFA coefficient.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=3, noise=0.05)
>>> dfa1 = nk.fractal_dfa(signal, show=True)
>>> dfa1
>>> dfa2 = nk.fractal_mfdfa(signal, q=np.arange(-3, 4), show=True)
>>> dfa2
```

References

- Ihlen, E. A. F. E. (2012). Introduction to multifractal detrended fluctuation analysis in Matlab. *Frontiers in physiology*, 3, 141.
- Hardstone, R., Poil, S. S., Schiavone, G., Jansen, R., Nikulin, V. V., Mansvelder, H. D., & Linkenkaer-Hansen, K. (2012). Detrended fluctuation analysis: a scale-free view on neuronal oscillations. *Frontiers in physiology*, 3, 450.
- [nolds](#)
- [Youtube introduction](#)

complexity_dimension (*signal*, *delay=1*, *dimension_max=20*, *method='afnn'*, *show=False*, *R=10.0*, *A=2.0*, ***kwargs*)
 Estimate optimal Dimension (m) for time-delay embedding.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `complexity_delay()`).
- **dimension_max** (*int*) – The maximum embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’) to test.
- **method** (*str*) – Method can either be `afnn` (average false nearest neighbour) or `fnn` (false nearest neighbour).
- **show** (*bool*) – Visualize the result.
- **R** (*float*) – Relative tolerance (for `fnn` method).
- **A** (*float*) – Absolute tolerance (for `fnn` method)
- ****kwargs** – Other arguments.

Returns *int* – Optimal dimension.

See also:

`complexity_delay()`, `complexity_embedding()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Artificial example
>>> signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01)
>>> delay = nk.complexity_delay(signal, delay_max=500)
>>>
>>> values = nk.complexity_dimension(signal, delay=delay, dimension_max=20,
    ↪ show=True)
```

References

- Cao, L. (1997). Practical method for determining the minimum embedding dimension of a scalar time series. *Physica D: Nonlinear Phenomena*, 110(1-2), 43-50.

complexity_embedding (*signal*, *delay=1*, *dimension=3*, *show=False*)

Time-delay embedding of a time series (a signal)

A dynamical system can be described by a vector of numbers, called its ‘state’, that aims to provide a complete description of the system at some point in time. The set of all possible states is called the ‘state space’.

Takens’s (1981) embedding theorem suggests that a sequence of measurements of a dynamic system includes in itself all the information required to completely reconstruct the state space. Delay coordinate embedding attempts to identify the state s of the system at some time t by searching the past history of observations for similar states, and, by studying the evolution of similar states, infer information about the future of the system.

How to visualize the dynamics of a system? A sequence of state values over time is called a trajectory. Depending on the system, different trajectories can evolve to a common subset of state space called an attractor. The presence and behavior of attractors gives intuition about the underlying dynamical system. We can visualize the system and its attractors by plotting the trajectory of many different initial state values and numerically integrating them to approximate their continuous time evolution on discrete computers.

This function is adapted from [EntroPy](#) and is equivalent to the `delay_embedding()` function from ‘nolds’.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **show** (*bool*) – Plot the reconstructed attractor.

Returns *array* – Embedded time-series, of shape $(n_times - (order - 1) * delay, order)$

See also:

`embedding_delay()`, `embedding_dimension()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Artificial example
>>> signal = nk.signal_simulate(duration=2, frequency=5, noise=0.01)
>>>
>>> embedded = nk.complexity_embedding(signal, delay=50, dimension=2, show=True)
>>> embedded = nk.complexity_embedding(signal, delay=50, dimension=3, show=True)
>>> embedded = nk.complexity_embedding(signal, delay=50, dimension=4, show=True)
>>>
>>> # Realistic example
>>> ecg = nk.ecg_simulate(duration=60*4, sampling_rate=200)
>>> signal = nk.ecg_rate(nk.ecg_peaks(ecg, sampling_rate=200)[0], sampling_
→rate=200, desired_length=len(ecg))
>>>
>>> embedded = nk.complexity_embedding(signal, delay=250, dimension=2, show=True)
>>> embedded = nk.complexity_embedding(signal, delay=250, dimension=3, show=True)
>>> embedded = nk.complexity_embedding(signal, delay=250, dimension=4, show=True)
```

References

- Gautama, T., Mandic, D. P., & Van Hulle, M. M. (2003, April). A differential entropy based method for determining the optimal embedding parameters of a signal. In 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). (Vol. 6, pp. VI-29). IEEE.

complexity_fuzzycmse (*signal*, *scale*='default', *dimension*=2, *r*='default', ***, *composite*=True, *refined*=False, *fuzzy*=True, *show*=False, ***kwargs*)

Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzycmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If 'default', will use `range(len(signal) / (dimension + 10))` (see discussion [here](#)). If 'max', will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted 'm' or 'd', sometimes referred to as 'order'). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If 'default', will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the 'refined' composite MSE (RCMSE; Wu, 2014)

- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, Fuzzy-CMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- `pyEntropy` <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

complexity_fuzzyen (*signal*, *delay=1*, *dimension=2*, *r='default'*, ***kwargs*)
Fuzzy entropy (FuzzyEn)

Python implementations of the fuzzy entropy (FuzzyEn) of a signal.

This function can be called either via `entropy_fuzzy()` or `complexity_fuzzyen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- ****kwargs** – Other arguments.

Returns *float* – The fuzzy entropy as float value.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy = nk.entropy_fuzzy(signal)
>>> entropy
```

complexity_fuzzymse (*signal, scale='default', dimension=2, r='default', composite=False, refined=False, *, fuzzy=True, show=False, **kwargs*)
Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzymrcmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If ‘default’, will use `range(len(signal) / (dimension + 10))` (see discussion [here](#)). If ‘max’, will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).

- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the ‘refined’ composite MSE (RCMSE; Wu, 2014)
- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, Fuzzy-CMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- *pyEntropy* <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

complexity_fuzzycrmse (*signal*, *scale*='default', *dimension*=2, *r*='default', *composite*=False, *, *refined*=True, *fuzzy*=True, *show*=False, ***kwargs*)
Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzymrcmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If ‘default’, will use `range(len(signal) / (dimension + 10))` (see discussion [here](#)). If ‘max’, will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the ‘refined’ composite MSE (RCMSE; Wu, 2014)
- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- *pyEntropy* <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

complexity_mfdfa (*signal*, *windows='default'*, *overlap=True*, *integrate=True*, *order=1*, ***, *multifractal=True*, *q=2*, *show=False*)
(Multifractal) Detrended Fluctuation Analysis (DFA or MFDFA)

Python implementation of Detrended Fluctuation Analysis (DFA) or Multifractal DFA of a signal. Detrended fluctuation analysis, much like the Hurst exponent, is used to find long-term statistical dependencies in time series.

This function can be called either via `fractal_dfa()` or `complexity_dfa()`, and its multifractal variant can be directly accessed via `fractal_mfdfa()` or `complexity_mfdfa()`

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **windows** (*list*) – A list containing the lengths of the windows (number of data points in each subseries). Also referred to as ‘lag’ or ‘scale’. If ‘default’, will set it to a logarithmic scale (so that each window scale has the same weight) with a minimum of 4 and maximum of a tenth of the length (to have more than 10 windows to calculate the average fluctuation).
- **overlap** (*bool*) – Defaults to True, where the windows will have a 50% overlap with each other, otherwise non-overlapping windows will be used.
- **integrate** (*bool*) – It is common practice to convert the signal to a random walk (i.e., detrend and integrate, which corresponds to the signal ‘profile’). Note that it leads to the flattening of the signal, which can lead to the loss of some details (see Ihlen, 2012 for an explanation). Note that for strongly anticorrelated signals, this transformation should be applied two times (i.e., provide `np.cumsum(signal - np.mean(signal))` instead of `signal`).
- **order** (*int*) – The order of the polynomial trend, 1 for the linear trend.
- **multifractal** (*bool*) – If true, compute Multifractal Detrended Fluctuation Analysis (MFDFA), in which case the argument `q` is taken into account.
- **q** (*list*) – The sequence of fractal exponents when `multifractal=True`. Must be a sequence between -10 and 10 (note that zero will be removed, since the code does

not converge there). Setting $q = 2$ (default) gives a result close to a standard DFA. For instance, Ihlen (2012) uses `q=[-5, -3, -1, 0, 1, 3, 5]`.

- **show** (*bool*) – Visualise the trend between the window size and the fluctuations.

Returns **dfa** (*float*) – The DFA coefficient.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=3, noise=0.05)
>>> dfa1 = nk.fractal_dfa(signal, show=True)
>>> dfa1
>>> dfa2 = nk.fractal_mdfa(signal, q=np.arange(-3, 4), show=True)
>>> dfa2
```

References

- Ihlen, E. A. F. E. (2012). Introduction to multifractal detrended fluctuation analysis in Matlab. *Frontiers in physiology*, 3, 141.
- Hardstone, R., Poil, S. S., Schiavone, G., Jansen, R., Nikulin, V. V., Mansvelder, H. D., & Linkenkaer-Hansen, K. (2012). Detrended fluctuation analysis: a scale-free view on neuronal oscillations. *Frontiers in physiology*, 3, 450.
- [nolds](#)
- [Youtube introduction](#)

complexity_mse (*signal*, *scale*='default', *dimension*=2, *r*='default', *composite*=False, *refined*=False, *fuzzy*=False, *show*=False, ***kwargs*)

Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzysrcmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If 'default', will use `range(len(signal) / (dimension + 10))` (see discussion [here](#)). If 'max', will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted 'm' or 'd', sometimes referred to as 'order'). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If 'default', will be set to 0.2 times the standard deviation of the signal (for dimension = 2).

- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the ‘refined’ composite MSE (RCMSE; Wu, 2014)
- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, Fuzzy-CMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- *pyEntropy* <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

complexity_optimize (*signal*, *delay_max=100*, *delay_method='fraser1986'*, *dimension_max=20*, *dimension_method='afnn'*, *r_method='maxApEn'*, *show=False*)

Find optimal complexity parameters.

Estimate optimal complexity parameters Dimension (m), Time Delay (tau) and tolerance 'r'.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay_max** (*int*) – See `complexity_delay()`.
- **delay_method** (*str*) – See `complexity_delay()`.
- **dimension_max** (*int*) – See `complexity_dimension()`.
- **dimension_method** (*str*) – See `complexity_dimension()`.
- **r_method** (*str*) – See `complexity_r()`.
- **show** (*bool*) – Defaults to False.

Returns

- **optimal_dimension** (*int*) – Optimal dimension.
- **optimal_delay** (*int*) – Optimal time delay.

See also:

`complexity_dimension()`, `complexity_delay()`, `complexity_r()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Artificial example
>>> signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01)
>>> parameters = nk.complexity_optimize(signal, show=True)
>>> parameters
```

References

- Gautama, T., Mandic, D. P., & Van Hulle, M. M. (2003, April). A differential entropy based method for determining the optimal embedding parameters of a signal. In 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). (Vol. 6, pp. VI-29). IEEE.
- Camplani, M., & Cannas, B. (2009). The role of the embedding dimension and time delay in time series forecasting. IFAC Proceedings Volumes, 42(7), 316-320.
- Rosenstein, M. T., Collins, J. J., & De Luca, C. J. (1994). Reconstruction expansion as a geometry-based framework for choosing proper delay times. Physica-Section D, 73(1), 82-98.
- Cao, L. (1997). Practical method for determining the minimum embedding dimension of a scalar time series. Physica D: Nonlinear Phenomena, 110(1-2), 43-50.
- Lu, S., Chen, X., Kanters, J. K., Solomon, I. C., & Chon, K. H. (2008). Automatic selection of the threshold value r for approximate entropy. IEEE Transactions on Biomedical Engineering, 55(8), 1966-1972.

complexity_plot (*signal*, *delay_max=100*, *delay_method='fraser1986'*, *dimension_max=20*, *dimension_method='afun'*, *r_method='maxApEn'*, ***, *show=True*)

Find optimal complexity parameters.

Estimate optimal complexity parameters Dimension (m), Time Delay (tau) and tolerance 'r'.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay_max** (*int*) – See `complexity_delay()`.
- **delay_method** (*str*) – See `complexity_delay()`.
- **dimension_max** (*int*) – See `complexity_dimension()`.
- **dimension_method** (*str*) – See `complexity_dimension()`.
- **r_method** (*str*) – See `complexity_r()`.
- **show** (*bool*) – Defaults to False.

Returns

- **optimal_dimension** (*int*) – Optimal dimension.
- **optimal_delay** (*int*) – Optimal time delay.

See also:

`complexity_dimension()`, `complexity_delay()`, `complexity_r()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Artificial example
>>> signal = nk.signal_simulate(duration=10, frequency=1, noise=0.01)
>>> parameters = nk.complexity_optimize(signal, show=True)
>>> parameters
```

References

- Gautama, T., Mandic, D. P., & Van Hulle, M. M. (2003, April). A differential entropy based method for determining the optimal embedding parameters of a signal. In 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). (Vol. 6, pp. VI-29). IEEE.
- Camplani, M., & Cannas, B. (2009). The role of the embedding dimension and time delay in time series forecasting. IFAC Proceedings Volumes, 42(7), 316-320.
- Rosenstein, M. T., Collins, J. J., & De Luca, C. J. (1994). Reconstruction expansion as a geometry-based framework for choosing proper delay times. Physica-Section D, 73(1), 82-98.
- Cao, L. (1997). Practical method for determining the minimum embedding dimension of a scalar time series. Physica D: Nonlinear Phenomena, 110(1-2), 43-50.
- Lu, S., Chen, X., Kanters, J. K., Solomon, I. C., & Chon, K. H. (2008). Automatic selection of the threshold value r for approximate entropy. IEEE Transactions on Biomedical Engineering, 55(8), 1966-1972.

complexity_r (*signal, delay=None, dimension=None, method='maxApEn', show=False*)

Estimate optimal tolerance (similarity threshold) :Parameters: * **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.

- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **method** (*str*) – If ‘maxApEn’, rmax where ApEn is max will be returned. If ‘traditional’, $r = 0.2 * \text{standard deviation of the signal}$ will be returned.
- **show** (*bool*) – If true and method is ‘maxApEn’, will plot the ApEn values for each value of r.

Returns *float* – The optimal r as a similarity threshold. It corresponds to the filtering level - max absolute difference between segments.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> delay = nk.complexity_delay(signal)
>>> dimension = nk.complexity_dimension(signal, delay=delay)
>>> r = nk.complexity_r(signal, delay, dimension)
>>> r
```

References

- Lu, S., Chen, X., Kanters, J. K., Solomon, I. C., & Chon, K. H. (2008). Automatic selection of the threshold value r for approximate entropy. *IEEE Transactions on Biomedical Engineering*, 55(8), 1966-1972.

complexity_rcmse (*signal*, *scale*='default', *dimension*=2, *r*='default', *composite*=False, *, *refined*=True, *fuzzy*=False, *show*=False, ***kwargs*)

Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzyrcmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If ‘default’, will use $\text{range}(\text{len}(\text{signal}) / (\text{dimension} + 10))$ (see discussion [here](#)). If ‘max’, will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.

- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the ‘refined’ composite MSE (RCMSE; Wu, 2014)
- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, Fuzzy-CMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- `pyEntropy` <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

complexity_sampen (*signal*, *delay=1*, *dimension=2*, *r='default'*, ***kwargs*)
Sample Entropy (SampEn)

Python implementation of the sample entropy (SampEn) of a signal.

This function can be called either via `entropy_sample()` or `complexity_sampen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- ****kwargs** (*optional*) – Other arguments.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_fuzzy()`

Returns *float* – The sample entropy as float value.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy = nk.entropy_sample(signal)
>>> entropy
```

complexity_se (*signal*)
Shannon entropy (SE)

Python implementation of Shannon entropy (SE). Entropy is a measure of unpredictability of the state, or equivalently, of its average information content. Shannon entropy (SE) is one of the first and most basic measure of entropy and a foundational concept of information theory. Shannon’s entropy quantifies the amount of information in a variable.

This function can be called either via `entropy_shannon()` or `complexity_se()`.

Parameters **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.

Returns *float* – The Shannon entropy as float value.

See also:

`entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy = nk.entropy_shannon(signal)
>>> entropy
```

References

- *pyEntropy* <<https://github.com/nikdon/pyEntropy>>`_
- *EntroPy* <<https://github.com/raphaelvallat/entropy>>`_
- *nolds* <<https://github.com/CSchoel/nolds>>`_

complexity_simulate (*duration=10, sampling_rate=1000, method='ornstein', hurst_exponent=0.5, **kwargs*)

Simulate chaotic time series.

Generates time series using the discrete approximation of the Mackey-Glass delay differential equation described by Grassberger & Procaccia (1983).

Parameters

- **duration** (*int*) – Desired length of duration (s).
- **sampling_rate** (*int*) – The desired sampling rate (in Hz, i.e., samples/second).
- **duration** (*int*) – The desired length in samples.
- **method** (*str*) – The method. can be ‘hurst’ for a (fractional) Ornstein–Uhlenbeck process or ‘mackeyglass’ to use the Mackey–Glass equation.
- **hurst_exponent** (*float*) – Defaults to 0.5.
- ****kwargs** – Other arguments.

Returns *array* – Simulated complexity time series.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal1 = nk.complexity_simulate(duration=30, sampling_rate=100, method=
↳ "ornstein")
>>> signal2 = nk.complexity_simulate(duration=30, sampling_rate=100, method=
↳ "mackeyglass")
>>> nk.signal_plot([signal1, signal2])
```

Returns *x (array)* – Array containing the time series.

entropy_approximate (*signal, delay=1, dimension=2, r='default', corrected=False, **kwargs*)

Approximate entropy (ApEn)

Python implementations of the approximate entropy (ApEn) and its corrected version (cApEn). Approximate entropy is a technique used to quantify the amount of regularity and the unpredictability of fluctuations over time-series data. The advantages of ApEn include lower computational demand (ApEn can be designed to work

for small data samples (< 50 data points) and can be applied in real time) and less sensitive to noise. However, ApEn is heavily dependent on the record length and lacks relative consistency.

This function can be called either via `entropy_approximate()` or `complexity_apen()`, and the corrected version via `complexity_capen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (similarity threshold). It corresponds to the filtering level - max absolute difference between segments. If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **corrected** (*bool*) – If true, will compute corrected ApEn (cApEn), see Porta (2007).
- ****kwargs** – Other arguments.

See also:

`entropy_shannon()`, `entropy_sample()`, `entropy_fuzzy()`

Returns *float* – The approximate entropy as float value.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_approximate(signal)
>>> entropy1
>>> entropy2 = nk.entropy_approximate(signal, corrected=True)
>>> entropy2
```

References

- EntropyPy <<https://github.com/raphaelvallat/entropy>>`_
- Sabeti, M., Katebi, S., & Boostani, R. (2009). Entropy and complexity measures for EEG signal classification of schizophrenic and control participants. *Artificial intelligence in medicine*, 47(3), 263-274.
- Shi, B., Zhang, Y., Yuan, C., Wang, S., & Li, P. (2017). Entropy analysis of short-term heartbeat interval time series during regular walking. *Entropy*, 19(10), 568.

entropy_fuzzy (*signal, delay=1, dimension=2, r='default', **kwargs*)
Fuzzy entropy (FuzzyEn)

Python implementations of the fuzzy entropy (FuzzyEn) of a signal.

This function can be called either via `entropy_fuzzy()` or `complexity_fuzzyen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- ****kwargs** – Other arguments.

Returns *float* – The fuzzy entropy as float value.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy = nk.entropy_fuzzy(signal)
>>> entropy
```

entropy_multiscale (*signal, scale='default', dimension=2, r='default', composite=False, refined=False, fuzzy=False, show=False, **kwargs*)

Multiscale entropy (MSE) and its Composite (CMSE), Refined (RCMSE) or fuzzy version.

Python implementations of the multiscale entropy (MSE), the composite multiscale entropy (CMSE), the refined composite multiscale entropy (RCMSE) or their fuzzy version (FuzzyMSE, FuzzyCMSE or FuzzyRCMSE).

This function can be called either via `entropy_multiscale()` or `complexity_mse()`. Moreover, variants can be directly accessed via `complexity_cmse()`, `complexity_rcmse()`, `complexity_fuzzymse()` and `complexity_fuzzysrcmse()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **scale** (*str or int or list*) – A list of scale factors used for coarse graining the time series. If ‘default’, will use `range(len(signal) / (dimension + 10))` (see discussion [here](#)). If ‘max’, will use all scales until half the length of the signal. If an integer, will create a range until the specified int.
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.

- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- **composite** (*bool*) – Returns the composite multiscale entropy (CMSE), more accurate than MSE.
- **refined** (*bool*) – Returns the ‘refined’ composite MSE (RCMSE; Wu, 2014)
- **fuzzy** (*bool*) – Returns the fuzzy (composite) multiscale entropy (FuzzyMSE, Fuzzy-CMSE or FuzzyRCMSE).
- **show** (*bool*) – Show the entropy values for each scale factor.
- ****kwargs** – Optional arguments.

Returns *float* – The point-estimate of multiscale entropy (MSE) as a float value corresponding to the area under the MSE values curvee, which is essentially the sum of sample entropy values over the range of scale factors.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy1 = nk.entropy_multiscale(signal, show=True)
>>> entropy1
>>> entropy2 = nk.entropy_multiscale(signal, show=True, composite=True)
>>> entropy2
>>> entropy3 = nk.entropy_multiscale(signal, show=True, refined=True)
>>> entropy3
```

References

- `pyEntropy` <<https://github.com/nikdon/pyEntropy>>`_
- Richman, J. S., & Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), H2039-H2049.
- Costa, M., Goldberger, A. L., & Peng, C. K. (2005). Multiscale entropy analysis of biological signals. *Physical review E*, 71(2), 021906.
- Gow, B. J., Peng, C. K., Wayne, P. M., & Ahn, A. C. (2015). Multiscale entropy analysis of center-of-pressure dynamics in human postural control: methodological considerations. *Entropy*, 17(12), 7926-7947.
- Norris, P. R., Anderson, S. M., Jenkins, J. M., Williams, A. E., & Morris Jr, J. A. (2008). Heart rate multiscale entropy at three hours predicts hospital mortality in 3,154 trauma patients. *Shock*, 30(1), 17-22.
- Liu, Q., Wei, Q., Fan, S. Z., Lu, C. W., Lin, T. Y., Abbod, M. F., & Shieh, J. S. (2012). Adaptive computation of multiscale entropy and its application in EEG signals for monitoring depth of anesthesia during surgery. *Entropy*, 14(6), 978-992.

entropy_sample (*signal*, *delay=1*, *dimension=2*, *r='default'*, ***kwargs*)
Sample Entropy (SampEn)

Python implementation of the sample entropy (SampEn) of a signal.

This function can be called either via `entropy_sample()` or `complexity_sampen()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*float*) – Tolerance (i.e., filtering level - max absolute difference between segments). If ‘default’, will be set to 0.2 times the standard deviation of the signal (for dimension = 2).
- ****kwargs** (*optional*) – Other arguments.

See also:

`entropy_shannon()`, `entropy_approximate()`, `entropy_fuzzy()`

Returns *float* – The sample entropy as float value.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy = nk.entropy_sample(signal)
>>> entropy
```

entropy_shannon (*signal*)
Shannon entropy (SE)

Python implementation of Shannon entropy (SE). Entropy is a measure of unpredictability of the state, or equivalently, of its average information content. Shannon entropy (SE) is one of the first and most basic measure of entropy and a foundational concept of information theory. Shannon’s entropy quantifies the amount of information in a variable.

This function can be called either via `entropy_shannon()` or `complexity_se()`.

Parameters **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.

Returns *float* – The Shannon entropy as float value.

See also:

`entropy_approximate()`, `entropy_sample()`, `entropy_fuzzy()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>> entropy = nk.entropy_shannon(signal)
>>> entropy
```

References

- *pyEntropy* <<https://github.com/nikdon/pyEntropy>>`_
- *EntroPy* <<https://github.com/raphaelvallat/entropy>>`_
- *nolds* <<https://github.com/CSchoel/nolds>>`_

fractal_correlation (*signal*, *delay=1*, *dimension=2*, *r=64*, *show=False*)
Correlation Dimension.

Python implementation of the Correlation Dimension D2 of a signal.

This function can be called either via `fractal_correlation()` or `complexity_d2()`.

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **delay** (*int*) – Time delay (often denoted ‘Tau’, sometimes referred to as ‘lag’). In practice, it is common to have a fixed time lag (corresponding for instance to the sampling rate; Gautama, 2003), or to find a suitable value using some algorithmic heuristics (see `delay_optimal()`).
- **dimension** (*int*) – Embedding dimension (often denoted ‘m’ or ‘d’, sometimes referred to as ‘order’). Typically 2 or 3. It corresponds to the number of compared runs of lagged data. If 2, the embedding returns an array with two columns corresponding to the original signal and its delayed (by Tau) version.
- **r** (*str or int or list*) – The sequence of radiuses to test. If an integer is passed, will get an exponential sequence ranging from 2.5% to 50% of the distance range. Methods implemented in other packages can be used via setting `r='nolds'` or `r='Corr_Dim'`.
- **show** (*bool*) – Plot of correlation dimension if True. Defaults to False.

Returns **D2** (*float*) – The correlation dimension D2.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=2, frequency=5)
>>>
>>> fractal1 = nk.fractal_correlation(signal, r="nolds", show=True)
>>> fractal1
>>> fractal2 = nk.fractal_correlation(signal, r=32, show=True)
>>> fractal2
>>>
```

(continues on next page)

(continued from previous page)

```

>>> signal = nk.rsp_simulate(duration=120, sampling_rate=50)
>>>
>>> fractal3 = nk.fractal_correlation(signal, r="nolds", show=True)
>>> fractal3
>>> fractal4 = nk.fractal_correlation(signal, r=32, show=True)
>>> fractal4

```

References

- Bolea, J., Laguna, P., Remartínez, J. M., Rovira, E., Navarro, A., & Bailón, R. (2014). Methodological framework for estimating the correlation dimension in HRV signals. *Computational and mathematical methods in medicine*, 2014.
- Boon, M. Y., Henry, B. I., Suttle, C. M., & Dain, S. J. (2008). The correlation dimension: A useful objective measure of the transient visual evoked potential?. *Journal of vision*, 8(1), 6-6.
- [nolds](#)
- [Corr_Dim](#)

fractal_dfa (*signal*, *windows*='default', *overlap*=True, *integrate*=True, *order*=1, *multifractal*=False, *q*=2, *show*=False)
(Multifractal) Detrended Fluctuation Analysis (DFA or MFDFA)

Python implementation of Detrended Fluctuation Analysis (DFA) or Multifractal DFA of a signal. Detrended fluctuation analysis, much like the Hurst exponent, is used to find long-term statistical dependencies in time series.

This function can be called either via `fractal_dfa()` or `complexity_dfa()`, and its multifractal variant can be directly accessed via `fractal_mfdfa()` or `complexity_mfdfa()`

Parameters

- **signal** (*Union[list, np.array, pd.Series]*) – The signal (i.e., a time series) in the form of a vector of values.
- **windows** (*list*) – A list containing the lengths of the windows (number of data points in each subseries). Also referred to as ‘lag’ or ‘scale’. If ‘default’, will set it to a logarithmic scale (so that each window scale has the same weight) with a minimum of 4 and maximum of a tenth of the length (to have more than 10 windows to calculate the average fluctuation).
- **overlap** (*bool*) – Defaults to True, where the windows will have a 50% overlap with each other, otherwise non-overlapping windows will be used.
- **integrate** (*bool*) – It is common practice to convert the signal to a random walk (i.e., detrend and integrate, which corresponds to the signal ‘profile’). Note that it leads to the flattening of the signal, which can lead to the loss of some details (see Ihlen, 2012 for an explanation). Note that for strongly anticorrelated signals, this transformation should be applied two times (i.e., provide `np.cumsum(signal - np.mean(signal))` instead of `signal`).
- **order** (*int*) – The order of the polynomial trend, 1 for the linear trend.
- **multifractal** (*bool*) – If true, compute Multifractal Detrended Fluctuation Analysis (MFDFA), in which case the argument `q` is taken into account.
- **q** (*list*) – The sequence of fractal exponents when `multifractal=True`. Must be a sequence between -10 and 10 (note that zero will be removed, since the code does

not converge there). Setting $q = 2$ (default) gives a result close to a standard DFA. For instance, Ihlen (2012) uses `q=[-5, -3, -1, 0, 1, 3, 5]`.

- **show** (*bool*) – Visualise the trend between the window size and the fluctuations.

Returns **dfa** (*float*) – The DFA coefficient.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=3, noise=0.05)
>>> dfal = nk.fractal_dfa(signal, show=True)
>>> dfal
>>> dfa2 = nk.fractal_mfdfa(signal, q=np.arange(-3, 4), show=True)
>>> dfa2
```

References

- Ihlen, E. A. F. E. (2012). Introduction to multifractal detrended fluctuation analysis in Matlab. *Frontiers in physiology*, 3, 141.
- Hardstone, R., Poil, S. S., Schiavone, G., Jansen, R., Nikulin, V. V., Mansvelder, H. D., & Linkenkaer-Hansen, K. (2012). Detrended fluctuation analysis: a scale-free view on neuronal oscillations. *Frontiers in physiology*, 3, 450.
- [nolds](#)
- [Youtube introduction](#)

fractal_mandelbrot (*size=1000, real_range=- 2, 2, imaginary_range=- 2, 2, threshold=4, iterations=25, buddha=False, show=False*)

Generate a Mandelbrot (or a Buddhabrot) fractal.

Vectorized function to efficiently generate an array containing values corresponding to a Mandelbrot fractal.

