# onebone

Kyunghwan Kim

Aug 11, 2022

# DOCUMENTATION

Onebone is an open-source software for signal analysis about predictive maintenance, being used for research activities at ONEPREDICT Corp.. It includes modules for preprocessing, health feature, and more. If you need to analyze signals for industrial equipments like turbines, a rotary machinery or componets like gears, bearings, give onebone a try!

CHAPTER

ONE

# GETTING STARTED

**1. Prerequisite**

onebone requires Python 3.6.5+.

**2. Installation**

onebone can be installed via pip from PyPI.

```
$ pip install onebone
```

It can be checked as follows whether the onebone has been installed.

```
>>> import onebone
>>> onebone.__version__
```

**3. Usage**

It assumes that the user has already installed the onebone package. You can import directly the function, for example:

```
>>> from onebone.feature import tacho_to_rpm
```

CHAPTER

# TWO

# CONTENTS

## 2.1 onebone

### 2.1.1 onebone.feature

#### onebone.feature.correlations

Signal correlation features.

- Author: Kibum Kim
- Contact: kibum.kim@onepredict.com

onebone.feature.correlations.**phase_alignment**(*y: ndarray*, *fs: Union[int, float]*) → Tuple[ndarray, ndarray]

Compute phase alignment(PA) of a set of 1D signals

$$PA_f = \left| \frac{1}{n} \sum_j e^{iw_{j,f}} \right|$$

Where $PA_f$ is phase alignment value on frequency $f$, $n$ is the number of signals, and $j$ is the index of each signals. $w_{j,f}$ denotes phase of signal $j$ at frequency $f$.

This function computes mean vector of unit phase vectors of 1D signals. This process is repeated for every frequency unit.

**Parameters**

- **y** (*numpy.ndarray of shape (n, signal_length)*) – n denotes the number of signals, and each signal shold be 1D time domain.
- **fs** (*int or float*) – Sample rate. The sample rate is the number of samples per unit time.

**Returns**

- **freq** (*numpy.ndarray*) – Frequency array
- **phase_alignment** (*numpy.ndarray*) – Phase alignment value of each frequency

**Examples**

```
>>> fs, n = 1000.0, 1000
>>> x = np.linspace(0.0, n  / fs, n, endpoint=False)
>>> signal = 3 * np.sin(50. * np.pi * x) + 2 * np.sin(80.0 * np.pi * x)
```

(continues on next page)

(continued from previous page)

```
>>> y = np.array([signal + np.random.uniform(low=-1, high=1, size=(n,)) for i in␣
↪range(10)])
>>> freq, pa_result = phase_alignment(y, fs)
```

### onebone.feature.frequency

Frequency domain feature.

- Author: Kangwhi Kim
- Contact: kangwhi.kim@onepredict.com

onebone.feature.frequency.**mdf**(*amp: ndarray*, *fs: Union[int, float] = 1*, *freq_range: Optional[Tuple] = None*, *axis: int = -1*, *keepdims: bool = False*) → float

Compute the median frequency.

$$\sum_{j=1}^{MDF} A_j = \sum_{j=MDF}^{M} A_j = \frac{1}{2}\sum_{j=1}^{M} P_j \quad [1]$$

Where $A_j$ is the amplitude value of spectrum at the frequency bin $j$, and $M$ is the length of frequency bin.

**Parameters**

- **amp** (*numpy.ndarray of shape (signal_length,), (n, signal_length)*) – The amplitudes of a spectrum of time-domain signals.
- **fs** (*int or float, default=1*) – Sample rate. The sample rate is the number of samples per unit time. If *fs* is 1, then *mdf* is the normalized frequency; (0 ~ 1).
- **freq_range** (*tuple, default=None*) – Frequency range, specified as a two-element tuple of real values. If *freq_range* is None, then *mdf* uses the entire bandwidth of the input signal. That is, *freq_range* is (0, *fs* / 2).
- **axis** (*int, default=-1*) – Axis along which *mdf* is performed. The default, *axis `=-1, will calculate the `mdf* along last axis of *x*. If *axis* is negative, it counts from the last to the first axis.
- **keepdims** (*bool, default=False*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one.

**Returns**

**mdf** (*float or numpy.ndarray*) – Median frequency. If *fs* is 1, then *mdf* has units of cycle/sample. But, if you specify the *fs*, then *mdf* has the same units as *fs*. E.g. cycle/sec.