Parameters

- **size** (*int*) – The size in pixels (corresponding to the width of the figure).
- **real_range** (*tuple*) – The mandelbrot set is defined within the -2, 2 complex space (the real being the x-axis and the imaginary the y-axis). Adjusting these ranges can be used to pan, zoom and crop the figure.
- **imaginary_range** (*tuple*) – The mandelbrot set is defined within the -2, 2 complex space (the real being the x-axis and the imaginary the y-axis). Adjusting these ranges can be used to pan, zoom and crop the figure.
- **iterations** (*int*) – Number of iterations.
- **threshold** (*int*) – The threshold used, increasing it will increase the sharpness (not used for buddhabrots).
- **buddha** (*bool*) – Whether to return a buddhabrot.
- **show** (*bool*) – Visualize the fractal.

Returns *fig* – Plot of fractal.

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Mandelbrot fractal
>>> nk.fractal_mandelbrot(show=True)
array(...)

>>> # Zoom at seahorse valley
>>> nk.fractal_mandelbrot(real_range=(-0.76, -0.74), imaginary_range=(0.09, 0.11),
...                        iterations=100, show=True)
array(...)
>>>
>>> # Draw manually
>>> m = nk.fractal_mandelbrot(real_range=(-2, 0.75), imaginary_range=(-1.25, 1.
->25))
>>> plt.imshow(m.T, cmap="viridis")
>>> plt.axis("off")
>>> plt.show()
>>>
>>> # Buddhabrot
>>> b = nk.fractal_mandelbrot(size=1500, real_range=(-2, 0.75), imaginary_range=(-
->1.25, 1.25),
...                           buddha=True, iterations=200)
>>> plt.imshow(b.T, cmap="gray")
>>> plt.axis("off")
>>> plt.show()
>>>
>>> # Mixed
>>> m = nk.fractal_mandelbrot()
>>> b = nk.fractal_mandelbrot(buddha=True, iterations=200)
>>>
>>> mixed = m - b
>>> plt.imshow(mixed.T, cmap="gray")
>>> plt.axis("off")
>>> plt.show()
```

fractal_mfdfa (*signal*, *windows*='default', *overlap*=True, *integrate*=True, *order*=1, *, *multifractal*=True, *q*=2, *show*=False)
(Multifractal) Detrended Fluctuation Analysis (DFA or MFDFA)

Python implementation of Detrended Fluctuation Analysis (DFA) or Multifractal DFA of a signal. Detrended fluctuation analysis, much like the Hurst exponent, is used to find long-term statistical dependencies in time series.

This function can be called either via `fractal_dfa()` or `complexity_dfa()`, and its multifractal variant can be directly accessed via `fractal_mfdfa()` or `complexity_mfdfa()`

Parameters

- **signal** (*Union*[*list*, *np.array*, *pd.Series*]) – The signal (i.e., a time series) in the form of a vector of values.
- **windows** (*list*) – A list containing the lengths of the windows (number of data points in each subseries). Also referred to as ‘lag’ or ‘scale’. If ‘default’, will set it to a logarithmic scale (so that each window scale has the same weight) with a minimum of 4 and maximum of a tenth of the length (to have more than 10 windows to calculate the average fluctuation).

- **overlap** (*bool*) – Defaults to True, where the windows will have a 50% overlap with each other, otherwise non-overlapping windows will be used.
- **integrate** (*bool*) – It is common practice to convert the signal to a random walk (i.e., detrend and integrate, which corresponds to the signal ‘profile’). Note that it leads to the flattening of the signal, which can lead to the loss of some details (see Ihlen, 2012 for an explanation). Note that for strongly anticorrelated signals, this transformation should be applied two times (i.e., provide `np.cumsum(signal - np.mean(signal))` instead of `signal`).
- **order** (*int*) – The order of the polynomial trend, 1 for the linear trend.
- **multifractal** (*bool*) – If true, compute Multifractal Detrended Fluctuation Analysis (MFDFA), in which case the argument `q` is taken into account.
- **q** (*list*) – The sequence of fractal exponents when `multifractal=True`. Must be a sequence between -10 and 10 (note that zero will be removed, since the code does not converge there). Setting `q = 2` (default) gives a result close to a standard DFA. For instance, Ihlen (2012) uses `q=[-5, -3, -1, 0, 1, 3, 5]`.
- **show** (*bool*) – Visualise the trend between the window size and the fluctuations.

Returns `dfa` (*float*) – The DFA coefficient.

Examples

```
>>> import neurokit2 as nk
>>>
>>> signal = nk.signal_simulate(duration=3, noise=0.05)
>>> dfa1 = nk.fractal_dfa(signal, show=True)
>>> dfa1
>>> dfa2 = nk.fractal_mdfa(signal, q=np.arange(-3, 4), show=True)
>>> dfa2
```

References

- Ihlen, E. A. F. E. (2012). Introduction to multifractal detrended fluctuation analysis in Matlab. *Frontiers in physiology*, 3, 141.
- Hardstone, R., Poil, S. S., Schiavone, G., Jansen, R., Nikulin, V. V., Mansvelder, H. D., & Linkenkaer-Hansen, K. (2012). Detrended fluctuation analysis: a scale-free view on neuronal oscillations. *Frontiers in physiology*, 3, 450.
- [nolds](#)
- [Youtube introduction](#)

7.14 Miscellaneous

Submodule for NeuroKit.

bio_analyze (*data*, *sampling_rate=1000*, *method='auto'*)

Automated analysis of bio signals.

Wrapper for other bio analyze functions of electrocardiography signals (ECG), respiration signals (RSP), electrodermal activity (EDA) and electromyography signals (EMG).

Parameters

- **data** (*DataFrame*) – The DataFrame containing all the processed signals, typically produced by *bio_process()*, *ecg_process()*, *rsp_process()*, *eda_process()*, or *emg_process()*.
- **sampling_rate** (*int*) – The sampling frequency of the signals (in Hz, i.e., samples/second). Defaults to 1000.
- **method** (*str*) – Can be one of ‘event-related’ for event-related analysis on epochs, or ‘interval-related’ for analysis on longer periods of data. Defaults to ‘auto’ where the right method will be chosen based on the mean duration of the data (‘event-related’ for duration under 10s).

Returns *DataFrame* – DataFrame of the analyzed bio features. See docstrings of *ecg_analyze()*, *rsp_analyze()*, *eda_analyze()* and *emg_analyze()* for more details. Also returns Respiratory Sinus Arrhythmia features produced by *ecg_rsa()* if interval-related analysis is carried out.

See also:

`ecg_analyze()`, `rsp_analyze()`, `eda_analyze()`, `emg_analyze()`

Examples

```
>>> import neurokit2 as nk
>>>
>>> # Example 1: Event-related analysis
>>> # Download data
>>> data = nk.data("bio_eventrelated_100hz")
>>>
>>> # Process the data
>>> df, info = nk.bio_process(ecg=data["ECG"], rsp=data["RSP"], eda=data["EDA"],
...                           keep=data["Photosensor"], sampling_rate=100)
>>>
>>> # Build epochs
>>> events = nk.events_find(data["Photosensor"], threshold_keep='below',
...                          event_conditions=["Negative", "Neutral",
...                                              "Neutral", "Negative"])
>>> epochs = nk.epochs_create(df, events, sampling_rate=100, epochs_start=-0.1,
...                           epochs_end=1.9)
>>>
>>> # Analyze
>>> nk.bio_analyze(epochs, sampling_rate=100)
>>>
>>> # Example 2: Interval-related analysis
>>> # Download data
>>> data = nk.data("bio_resting_5min_100hz")
>>>
>>> # Process the data
```

(continues on next page)

(continued from previous page)

```
>>> df, info = nk.bio_process(ecg=data["ECG"], rsp=data["RSP"], sampling_rate=100)
>>>
>>> # Analyze
>>> nk.bio_analyze(df, sampling_rate=100)
```

bio_process (*ecg=None, rsp=None, eda=None, emg=None, keep=None, sampling_rate=1000*)

Automated processing of bio signals.

Wrapper for other bio processing functions of electrocardiography signals (ECG), respiration signals (RSP), electrodermal activity (EDA) and electromyography signals (EMG).

Parameters

- **data** (*DataFrame* # *pylint: disable=W0611*) – The DataFrame containing all the respective signals (e.g., *ecg*, *rsp*, Photosensor etc.). If provided, there is no need to fill in the other arguments denoting the channel inputs. Defaults to *None*.
- **ecg** (*Union[list, np.array, pd.Series]*) – The raw ECG channel.
- **rsp** (*Union[list, np.array, pd.Series]*) – The raw RSP channel (as measured, for instance, by a respiration belt).
- **eda** (*Union[list, np.array, pd.Series]*) – The raw EDA channel.
- **emg** (*Union[list, np.array, pd.Series]*) – The raw EMG channel.
- **keep** (*DataFrame*) – Dataframe or channels to add by concatenation to the processed dataframe (for instance, the Photosensor channel).
- **sampling_rate** (*int*) – The sampling frequency of the signals (in Hz, i.e., samples/second). Defaults to 1000.

Returns

- **bio_df** (*DataFrame*) – DataFrames of the following processed bio features: - “*ECG*”: the raw signal, the cleaned signal, the heart rate, and the R peaks indexes. Also generated by *ecg_process()*. - “*RSP*”: the raw signal, the cleaned signal, the rate, and the amplitude. Also generated by *rsp_process()*. - “*EDA*”: the raw signal, the cleaned signal, the tonic component, the phasic component, indexes of the SCR onsets, peaks, amplitudes, and half-recovery times. Also generated by *eda_process()*. - “*EMG*”: the raw signal, the cleaned signal, and amplitudes. Also generated by *emg_process()*. - “*RSA*”: Respiratory Sinus Arrhythmia features generated by *ecg_rsa()*, if both ECG and RSP are provided.
- **bio_info** (*dict*) – A dictionary containing the samples of peaks, troughs, amplitudes, onsets, offsets, periods of activation, recovery times of the respective processed signals.

See also:

`ecg_process()`, `rsp_process()`, `eda_process()`, `emg_process()`

Example

```
>>> import neurokit2 as nk
>>>
>>> ecg = nk.ecg_simulate(duration=30, sampling_rate=250)
>>> rsp = nk.rsp_simulate(duration=30, sampling_rate=250)
>>> eda = nk.eda_simulate(duration=30, sampling_rate=250, scr_number=3)
>>> emg = nk.emg_simulate(duration=30, sampling_rate=250, burst_number=3)
>>>
>>> bio_df, bio_info = nk.bio_process(ecg=ecg, rsp=rsp, eda=eda, emg=emg,
→sampling_rate=250)
>>>
>>> # Visualize all signals
>>> fig = nk.standardize(bio_df).plot(subplots=True)
>>> fig
```

Submodule for NeuroKit.

as_vector (*x*)

Convert to vector.

Examples

```
>>> import neurokit2 as nk
>>>
>>> x = nk.as_vector(x=range(3))
>>> y = nk.as_vector(x=[0, 1, 2])
>>> z = nk.as_vector(x=np.array([0, 1, 2]))
>>> z
>>>
>>> x = nk.as_vector(x=0)
>>> x
>>>
>>> x = nk.as_vector(x=pd.Series([0, 1, 2]))
>>> y = nk.as_vector(x=pd.DataFrame([0, 1, 2]))
>>> y
>>>
```

expspace (*start*, *stop*, *num*=50, *base*=1)

Exponential range.

Creates a list of integer values of a given length from start to stop, spread by an exponential function.

Parameters

- **start** (*int*) – Minimum range values.
- **stop** (*int*) – Maximum range values.
- **num** (*int*) – Number of samples to generate. Default is 50. Must be non-negative.
- **base** (*float*) – If 1, will use `np.exp()`, if 2 will use `np.exp2()`.

Returns *array* – An array of integer values spread by the exponential function.

Examples

```
>>> import neurokit2 as nk
>>> nk.expspace(start=4, stop=100, num=10)
array([ 4,  6,  8, 12, 17, 24, 34, 49, 70, 100])
```

find_closest (*closest_to*, *list_to_search_in*, *direction*='both', *strictly*=False, *return_index*=False)

Find the closest number in the array from a given number x.

Parameters

- **closest_to** (*float*) – The target number(s) to find the closest of.
- **list_to_search_in** (*list*) – The list of values to look in.
- **direction** (*str*) – “both” for smaller or greater, “greater” for only greater numbers and “smaller” for the closest smaller.
- **strictly** (*bool*) – False for strictly superior or inferior or True for including equal.
- **return_index** (*bool*) – If True, will return the index of the closest value in the list.

Returns **closest** (*int*) – The closest number in the array.

Example

```
>>> import neurokit2 as nk
>>>
>>> # Single number
>>> x = nk.find_closest(1.8, [3, 5, 6, 1, 2])
>>> x
>>>
>>> y = nk.find_closest(1.8, [3, 5, 6, 1, 2], return_index=True)
>>> y
>>>
>>> # Vectorized version
>>> x = nk.find_closest([1.8, 3.6], [3, 5, 6, 1, 2])
>>> x
```

find_consecutive (*x*)

Find and group consecutive values in a list.

Creates a list of integer values of a given length from start to stop, spread by an exponential function.

Parameters **x** (*list*) – The list to look in.

Returns *list* – A list of tuples corresponding to groups containing all the consecutive numbers.

Examples

```
>>> import neurokit2 as nk
>>>
>>> x = [2, 3, 4, 5, 12, 13, 14, 15, 16, 17, 20]
>>> nk.find_consecutive(x)
[(2, 3, 4, 5), (12, 13, 14, 15, 16, 17), (20,)]
```

listify (***kwargs*)

Transforms arguments into lists of the same length.

Examples

```
>>> import neurokit2 as nk
>>>
>>> nk.listify(a=3, b=[3, 5], c=[3])
{'a': [3, 3], 'b': [3, 5], 'c': [3, 3]}
```

BENCHMARKS

Contents:

8.1 Benchmarking of ECG Preprocessing Methods

.

We'd like to publish this study, but unfortunately we currently don't have the time. If you want to help to make it happen, please contact us!

8.1.1 Introduction

This work is a replication and extension of the work by Porr & Howell (2019), that compared the performance of different popular R-peak detectors.

8.1.2 Databases

Glasgow University Database

The GUDB Database (Howell & Porr, 2018) contains ECGs from 25 subjects. Each subject was recorded performing 5 different tasks for two minutes (sitting, doing a maths test on a tablet, walking on a treadmill, running on a treadmill, using a hand bike). The sampling rate is 250Hz for all the conditions.

The script to download and format the database using the Python package by Bernd Porr can be found .

MIT-BIH Arrhythmia Database

The MIT-BIH Arrhythmia Database (MIT-Arrhythmia; Moody & Mark, 2001) contains 48 excerpts of 30-min of two-channel ambulatory ECG recordings sampled at 360Hz and 25 additional recordings from the same participants including common but clinically significant arrhythmias (denoted as the MIT-Arrhythmia-x database).

The script to download and format the database using the can be found .

MIT-BIH Normal Sinus Rhythm Database

This database includes 18 clean long-term ECG recordings of subjects. Due to memory limits, we only kept the second hour of recording of each participant.

The script to download and format the database using the can be found .

Concatenate them together

```
import pandas as pd

# Load ECGs
ecgs_gudb = pd.read_csv("../data/gudb/ECGs.csv")
ecgs_mit1 = pd.read_csv("../data/mit_arrhythmia/ECGs.csv")
ecgs_mit2 = pd.read_csv("../data/mit_normal/ECGs.csv")

# Load True R-peaks location
rpeaks_gudb = pd.read_csv("../data/gudb/Rpeaks.csv")
rpeaks_mit1 = pd.read_csv("../data/mit_arrhythmia/Rpeaks.csv")
rpeaks_mit2 = pd.read_csv("../data/mit_normal/Rpeaks.csv")

# Concatenate
ecgs = pd.concat([ecgs_gudb, ecgs_mit1, ecgs_mit2], ignore_index=True)
rpeaks = pd.concat([rpeaks_gudb, rpeaks_mit1, rpeaks_mit2], ignore_index=True)
```

8.1.3 Study 1: Comparing Different R-Peaks Detection Algorithms

Procedure

Setup Functions

```
import neurokit2 as nk

def neurokit(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="neurokit")
    return info["ECG_R_Peaks"]

def pantompkins1985(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method=
    ↪ "pantompkins1985")
    return info["ECG_R_Peaks"]

def hamilton2002(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="hamilton2002
    ↪ ")
    return info["ECG_R_Peaks"]

def martinez2003(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="martinez2003
    ↪ ")
    return info["ECG_R_Peaks"]

def christov2004(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="christov2004
    ↪ ")
```

(continues on next page)

(continued from previous page)

```

    return info["ECG_R_Peaks"]

def gamboa2008(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="gamboa2008")
    return info["ECG_R_Peaks"]

def elgendi2010(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="elgendi2010")
    return info["ECG_R_Peaks"]

def engzeemod2012(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="engzeemod2012")
    return info["ECG_R_Peaks"]

def kalidas2017(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="kalidas2017")
    return info["ECG_R_Peaks"]

def rodrigues2020(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="rodrigues2020")
    return info["ECG_R_Peaks"]

```

Run the Benchmarking

Note: This takes a long time (several hours).

```

results = []
for method in [neurokit, pantompkins1985, hamilton2002, martinez2003, christov2004,
               gamboa2008, elgendi2010, engzeemod2012, kalidas2017, rodrigues2020]:
    result = nk.benchmark_ecg_preprocessing(method, ecgs, rpeaks)
    result["Method"] = method.__name__
    results.append(result)
results = pd.concat(results).reset_index(drop=True)
results.to_csv("data_detectors.csv", index=False)

```

Results

```

library(tidyverse)
library(easystats)
library(lme4)

data <- read.csv("data_detectors.csv", stringsAsFactors = FALSE) %>%
  mutate(Database = ifelse(str_detect(Database, "GUDB"), paste0(str_replace(Database,
    "GUDB_", "GUDB ("), ")"), Database),
    Method = fct_relevel(Method, "neurokit", "pantompkins1985", "hamilton2002",
    "martinez2003", "christov2004", "gamboa2008", "elgendi2010", "engzeemod2012",
    "kalidas2017", "rodrigues2020"),
    Participant = paste0(Database, Participant))

```

(continues on next page)

(continued from previous page)

```

colors <- c("neurokit"="#E91E63", "pantompkins1985"="#f44336", "hamilton2002"="#FF5722",
  "martinez2003"="#FF9800", "christov2004"="#FFC107", "gamboa2008"="#4CAF50",
  "elgendi2010"="#009688", "engzeemod2012"="#2196F3", "kalidas2017"="#3F51B5",
  "rodrigues2020"="#9C27B0")

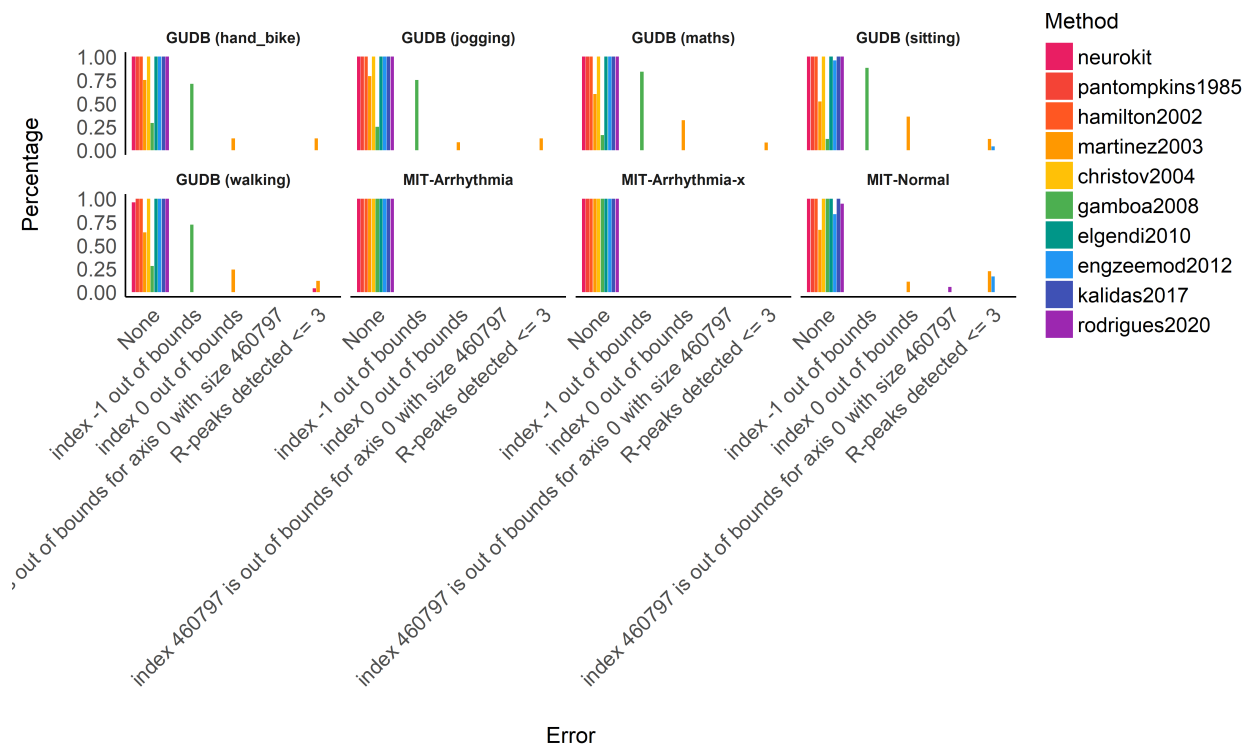
```

Errors and bugs

```

data %>%
  mutate(Error = case_when(
    Error == "index -1 is out of bounds for axis 0 with size 0" ~ "index -1 out of _
    bounds",
    Error == "index 0 is out of bounds for axis 0 with size 0" ~ "index 0 out of _
    bounds",
    TRUE ~ Error)) %>%
  group_by(Database, Method) %>%
  mutate(n = n()) %>%
  group_by(Database, Method, Error) %>%
  summarise(Percentage = n() / unique(n)) %>%
  ungroup() %>%
  mutate(Error = fct_relevel(Error, "None")) %>%
  ggplot(aes(x=Error, y=Percentage, fill=Method)) +
    geom_bar(stat="identity", position = position_dodge2(preserve = "single")) +
    facet_wrap(~Database, nrow=2) +
    theme_modern() +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    scale_fill_manual(values=colors)

```



Conclusion: It seems that gamboa2008 and martinez2003 are particularly prone to errors, especially in the case

of a noisy ECG signal. Aside from that, the other algorithms are quite resistant and bug-free.

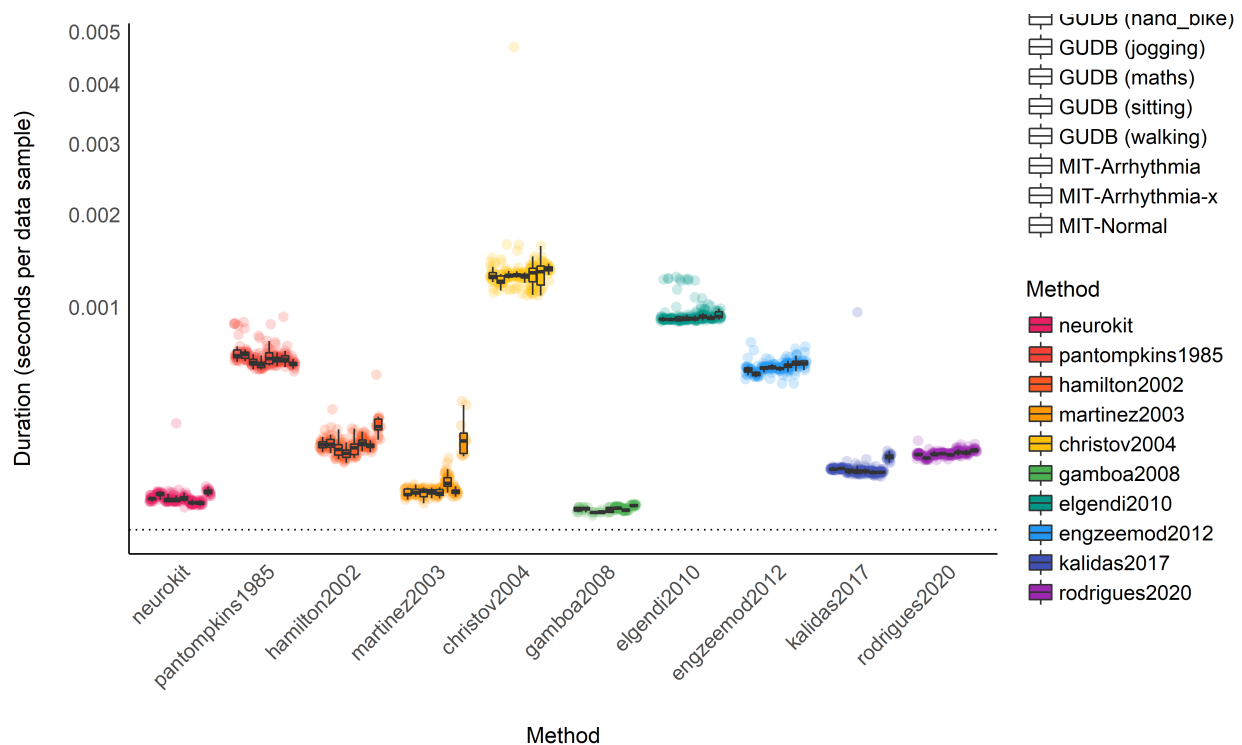
```
data <- filter(data, Error == "None")
data <- filter(data, !is.na(Score))
```

Computation Time

Descriptive Statistics

```
# Normalize duration
data <- data %>%
  mutate(Duration = (Duration) / (Recording_Length * Sampling_Rate))

data %>%
  ggplot(aes(x=Method, y=Duration, fill=Method)) +
  geom_jitter2(aes(color=Method, group=Database), size=3, alpha=0.2,
    position=position_jitterdodge()) +
  geom_boxplot(aes(alpha=Database), outlier.alpha = 0) +
  geom_hline(yintercept=0, linetype="dotted") +
  theme_modern() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_alpha_manual(values=seq(0, 1, length.out=8)) +
  scale_color_manual(values=colors) +
  scale_fill_manual(values=colors) +
  scale_y_sqrt() +
  ylab("Duration (seconds per data sample)")
```



Statistical Modelling

```

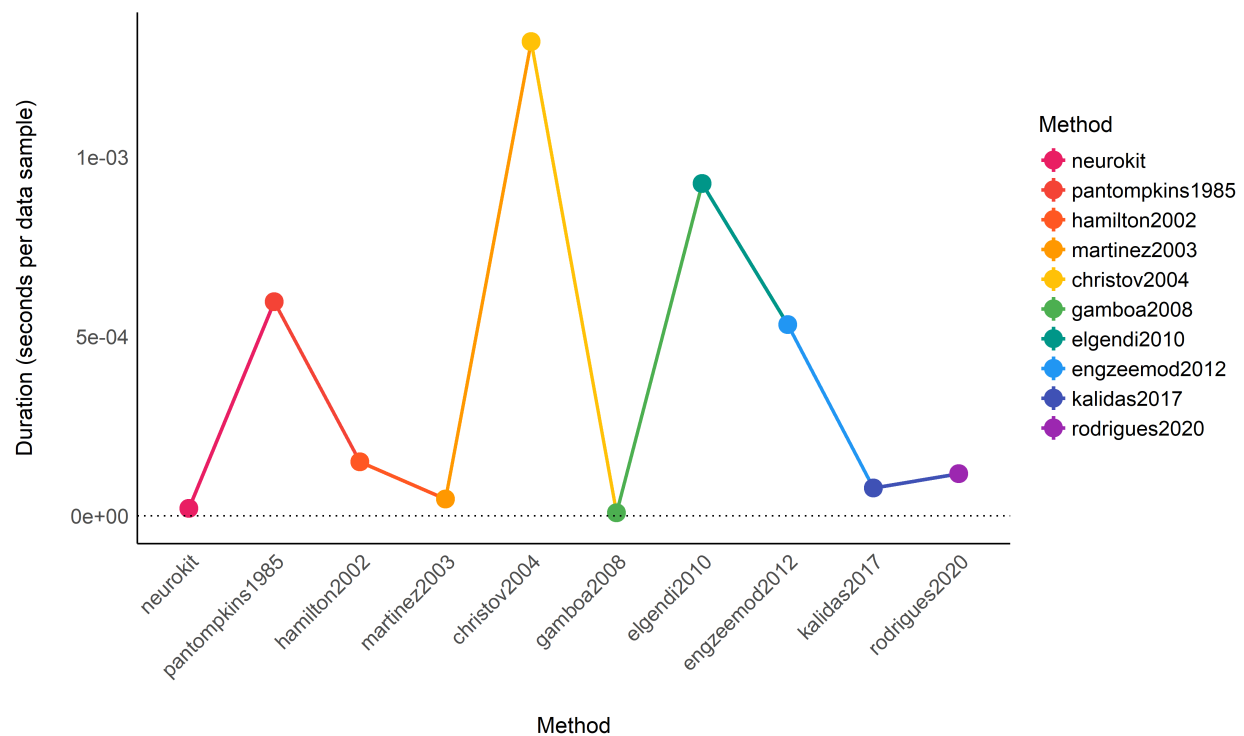
model <- lmer(Duration ~ Method + (1|Database) + (1|Participant), data=data)

means <- modelbased::estimate_means(model)

arrange(means, Mean)
##           Method      Mean      SE    CI_low  CI_high
## 1      gamboa2008 8.47e-06 9.34e-06 -9.98e-06 2.69e-05
## 2      neurokit  2.08e-05 7.20e-06  6.43e-06 3.52e-05
## 3    martinez2003 4.68e-05 8.00e-06  3.09e-05 6.27e-05
## 4    kalidas2017 7.77e-05 7.19e-06  6.33e-05 9.20e-05
## 5   rodriguez2020 1.17e-04 7.20e-06  1.03e-04 1.32e-04
## 6   hamilton2002 1.51e-04 7.19e-06  1.36e-04 1.65e-04
## 7   engzeemod2012 5.33e-04 7.24e-06  5.19e-04 5.47e-04
## 8 pantompkins1985 5.96e-04 7.19e-06  5.82e-04 6.11e-04
## 9    elgendi2010 9.26e-04 7.19e-06  9.12e-04 9.41e-04
## 10 christov2004 1.32e-03 7.19e-06  1.31e-03 1.34e-03

means %>%
  ggplot(aes(x=Method, y=Mean, color=Method)) +
  geom_line(aes(group=1), size=1) +
  geom_pointrange(aes(ymin=CI_low, ymax=CI_high), size=1) +
  geom_hline(yintercept=0, linetype="dotted") +
  theme_modern() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_color_manual(values=colors) +
  ylab("Duration (seconds per data sample)")

```



Conclusion: It seems that gamboa2008 and neurokit are the fastest methods, followed by martinez2003, kalidas2017, rodriguez2020 and hamilton2002. The other methods are then substantially slower.

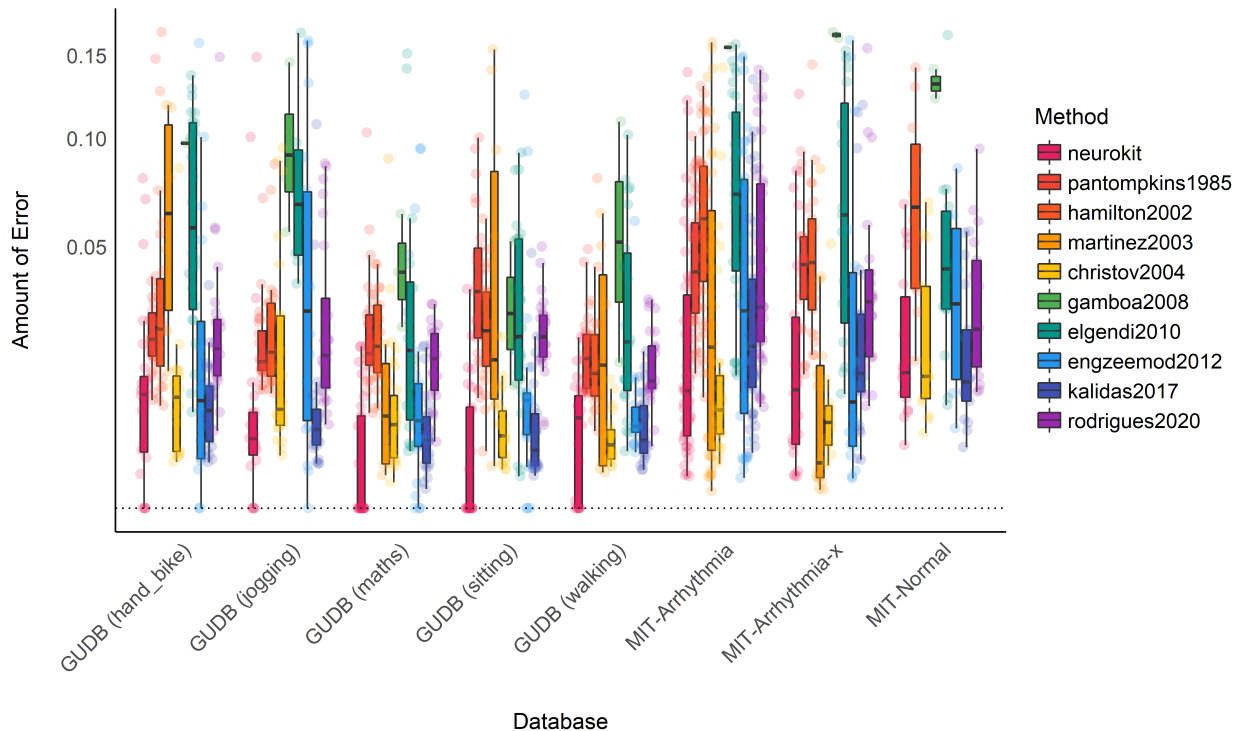
Accuracy

Note: The accuracy is computed as the absolute distance from the original “true” R-peaks location. As such, the closest to zero, the better the accuracy.

Descriptive Statistics

```
data <- data %>%
  mutate(Outlier = performance::check_outliers(Score, threshold = list(zscore = stats:
    ~:qnorm(p = 1 - 0.000001)))) %>%
  filter(Outlier == 0)

data %>%
  ggplot(aes(x=Database, y=Score)) +
    geom_boxplot(aes(fill=Method), outlier.alpha = 0, alpha=1) +
    geom_jitter2(aes(color=Method, group=Method), size=3, alpha=0.2,
    ~:position=position_jitterdodge()) +
    geom_hline(yintercept=0, linetype="dotted") +
    theme_modern() +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    scale_color_manual(values=colors) +
    scale_fill_manual(values=colors) +
    scale_y_sqrt() +
    ylab("Amount of Error")
```



Statistical Modelling

```

model <- lmer(Score ~ Method + (1|Database) + (1|Participant), data=data)

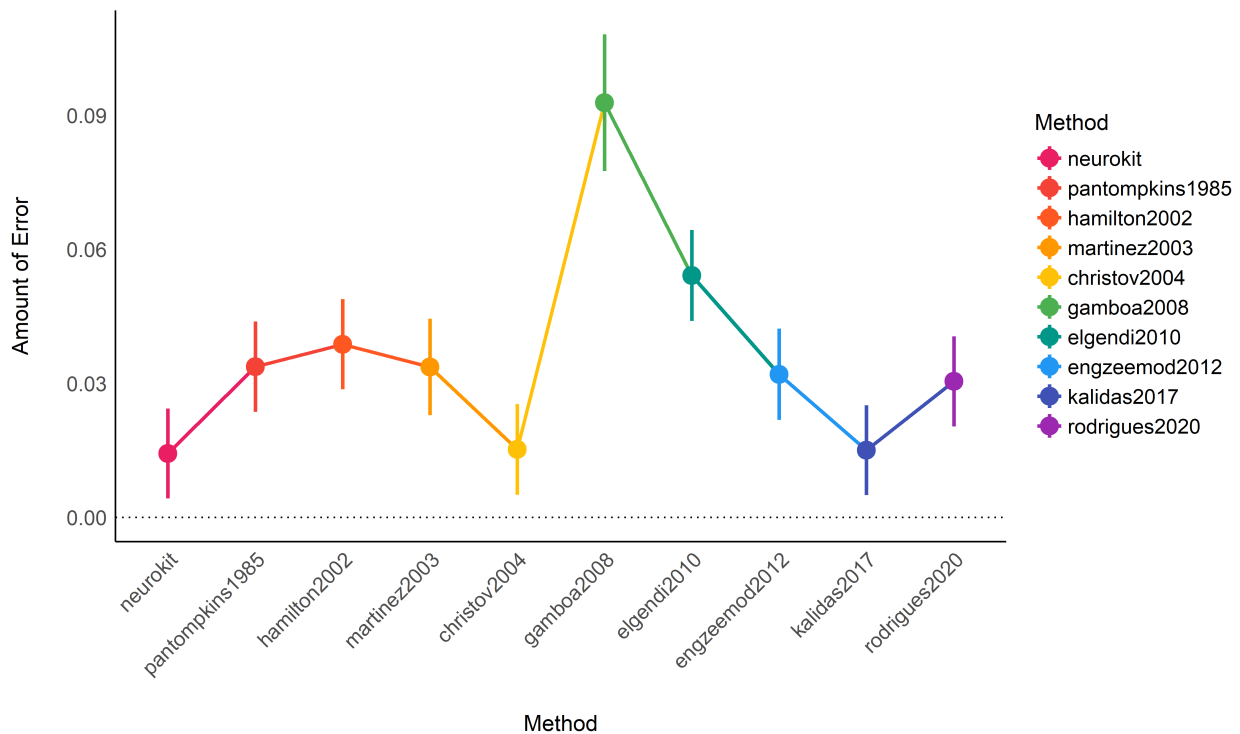
means <- modelbased::estimate_means(model)

arrange(means, abs(Mean))

##           Method      Mean      SE  CI_low CI_high
## 1      neurokit  0.0143 0.00448 0.00424  0.0244
## 2    kalidas2017  0.0151 0.00448 0.00500  0.0251
## 3 christov2004  0.0152 0.00455 0.00507  0.0253
## 4  rodriguez2020  0.0304 0.00449 0.02037  0.0405
## 5  engzeemod2012  0.0320 0.00462 0.02180  0.0422
## 6  martinez2003  0.0337 0.00507 0.02290  0.0445
## 7 pantompkins1985 0.0338 0.00453 0.02364  0.0439
## 8    hamilton2002 0.0387 0.00450 0.02865  0.0488
## 9    elgendi2010 0.0541 0.00459 0.04397  0.0643
## 10  gamboa2008  0.0928 0.00768 0.07750  0.1081

means %>%
  ggplot(aes(x=Method, y=Mean, color=Method)) +
  geom_line(aes(group=1), size=1) +
  geom_pointrange(aes(ymin=CI_low, ymax=CI_high), size=1) +
  geom_hline(yintercept=0, linetype="dotted") +
  theme_modern() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_color_manual(values=colors) +
  ylab("Amount of Error")

```



Conclusion: It seems that neurokit, kalidas2017 and christov2004 the most accurate algorithms to detect R-peaks. This pattern of results differs a bit from Porr & Howell (2019) that outlines engzeemod2012, elgendi2010, kalidas2017 as the most accurate and christov2004, hamilton2002

and pantompkins1985 as the worse. Discrepancies could be due to the differences in data and analysis, as here we used more databases and modelled them by respecting their hierarchical structure using mixed models.

Conclusion

Based on the accuracy / execution time criterion, it seems like neurokit is the best R-peak detection method, followed by kalidas2017.

8.1.4 Study 2: Normalization

Procedure

Setup Functions

```
import neurokit2 as nk

def none(ecg, sampling_rate):
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="neurokit")
    return info["ECG_R_Peaks"]

def mean_detrend(ecg, sampling_rate):
    ecg = nk.signal_detrend(ecg, order=0)
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="neurokit")
    return info["ECG_R_Peaks"]

def standardize(ecg, sampling_rate):
    ecg = nk.standardize(ecg)
    signal, info = nk.ecg_peaks(ecg, sampling_rate=sampling_rate, method="neurokit")
    return info["ECG_R_Peaks"]
```

Run the Benchmarking

Note: This takes a long time (several hours).

```
results = []
for method in [none, mean_detrend, standardize]:
    result = nk.benchmark_ecg_preprocessing(method, ecgs, rpeaks)
    result["Method"] = method.__name__
    results.append(result)
results = pd.concat(results).reset_index(drop=True)

results.to_csv("data_normalization.csv", index=False)
```

Results

```
library(tidyverse)
library(easystats)
library(lme4)

data <- read.csv("data_normalization.csv", stringsAsFactors = FALSE) %>%
  mutate(Database = ifelse(str_detect(Database, "GUDB"), paste0(str_replace(Database,
  ↪ "GUDB_", "GUDB (", ")"), Database),
    Method = fct_relevel(Method, "none", "mean_removal", "standardization"),
    Participant = paste0(Database, Participant)) %>%
  filter(Error == "None") %>%
  filter(!is.na(Score))

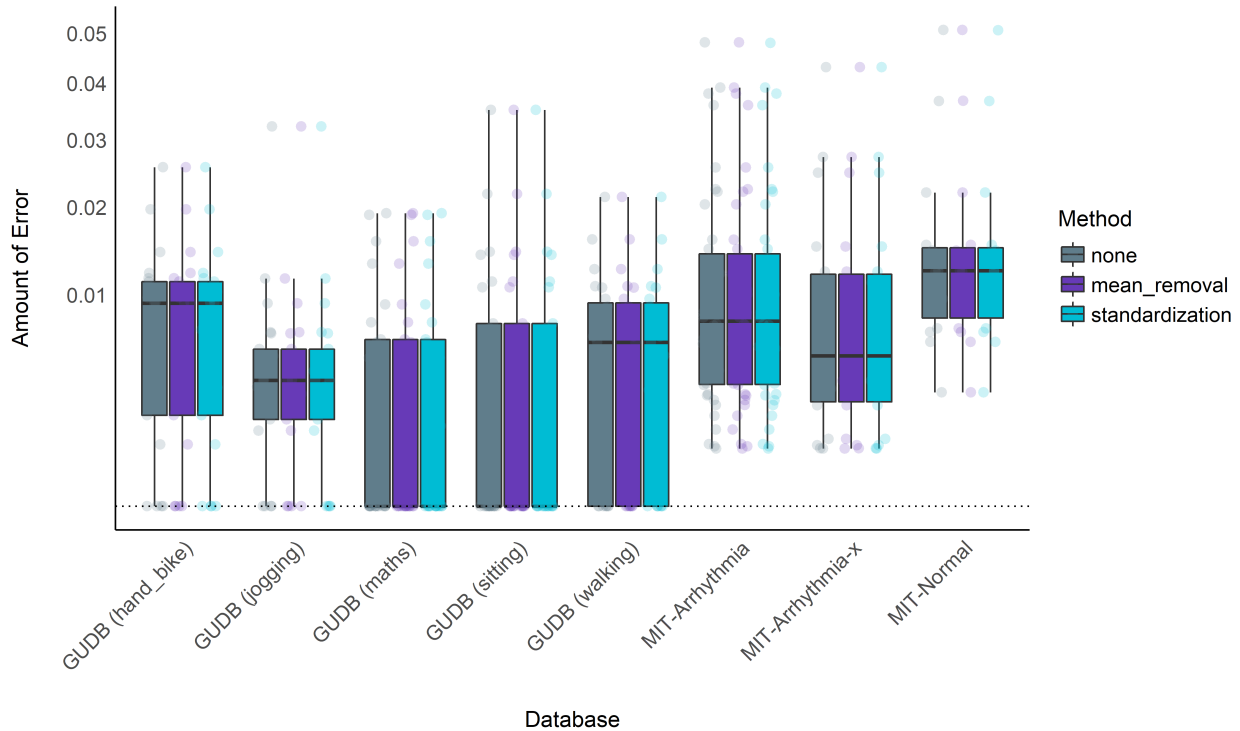
colors <- c("none"="#607D8B", "mean_removal"="#673AB7", "standardization"="#00BCD4")
```

Accuracy

Descriptive Statistics

```
data <- data %>%
  mutate(Outlier = performance::check_outliers(Score, threshold = list(zscore = stats:
  ↪ :qnorm(p = 1 - 0.000001)))) %>%
  filter(Outlier == 0)

data %>%
  ggplot(aes(x=Database, y=Score)) +
    geom_boxplot(aes(fill=Method), outlier.alpha = 0, alpha=1) +
    geom_jitter2(aes(color=Method, group=Method), size=3, alpha=0.2,
  ↪ position=position_jitterdodge()) +
    geom_hline(yintercept=0, linetype="dotted") +
    theme_modern() +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    scale_color_manual(values=colors) +
    scale_fill_manual(values=colors) +
    scale_y_sqrt() +
    ylab("Amount of Error")
```

Statistical Modelling

```
model <- lmer(Score ~ Method + (1|Database) + (1|Participant), data=data)

modelbased::estimate_contrasts(model)
## Level1 | Level2 | Difference | SE | 95% CI |
## -----
## t | df | p | Difference (std.)
## -----
## mean_removal | none | 1.59e-07 | 5.20e-07 | [-0.00e+00, 0.00e+00] | 0.
## 31 | 370.00 | 0.759 | 1.64e-05
## mean_removal | standardization | 7.75e-07 | 5.20e-07 | [-0.00e+00, 0.00e+00] | 1.
## 49 | 370.00 | 0.410 | 7.99e-05
## none | standardization | 6.15e-07 | 5.20e-07 | [-0.00e+00, 0.00e+00] | 1.
## 18 | 370.00 | 0.474 | 6.34e-05

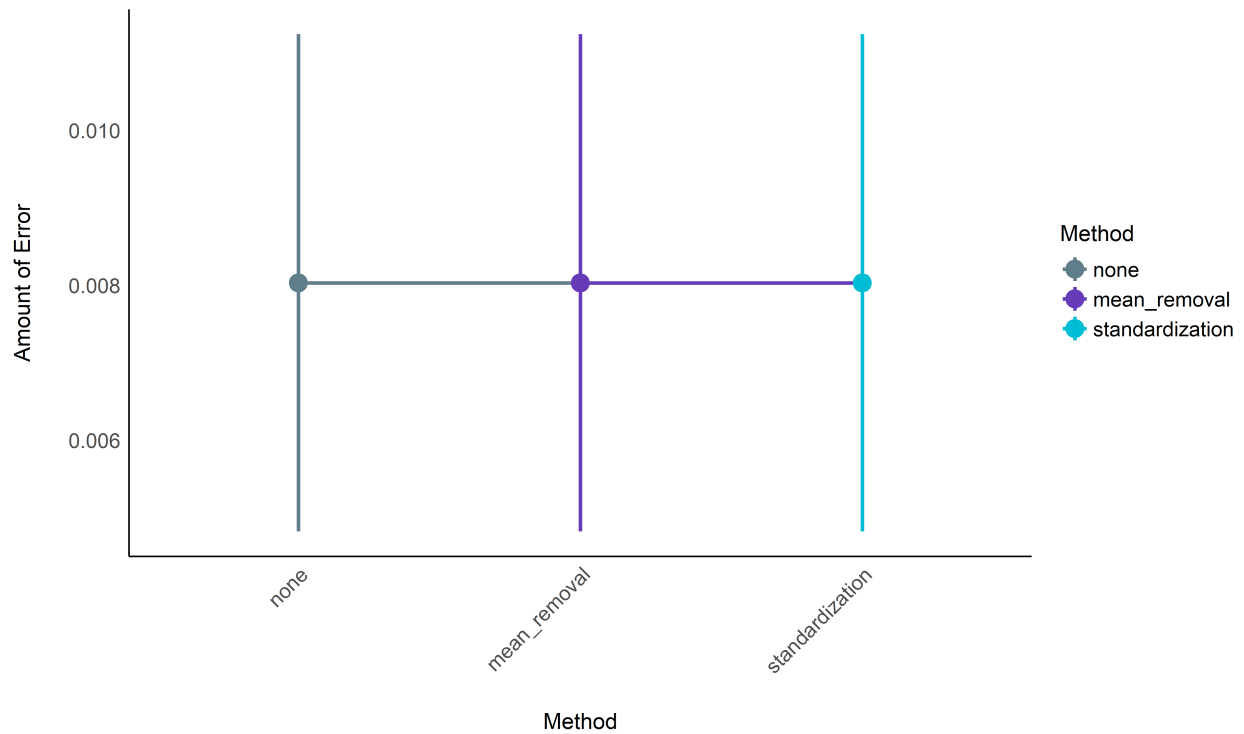
means <- modelbased::estimate_means(model)
arrange(means, abs(Mean))
## Method Mean SE CI_low CI_high
## 1 standardization 0.00803 0.00136 0.00482 0.0112
## 2 none 0.00803 0.00136 0.00482 0.0112
## 3 mean_removal 0.00803 0.00136 0.00482 0.0112

means %>%
  ggplot(aes(x=Method, y=Mean, color=Method)) +
  geom_line(aes(group=1), size=1) +
  geom_pointrange(aes(ymin=CI_low, ymax=CI_high), size=1) +
  theme_modern() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  scale_color_manual(values=colors) +
```

(continues on next page)

(continued from previous page)

```
ylab("Amount of Error")
```



Conclusion

No significant benefits added by normalization for the `neurokit` method.

8.2 References

DATASETS

NeuroKit includes datasets that can be used for testing. These datasets are not downloaded automatically with the package (to avoid increasing its weight), but can be downloaded via the `nk.data()` function.

9.1 ECG (1000 hz)

Type	Frequency	Signals
Single-subject	1000 Hz	ECG

```
data = nk.data(dataset="ecg_1000hz")
```

9.2 ECG - pandas (3000 hz)

Type	Frequency	Signals
Single-subject	3000 Hz	ECG

```
data = nk.data(dataset="ecg_3000_pandas")
```

9.3 Event-related (4 events)

Type	Frequency	Signals
Single-subject with events	100 Hz	ECG, EDA, RSP, Photosensor

```
data = nk.data(dataset="bio_eventrelated_100hz")
```

- Used in the following docstrings:
 - `bio_analyze()`
 - `ecg_analyze()`
 - `ecg_eventrelated()`
 - `ecg_rsa()`

- `ecg_rsp()`
- `eda_analyze()`
- `eda_eventrelated()`
- `eda_phasic()`
- `epochs_create()`
- `epochs_plot()`
- `epochs_to_df()`
- `rsp_analyze()`
- `rsp_eventrelated()`
- `signal_power()`
- Used in the following examples:
 - Event-related Analysis
 - Analyze Respiratory Rate Variability (RRV)

9.4 Resting state (5 min)

Type	Frequency	Signals
Single-subject resting state	100 Hz	ECG, PPG, RSP

```
data = nk.data(dataset="bio_resting_5min_100hz")
```

- Used in the following docstrings:
 - `bio_analyze()`
 - `ecg_analyze()`
 - `ecg_intervalrelated()`
 - `rsp_analyze()`
 - `rsp_intervalrelated()`
- Used in the following examples:
 - Interval-related Analysis

9.5 Resting state (8 min)

Type	Frequency	Signals
Single-subject resting state	100 Hz	ECG, RSP, EDA, Photosensor

```
data = nk.data(dataset="bio_resting_8min_100hz")
```

- Used in the following docstrings:

- `eda_analyze()`
- `eda_intervalrelated()`

CONTRIBUTING

We're glad that you are considering **joining the team** and the **community of open-science**. You can find step-by-step guides below that will show you how to make a perfect first contribution.

All people are very much welcome to contribute to code, documentation, testing and suggestions.

This package aims at being beginner-friendly. Even if you're new to this open-source way of life, new to coding and GitHub stuff, we encourage you to try submitting pull requests (PRs).

- *"I'd like to help, but I'm not good enough with programming yet"*

It's alright, don't worry! You can always dig in the code, in the documentation or tests. There are always some typos to fix, some docs to improve, some details to add, some code lines to document, some tests to add... Just explore the code structure, find where functions are located, where documentation is written, where tests are made, and see what you can fix. **Even the smaller PRs are appreciated.**

- *"I don't know how to code at all :("*

You can still contribute to the [documentation](#) by creating tutorials, help and info!

- *"I'd like to help, but I don't know where to start"*

You can look around the [issue section](#) to find some features / ideas / bugs to start working on. You can also open a new issue **just to say that you're there, interested in helping out**. We might have some ideas adapted to your skills.

- *"I'm not sure if my suggestion or idea is worthwhile"*

Enough with the impostor syndrome! All suggestions and opinions are good, and even if it's just a thought or so, it's always good to receive feedback.

- *"Why should I waste my time with this? Do I get any credit?"*

Software contributions are getting more and more valued in the academic world, so it is a good time to collaborate with us! All contributors will be added within the [authors list](#). We're also very keen on including them to eventual academic publications.

Anyway, starting is the most important! You will then enter a *whole new world, a new fantastic point of view*... So fork this repo, do some changes and submit them. We will then work together to make the best out of it

Contributing Guides:

10.1 Understanding NeuroKit

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing [see our guides](#)).

Let's start by reviewing some basic coding principles that might help you get familiar with NeuroKit

If you are reading this, it could be because you don't feel comfortable enough with Python and NeuroKit (*yet*), and you impatiently want to get to know it in order to start looking at your data.

“Tous les chemins mènent à Rome” (*all roads lead to Rome*)

Let me start by saying that there are multiple ways you'll be able to access the documentation in order to get to know different functions, follow examples and other tutorials. So keep in mind that you will eventually find your own workflow, and that these tricks are shared simply to help you get to know your options.

10.1.1 1. readthedocs

You probably already saw the [README](#) file that shows up on NeuroKit's Github home page (right after the list of directories). It contains a brief overview of the project, some examples and figures. *But, most importantly, there are the links that will take you to the Documentation.*

Documentation basically means code explanations, references and examples.

In the Documentation section of the README, you'll find links to the [readthedocs website](#) like this one :

Hint: Did you know that you can access the documentation website using the `rtfd` domain name `https://neurokit2.rtfd.io/`, which stands for **READ THE F**** DOCS**

And a link to the [API \(or Application Program Interface\)](#), containing the list of functions) like this one:

All the info you will see on that webpage is rendered directly from the code, meaning that the website reads the code and generates a HTML page from it. **That's why it's important to structure your code in a standard manner** (You can learn how to contribute [here](#)).

The API is organized by types of signals. You'll find that each function has a **description**, and that most of them refer to peer-reviewed papers or other GitHub repositories. Also, for each function, **parameters** are described in order. Some of them will take many different **options** and all of them should be described as well.

If the options are not explained, they should be.

It's not your fault you don't understand. That's why we need you to contribute.

Example

In the **ECG section**, the `ecg_findpeaks` function takes **4 parameters**. One of them is **method**: each method refers to a peer-reviewed paper that published a peak detection algorithm. You can also see what the function **returns** and what **type of data** has been returned (integers and floating point numbers, strings, etc). Additionally, you can find **related functions** in the **See also** part. An small **example** of the function should also be found. You can copy paste it in your Python kernel, or in a Jupyter Notebook, to see what it does.

10.1.2 2. The code on Github

Now that you're familiar with *readthedocs* website, let's go back to the [repo](#). What you have to keep in mind is that *everything you saw in the previous section is in the Github repository*. The website pages, the lines that you are currently reading, are stored in the repository, which is then automatically uploaded to the website. Everything is cross-referenced, everything relates to the core which can be found in the repo. If you got here, you probably already know that a repository is like a *tree containing different branches* or directories that eventually lead you to a **script**, in which you can find a **function**.

Example

Ready for inception ? let's find the location of the file you're currently reading. Go under `docs` and find it by yourself... it should be straight-forward.

Hint: As you can see, there are several sections (see the Table of Content on the left), and we are in the **tutorials** section. So you might want to look into the **tutorials** folder :)

See! It's super handy because you can visit the scripts without downloading it. Github also renders Jupyter Notebook quite well, so you can not only see the script, but also figures and markdown sections where the coder discusses results.

10.1.3 3. The code on YOUR machine

Now, you're probably telling yourself :

If I want to use these functions, they should be somewhere on my computer!

For that, I encourage you to visit the [installation page](#) if you didn't already. Once Python is installed, its default pathway should be :

Python directory

Windows

- `C:\Users\<username>\anaconda3\`

(if the directory doesn't match, just search for the folder name `anaconda3` or `miniconda3`).

Mac

- `/Users/<username>/anaconda3`

Or, if you're using [WinPython](#) it should be in the folder of its installation (e.g., `C:\Users\<username>\Desktop\WPy-3710\`).

Linux users should know that already

Environment and NeuroKit directory

NeuroKit, along with all the other packages, are located in the `python` directory in the `site-package` folder (itself in the `Lib` folder). It should be located under the environment where you installed it (*if you didn't do it already, set a computing environment. Otherwise, you can run into problems when running your code*). The directory should look like this:

- `C:\Users\<username>\anaconda3\envs\<yourenv>\lib\site-package\neurokit2`

Or, if you're using [WinPython](#):

- `C:\Users\<username>\Desktop\WPy-3710\python-3.7.1.amd64\Lib\site-package\neurokit2`

Example

Take the ECG again :

From the specified directory, I can note that the different folders are arranged in the same way as in the [readthedocs](#) website.

Let's say I want to go back to the same function `ecg_findpeaks()`: I'd click on `ecg` folder, and from there I can see the source code for the function under `ecg_findpeaks.py`.

10.2 Contributing guide

NeuroKit2 welcomes everyone to contribute to code, documentation, testing and suggestions.

This package aims at being beginner-friendly. And if you're not yet familiar with how contribution can be done to open-source packages, or with **how to use GitHub**, this guide is for you!

Let's dive right into it!

10.2.1 NeuroKit's style

Structure and code

- The NeuroKit package is organized into submodules, such as `ecg`, `signal`, `statistics`, etc. New functions should be created within at the appropriate places.
- The API (the functions) should be consistent, with functions starting with a prefix (`plot_`, `ecg_`, `eda_`, etc.) so that the user can easily find them by typing the “intuitive” prefix.
- Authors of code contribution are invited to follow the [PEP 8](#) style sheet to write some nice (and readable) python.
- That being said, *human readability* should always be favoured over anything else. Ideally, we would like the code in NeuroKit to be understandable even by non-programmers.

- Contrary to Python recommendations, we prefer some nicely nested loops, rather than complex one-liners [*“that” for s if h in i for t in range(“don’t”) if “understand” is False*].
- The maximum **line length** is 100 characters
- Please *document and comment* your code, so that the purpose of each step (or code line) is stated in a clear and understandable way.
- Don’t forget to add tests and documentation (a description, examples, etc.) to your functions.

Run code checks

Once you’re satisfied by the code you’ve written, you will need to run some checks to make sure it is “standardized”. You will need to open the command line and install the following packages:

```
pip install isort black docformatter flake8 pylint
```

Now, navigate to the folder where your script is by typing `cd C:\the\folder\of\my\file`. Once you there, you can run the following commands:

```
isort myfile.py -l 120 --balanced --multi-line 3 --lines-between-types 1 --lines-
  ↳after-imports 2 --trailing-comma
black myfile.py --line-length 120
docformatter myfile.py --wrap-summaries 120 --wrap-descriptions 113 --blank --in-place

flake8 myfile.py --max-line-length=127 --max-complexity=10 --ignore E303,C901,E203,
  ↳W503
pylint myfile.py --max-line-length=127 --load-plugins=pylint.extensions.docparams --
  ↳load-plugins=pylint.extensions.docstyle --variable-naming-style=any --argument-
  ↳naming-style=any --reports=n --suggestion-mode=y --disable=E303 --disable=R0913 --
  ↳disable=R0801 --disable=C0114 --disable=E203 --disable=E0401 --disable=W9006 --
  ↳disable=C0330 --disable=R0914 --disable=R0912 --disable=R0915 --disable=W0102 --
  ↳disable=W0511 --disable=C1801 --disable=C0111 --disable=R1705 --disable=R1720 --
  ↳disable=C0301 --disable=C0415 --disable=C0103 --disable=C0302 --disable=R1716 --
  ↳disable=W0632 --disable=E1136 --extension-pkg-whitelist=numpy
```

The first three commands will make some modifications to your code so that it is nicely formatted, while the two last will run some checks to detect any additional issues. Please try to fix them!

PS: If you want to check the whole package (i.e., all the files of the package), run:

```
isort neurokit2 -l 120 --balanced --multi-line 3 --lines-between-types 1 --lines-
  ↳after-imports 2 --trailing-comma --skip neurokit2/complexity/___init___py --recursive
black neurokit2 --line-length 120
docformatter neurokit2 --wrap-summaries 120 --wrap-descriptions 113 --blank --in-
  ↳place --recursive

flake8 neurokit2 --exclude neurokit2/___init___py --max-line-length=127 --max-
  ↳complexity=10 --ignore E303,C901,E203,W503
pylint neurokit2 --max-line-length=127 --load-plugins=pylint.extensions.docparams --
  ↳load-plugins=pylint.extensions.docstyle --variable-naming-style=any --argument-
  ↳naming-style=any --reports=n --suggestion-mode=y --disable=E303 --disable=R0913 --
  ↳disable=R0801 --disable=C0114 --disable=E203 --disable=E0401 --disable=W9006 --
  ↳disable=C0330 --disable=R0914 --disable=R0912 --disable=R0915 --disable=W0102 --
  ↳disable=W0511 --disable=C1801 --disable=C0111 --disable=R1705 --disable=R1720 --
  ↳disable=C0301 --disable=C0415 --disable=C0103 --disable=C0302 --disable=R1716 --
  ↳disable=W0632 --disable=E1136 --extension-pkg-whitelist=numpy --exit-zero
```

Avoid Semantic Errors

Most errors detected by our code checks can be easily automated with `isort`, `black`, and `docformatter`. This leaves us with the semantic errors picked up by `pylint`, the last style check, which often have to be fixed manually. Below is a list of the most common semantic errors that occur when writing code/documentation, so before you commit any changes, do make sure you have fixed these.

Documentation

- Missing function arguments in `Parameters` and `Returns`.
- In internal functions, missing `Returns` section detected only if `Parameters` is documented but is not followed by returns documentation.
- Failure to detect documentation of arguments when they are done simultaneously in one line:

```
a, b, c, discard, n, sampling_rate, x0 : int
```

will result in a `pylint` error like `a, b, c, discard, n, sampling_rate, x0" missing in parameter documentation (missing-param-doc)` so do document each argument separately.

- Argument name different from documentation

Code

- Unused arguments
- Unused variables
- Merge if arguments, for example: `if isinstance(ecg, (list, pd.Series))` rather than `if isinstance(ecg, list)` or `if isinstance(ecg, pd.Series)`

Development workflow

The NeuroKit GitHub repository has two main branches, **master** and the **dev**. The typical workflow is to work and make changes **on the dev branch**. This dev branch has a pull request (PR) opened to track individual commits (changes). And every now and then (when a sufficient number of changes have been made), the dev branch is **merged into master**, leading to an update of the version number and an upload to PyPi.

The important thing is that you should **not directly make changes on the master branch**, because *master* is usually behind *dev* (which means for instance, maybe the things you are changing on *master* have already been changed on *dev*). The *master* should be a stable, tested branch, and *dev* is the place to experiment.

This is a summary of the typical workflow for contributing using GitHub (a detailed guide is available below):

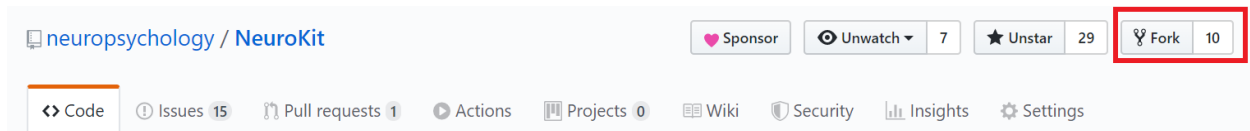
1. Download [GitHub Desktop](#) and follow the small tutorial that it proposes
2. *Fork* the NeuroKit repository (this can be done on the GitHub website page by clicking on the *Fork* button), and clone it using GitHub Desktop to your local computer (it will copy over the whole repo from GitHub to your local machine)
3. In GitHub Desktop, switch to the *dev* branch. You are now on the *dev* branch (of your own fork)
4. From there, create a new branch, called for example “bugfix-functionX” or “feature-readEEG” or “typofix”
5. Make some changes and push them (this will update **your** fork)
6. Create a pull request (PR) from your fork to the “origin” (the original repo) *dev* branch
7. This will trigger automated checks that you can explore and fix
8. Wait for it to be merged into dev, and later see it being merged into master

10.2.2 How to use GitHub to contribute

Step 1: Fork it

A *fork* is a copy of a repository. Working with the fork allows you to freely experiment with changes without affecting the original project.

Hit the **Fork** button in the top right corner of the page and in a few seconds, you will have a copy of the repository in your own GitHub account.



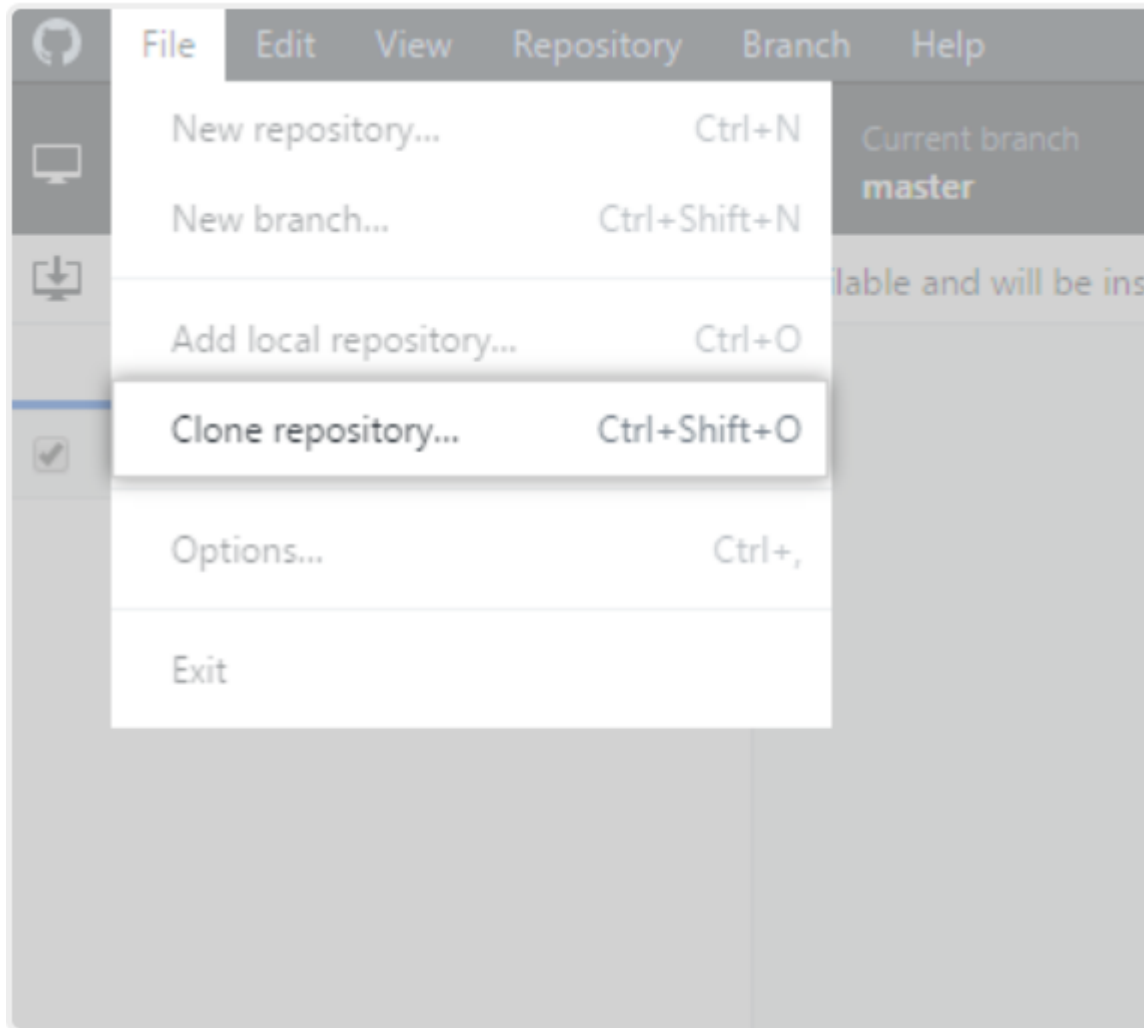
Now, that is the *remote* copy of the project. The next step is to make a *local* copy in your computer.

While you can explore Git to manage your Github developments, we recommend downloading [Github Desktop](#) instead. It makes the process way easier and more straightforward.

Step 2: Clone it

Cloning allows you to make a *local* copy of any repositories on Github.

Go to **File** menu, click **Clone Repository** and since you have forked Neurokit2, you should be able to find it easily under **Your repositories**.



Choose the local path of where you want to save your *local* copy and as simple as that, you have a working repository in your computer.

Step 3: Find it and fix it

And here is where the fun begins. You can start contributing by fixing a bug (or even a typo in the code) that has been annoying you. Or you can go to the [issue section](#) to hunt for issues that you can address.

For example, here, as I tried to run the example in *ecg_fixpeaks()* file, I ran into a bug! A typo error!

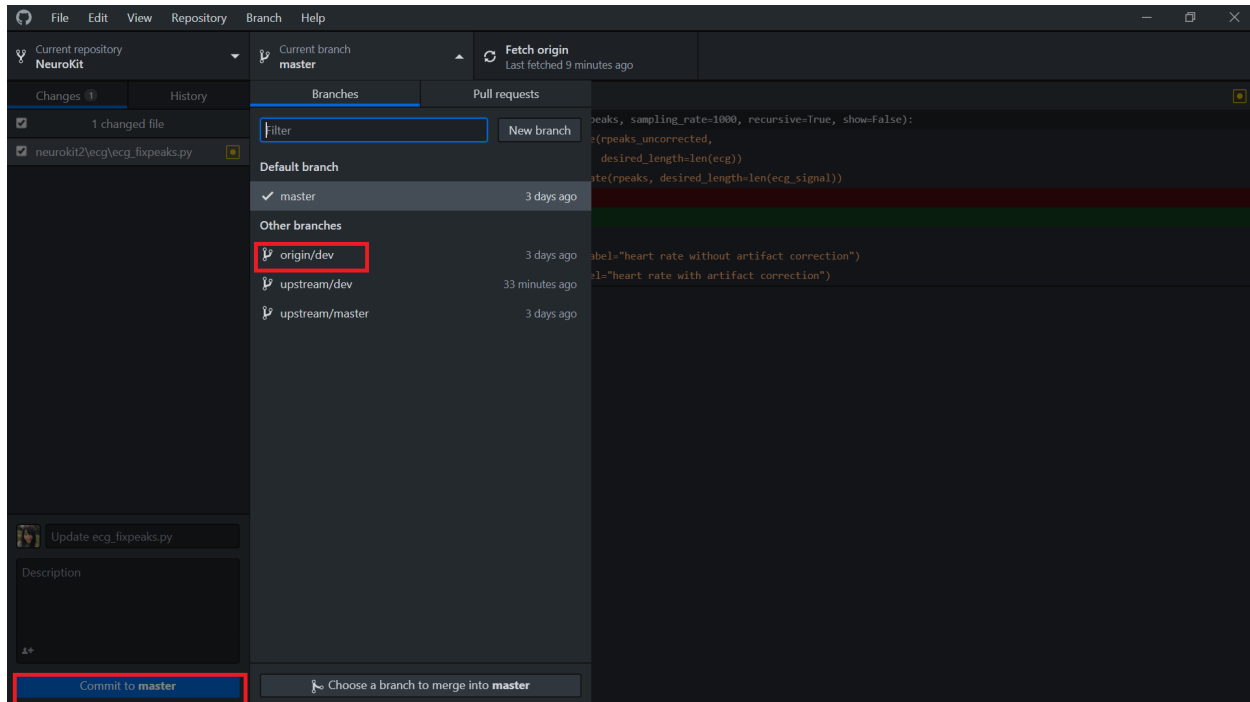
Fix it and hit the save button! That's one contribution I made to the package!

To save the changes you made (e.g. the typo that was just fixed) to your *local* copy of the repository, the next step is to *commit* it.

Step 4: Commit it and push it

In your Github Desktop, you will now find the changes that you made highlighted in **red** (removed) or **green** (added).

The first thing that you have to do is to switch from the default - *Commit to Master* to *Commit to dev*. Always commit to your dev branch as it is the branch with the latest changes. Then give the changes you made a good and succinct title and hit the *Commit* button.



Committing allows your changes to be saved in your *local* copy of the repository and in order to have the changes saved in your **remote** copy, you have to **push** the commit that you just made.

Step 4: Create pull request

The last step to make your contribution official is to create a pull request.

Tam-Pham / NeuroKit
forked from neuropsychology/NeuroKit

NeuroKit2: The Python Toolbox for Neurophysiological Signal Processing <https://neurokit2.readthedocs.io/>

Manage topics

819 commits 2 branches 0 packages 2 releases 9 contributors MIT

Branch: dev New pull request Create new file Upload files Find file Clone or download

This branch is 40 commits behind neuropsychology:dev. Pull request Compare

Commit	Message	Time
DominiqueMakowski	bio_process is in da game	Latest commit 1e1df5c 3 days ago
.github	sanitize input	17 days ago
.idea	revert	last month
data	Create example_bio_100hz.csv	28 days ago
docs	clean unused images	3 days ago
neurokit2	bio_process is in da game	3 days ago

Go to your *remote* repository on Github page, the *New Pull Request* button is located right on top of the folders. Do remember to change your branch to *dev* since your commits were pushed to the dev branch previously.

And now, all that is left is for the maintainers of the package to review your work and they can either request additional changes or merge it to the original repository.

Step 5: Let's do it

Let's do it for real! If you have a particular feature in mind that you would want to add, we would recommend first opening an *issue* to let us know, so we can eventually guide you and give you some advice. And if you don't know where to start or what to do, then read our *ideas for first contributions*. Good luck

10.2.3 Useful reads

For instance, one way of starting to contribute could be to improve this file, fix typos, clarify things, add resources links etc. :)

- [Understanding the GitHub flow](#)
- [How to create a Pull Request](#)
- [Why and How to Contribute](#)

10.2.4 What's next?

- [Ideas for first contributions](#)

10.3 Ideas for first contributions

Hint: Spotted a typo? Would like to add something or make a correction? Join us by contributing ([see our guides](#)).

Now that you're familiar with [how to use GitHub](#), you're ready to get your hands dirty and contribute to open-science? But you're not sure **where to start or what to do**? We got you covered!

In this guide, we will discuss the two best types of contributions for beginners, as they are easy to make, super useful and safe (you cannot break the package).

10.3.1 Look for “good first contribution” issues

If you know how to code a bit, you can check out the issues that have been flagged as [good for first contribution](#). This means that they are issue or features ideas that we believe are accessible to beginners. If you're interested, do not hesitate to comment on these issues to know more, have more info or ask for guidance! We'll be really happy to help in any way we can .

10.3.2 Improving documentation

One of the easiest thing is to improve, complete or fix the documentation for functions. For instance the `ecg_simulate()` function has a documentation with a general description, a description of the arguments, some example etc. As you've surely noticed, sometimes more details would be needed, some typos are present, or some references could be added.

The documentation for functions is located alongside the function *definition* (the code of the function). If you've read [understanding NeuroKit](#), you know that the code of the `ecg_simulate()` function is [here](#). And as you can see, just below the function name, there is a big *string* (starting and ending with `"""`) containing the documentation.

This thing is called the *docstring*.

If you modify it here, then it will be updated automatically on the website!

10.3.3 Adding tests

Tests are super important for programmers to make sure that the changes that we make at one location don't create unexpected changes at another place.

Adding them is a good first issue for new contributors, as it takes little time, doesn't require advanced programming skills and is a good occasion to discover functions and how they work.

By clicking on the “[coverage](#)” badge under the logo on the README page, then on the “neurokit2” folder button at the bottom, you can see the [breakdown of testing coverage](#) for each submodules (folders), and if you click on one of them, the coverage for each individual file/function ([example here](#)).

This percentage of coverage needs be improved

The common approach is to identify functions, methods or arguments that are not tested, and then try to write a small test to cover them (i.e., a small self-contained piece of code that will run through a given portion of code and which output is tested (e.g., `assert x == 3`) and depends on the correct functioning of that code), and then add this test to the appropriate [testing file](#).

For instance, let's imagine the following function:

```
def domsfunction(x, method="great"):
    if method == "great":
        z = x + 3
    else:
        z = x + 4
    return z
```

In order to test that function, I have to write some code that “runs through” it and put in a function which name starts with `test_*`, for instance:

```
def test_domsfunction():
    # Test default parameters
    output = domsfunction(1)
    assert output == 4
```

This will go through the function, which default method is “*great*”, therefore adds 3 to the input (here 1), and so the result *should* be 4. And the test makes sure that it is 4. However, we also need to add a second test to cover the other method of the function (when `method != “great”`), for instance:

```
def test_domsfunction():
    # Test default parameters
    output = domsfunction(1)
    assert output == 4

    # Test other method
    output = domsfunction(1, method="whatever")
    assert isinstance(output, int)
```

I could have written `assert output == 5`, however, I decided instead to check the type of the output (whether it is an integer). That’s the thing with testing, it requires to be creative, but also in more complex cases, to be clever about what and how to test. But it’s an interesting challenge

You can see examples of tests in the existing [test files](#).

10.3.4 Adding examples and tutorials

How to write

The documentation that is on the [website](#) is automatically built by the hosting website, readthedocs, from [reStructured Text \(RST\) files](#) (a syntax similar to markdown) or from [jupyter notebooks \(.ipynb\)](#) Notebooks are preferred if your example contains code and images.

Where to add the files

These documentation files that we need to write are located in the `/docs/` folder. For instance, if you want to add an example, you need to create a new file, for instance `myexample.rst`, in the `docs/examples/` folder.

If you want to add images to an `.rst` file, best is to put them in the `/docs/img/` folder and to reference their link.

However, in order for this file to be easily **accessible from the website**, you also need to add it to the **table of content** located in the `index` file (just add the name of the file without the extension).

Do not hesitate to ask for more info by creating an [issue](#)!

PYTHON MODULE INDEX

n

- neurokit2.bio, [234](#)
- neurokit2.complexity, [202](#)
- neurokit2.data, [190](#)
- neurokit2.ecg, [107](#)
- neurokit2.eda, [146](#)
- neurokit2.eeg, [163](#)
- neurokit2.emg, [157](#)
- neurokit2.epochs, [191](#)
- neurokit2.events, [187](#)
- neurokit2.hrv, [128](#)
- neurokit2.misc, [236](#)
- neurokit2.ppg, [124](#)
- neurokit2.rsp, [135](#)
- neurokit2.signal, [165](#)
- neurokit2.stats, [194](#)

A

`as_vector()` (in module *neurokit2.misc*), 236

B

`bio_analyze()` (in module *neurokit2.bio*), 234

`bio_process()` (in module *neurokit2.bio*), 235

C

`complexity_apen()` (in module *neurokit2.complexity*), 202

`complexity_capen()` (in module *neurokit2.complexity*), 203

`complexity_cmse()` (in module *neurokit2.complexity*), 204

`complexity_d2()` (in module *neurokit2.complexity*), 205

`complexity_delay()` (in module *neurokit2.complexity*), 206

`complexity_dfa()` (in module *neurokit2.complexity*), 208

`complexity_dimension()` (in module *neurokit2.complexity*), 209

`complexity_embedding()` (in module *neurokit2.complexity*), 210

`complexity_fuzzycmse()` (in module *neurokit2.complexity*), 211

`complexity_fuzzyen()` (in module *neurokit2.complexity*), 212

`complexity_fuzzymse()` (in module *neurokit2.complexity*), 213

`complexity_fuzzycmse()` (in module *neurokit2.complexity*), 214

`complexity_mfdfa()` (in module *neurokit2.complexity*), 216

`complexity_mse()` (in module *neurokit2.complexity*), 217

`complexity_optimize()` (in module *neurokit2.complexity*), 218

`complexity_plot()` (in module *neurokit2.complexity*), 219

`complexity_r()` (in module *neurokit2.complexity*), 220

`complexity_rcmse()` (in module *neurokit2.complexity*), 221

`complexity_sampen()` (in module *neurokit2.complexity*), 222

`complexity_se()` (in module *neurokit2.complexity*), 223

`complexity_simulate()` (in module *neurokit2.complexity*), 224

`cor()` (in module *neurokit2.stats*), 194

D

`data()` (in module *neurokit2.data*), 190

`density()` (in module *neurokit2.stats*), 194

`distance()` (in module *neurokit2.stats*), 195

E

`ecg_analyze()` (in module *neurokit2.ecg*), 107

`ecg_clean()` (in module *neurokit2.ecg*), 108

`ecg_delineate()` (in module *neurokit2.ecg*), 109

`ecg_eventrelated()` (in module *neurokit2.ecg*), 110

`ecg_findpeaks()` (in module *neurokit2.ecg*), 111

`ecg_intervalrelated()` (in module *neurokit2.ecg*), 113

`ecg_peaks()` (in module *neurokit2.ecg*), 114

`ecg_phase()` (in module *neurokit2.ecg*), 115

`ecg_plot()` (in module *neurokit2.ecg*), 116

`ecg_process()` (in module *neurokit2.ecg*), 116

`ecg_quality()` (in module *neurokit2.ecg*), 118

`ecg_rate()` (in module *neurokit2.ecg*), 118

`ecg_rsa()` (in module *neurokit2.ecg*), 119

`ecg_rsp()` (in module *neurokit2.ecg*), 121

`ecg_segment()` (in module *neurokit2.ecg*), 122

`ecg_simulate()` (in module *neurokit2.ecg*), 123

`eda_analyze()` (in module *neurokit2.eda*), 146

`eda_autocor()` (in module *neurokit2.eda*), 147

`eda_changepoints()` (in module *neurokit2.eda*), 147

`eda_clean()` (in module *neurokit2.eda*), 148

`eda_eventrelated()` (in module *neurokit2.eda*), 148

`eda_findpeaks()` (in module *neurokit2.eda*), 150

eda_fixpeaks() (in module *neurokit2.eda*), 151
 eda_intervalrelated() (in module *neurokit2.eda*), 151
 eda_peaks() (in module *neurokit2.eda*), 152
 eda_phasic() (in module *neurokit2.eda*), 153
 eda_plot() (in module *neurokit2.eda*), 154
 eda_process() (in module *neurokit2.eda*), 155
 eda_simulate() (in module *neurokit2.eda*), 156
 emg_activation() (in module *neurokit2.emg*), 157
 emg_amplitude() (in module *neurokit2.emg*), 158
 emg_analyze() (in module *neurokit2.emg*), 158
 emg_clean() (in module *neurokit2.emg*), 159
 emg_eventrelated() (in module *neurokit2.emg*), 160
 emg_intervalrelated() (in module *neurokit2.emg*), 160
 emg_plot() (in module *neurokit2.emg*), 161
 emg_process() (in module *neurokit2.emg*), 161
 emg_simulate() (in module *neurokit2.emg*), 162
 entropy_approximate() (in module *neurokit2.complexity*), 224
 entropy_fuzzy() (in module *neurokit2.complexity*), 225
 entropy_multiscale() (in module *neurokit2.complexity*), 226
 entropy_sample() (in module *neurokit2.complexity*), 227
 entropy_shannon() (in module *neurokit2.complexity*), 228
 epochs_create() (in module *neurokit2.epochs*), 191
 epochs_plot() (in module *neurokit2.epochs*), 192
 epochs_to_array() (in module *neurokit2.epochs*), 193
 epochs_to_df() (in module *neurokit2.epochs*), 193
 events_find() (in module *neurokit2.events*), 187
 events_plot() (in module *neurokit2.events*), 188
 events_to_mne() (in module *neurokit2.events*), 189
 expspace() (in module *neurokit2.misc*), 236

F

find_closest() (in module *neurokit2.misc*), 237
 find_consecutive() (in module *neurokit2.misc*), 237
 fit_error() (in module *neurokit2.stats*), 195
 fit_loess() (in module *neurokit2.stats*), 196
 fit_mixture() (in module *neurokit2.stats*), 197
 fit_mse() (in module *neurokit2.stats*), 197
 fit_polynomial() (in module *neurokit2.stats*), 197
 fit_polynomial_findorder() (in module *neurokit2.stats*), 198
 fit_r2() (in module *neurokit2.stats*), 198
 fit_rmse() (in module *neurokit2.stats*), 198
 fractal_correlation() (in module *neurokit2.complexity*), 229

fractal_dfa() (in module *neurokit2.complexity*), 230
 fractal_mandelbrot() (in module *neurokit2.complexity*), 231
 fractal_mfdfa() (in module *neurokit2.complexity*), 232

H

hdi() (in module *neurokit2.stats*), 198
 hrv() (in module *neurokit2.hrv*), 128
 hrv_frequency() (in module *neurokit2.hrv*), 129
 hrv_nonlinear() (in module *neurokit2.hrv*), 130
 hrv_time() (in module *neurokit2.hrv*), 133

L

listify() (in module *neurokit2.misc*), 237

M

mad() (in module *neurokit2.stats*), 199
 mne_channel_add() (in module *neurokit2.eeg*), 163
 mne_channel_extract() (in module *neurokit2.eeg*), 164

module

neurokit2.bio, 234
 neurokit2.complexity, 202
 neurokit2.data, 190
 neurokit2.ecg, 107
 neurokit2.eda, 146
 neurokit2.eeg, 163
 neurokit2.emg, 157
 neurokit2.epochs, 191
 neurokit2.events, 187
 neurokit2.hrv, 128
 neurokit2.misc, 236
 neurokit2.ppg, 124
 neurokit2.rsp, 135
 neurokit2.signal, 165
 neurokit2.stats, 194

mutual_information() (in module *neurokit2.stats*), 200

N

neurokit2.bio
 module, 234
 neurokit2.complexity
 module, 202
 neurokit2.data
 module, 190
 neurokit2.ecg
 module, 107
 neurokit2.eda
 module, 146
 neurokit2.eeg
 module, 163

neurokit2.emg
 module, 157
 neurokit2.epochs
 module, 191
 neurokit2.events
 module, 187
 neurokit2.hrv
 module, 128
 neurokit2.misc
 module, 236
 neurokit2.ppg
 module, 124
 neurokit2.rsp
 module, 135
 neurokit2.signal
 module, 165
 neurokit2.stats
 module, 194

P

ppg_clean() (in module neurokit2.ppg), 124
 ppg_findpeaks() (in module neurokit2.ppg), 124
 ppg_plot() (in module neurokit2.ppg), 125
 ppg_process() (in module neurokit2.ppg), 126
 ppg_rate() (in module neurokit2.ppg), 126
 ppg_simulate() (in module neurokit2.ppg), 127

R

read_acqknowledge() (in module neurokit2.data), 190
 rescale() (in module neurokit2.stats), 200
 rsp_amplitude() (in module neurokit2.rsp), 135
 rsp_analyze() (in module neurokit2.rsp), 135
 rsp_clean() (in module neurokit2.rsp), 136
 rsp_eventrelated() (in module neurokit2.rsp), 137
 rsp_findpeaks() (in module neurokit2.rsp), 138
 rsp_fixpeaks() (in module neurokit2.rsp), 139
 rsp_intervalrelated() (in module neurokit2.rsp), 139
 rsp_peaks() (in module neurokit2.rsp), 140
 rsp_phase() (in module neurokit2.rsp), 141
 rsp_plot() (in module neurokit2.rsp), 142
 rsp_process() (in module neurokit2.rsp), 142
 rsp_rate() (in module neurokit2.rsp), 143
 rsp_rrv() (in module neurokit2.rsp), 144
 rsp_simulate() (in module neurokit2.rsp), 145

S

signal_autocor() (in module neurokit2.signal), 165
 signal_binarize() (in module neurokit2.signal), 165

signal_changepoints() (in module neurokit2.signal), 166
 signal_decompose() (in module neurokit2.signal), 166
 signal_detrend() (in module neurokit2.signal), 167
 signal_distort() (in module neurokit2.signal), 169
 signal_filter() (in module neurokit2.signal), 170
 signal_findpeaks() (in module neurokit2.signal), 172
 signal_fixpeaks() (in module neurokit2.signal), 173
 signal_formatpeaks() (in module neurokit2.signal), 175
 signal_interpolate() (in module neurokit2.signal), 175
 signal_merge() (in module neurokit2.signal), 176
 signal_period() (in module neurokit2.signal), 176
 signal_phase() (in module neurokit2.signal), 177
 signal_plot() (in module neurokit2.signal), 178
 signal_power() (in module neurokit2.signal), 178
 signal_psd() (in module neurokit2.signal), 179
 signal_rate() (in module neurokit2.signal), 180
 signal_recompose() (in module neurokit2.signal), 181
 signal_resample() (in module neurokit2.signal), 182
 signal_simulate() (in module neurokit2.signal), 184
 signal_smooth() (in module neurokit2.signal), 184
 signal_synchrony() (in module neurokit2.signal), 185
 signal_zerocrossings() (in module neurokit2.signal), 186
 standardize() (in module neurokit2.stats), 201
 summary_plot() (in module neurokit2.stats), 201