**References**

**Examples**

```
>>> import numpy as np
>>> fs = 100
>>> t = np.arange(0, 1, 1 / fs)
>>> x1 = np.sin(2 * np.pi * 10 * t)
```

(continues on next page)

(continued from previous page)

```
>>> x2 = np.sin(2 * np.pi * 20 * t)
>>> x3 = np.sin(2 * np.pi * 30 * t)
>>> x = x1 + x2 + x3
>>> amp = np.abs(np.fft.fft(x))
>>> mdf(amp, fs)
20
```

---

onebone.feature.frequency.**mnf**(*amp: ndarray*, *fs: Union[int, float] = 1*, *freq_range: Optional[Tuple] = None*, *axis: int = -1*, *keepdims: bool = False*) → Union[float, ndarray]

Compute the mean frequency.

$$MNF = \frac{\sum_{j=1}^{M} f_j A_j}{\sum_{j=1}^{M} A_j}$$ [1]

Where $f_j$ is the frequency value of spectrum at the bin $j$, $A_j$ is the amplitude value of spectrum at the frequency bin $j$, and $M$ is the length of frequency bin.

**Parameters**

- **amp** (*numpy.ndarray of shape (signal_length,), (n, signal_length)*) – The amplitudes of a spectrum of time-domain signals.
- **fs** (*int or float, default=1*) – Sample rate. The sample rate is the number of samples per unit time. If *fs* is 1, then *mnf* is the normalized frequency; (0 ~ 1).
- **freq_range** (*tuple, default=None*) – Frequency range, specified as a two-element tuple of real values. If *freq_range* is None, then *mnf* uses the entire bandwidth of the input signal. That is, *freq_range* is (0, *fs* / 2).
- **axis** (*int, default=-1*) – Axis along which *mnf* is performed. The default, *axis `=-1, will calculate the `mnf* along last axis of *x*. If *axis* is negative, it counts from the last to the first axis.
- **keepdims** (*bool, default=False*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one.

**Returns**

**mnf** (*float or numpy.ndarray*) – Mean frequency. If *fs* is 1, then *mnf* has units of cycle/sample. But, if you specify the *fs*, then *mnf* has the same units as *fs*. E.g. cycle/sec.

---

**References**

---

**Examples**

```
>>> import numpy as np
>>> fs = 100
>>> t = np.arange(0, 1, 1 / fs)
>>> x1 = np.sin(2 * np.pi * 10 * t)
>>> x2 = np.sin(2 * np.pi * 20 * t)
>>> x3 = np.sin(2 * np.pi * 30 * t)
>>> x = x1 + x2 + x3
>>> amp = np.abs(np.fft.fft(x))
```

(continues on next page)

(continued from previous page)

```
>>> mnf(amp, fs)
20
```

onebone.feature.frequency.**vcf**(*amp: ndarray*, *fs: Union[int, float] = 1*, *freq_range: Optional[Tuple] = None*, *axis: int = -1*, *keepdims: bool = False*) → float

Compute the variance of central frequency(mean frequency).

$$VCF = \frac{1}{\sum_{j=1}^{M} A_j} \sum_{j=1}^{M} A_j (f_j - MNF)^2 \quad [1]$$

Where $f_j$ is the frequency value of spectrum at the bin $j$, $A_j$ is the amplitude value of spectrum at the frequency bin $j$, $M$ is the length of frequency bin, $MNF$ is the mean frequency.

**Parameters**

- **amp** (*numpy.ndarray of shape (signal_length,), (n, signal_length)*) – The amplitudes of a spectrum of time-domain signals.
- **fs** (*int or float, default=1*) – Sample rate. The sample rate is the number of samples per unit time. If *fs* is 1, then *vcf* is the normalized frequency; (0 ~ 1).
- **freq_range** (*tuple, default=None*) – Frequency range, specified as a two-element tuple of real values. If *freq_range* is None, then *vcf* uses the entire bandwidth of the input signal. That is, *freq_range* is (0, *fs* / 2).
- **axis** (*int, default=-1*) – Axis along which *vcf* is performed. The default, *axis `=-1, will calculate the `vcf* along last axis of *x*. If *axis* is negative, it counts from the last to the first axis.
- **keepdims** (*bool, default=False*) – If this is set to True, the axes which are reduced are left in the result as dimensions with size one.

**Returns**

**vcf** (*float or numpy.ndarray*) – Median frequency. If *fs* is 1, then *vcf* has units of cycle/sample. But, if you specify the *fs*, then *vcf* has the same units as *fs*. E.g. cycle/sec.

**References**

**Examples**

```
>>> import numpy as np
>>> fs = 100
>>> t = np.arange(0, 1, 1 / fs)
>>> x1 = np.sin(2 * np.pi * 10 * t)
>>> x2 = np.sin(2 * np.pi * 20 * t)
>>> x3 = np.sin(2 * np.pi * 30 * t)
>>> x = x1 + x2 + x3
>>> amp = np.abs(np.fft.fft(x))
>>> vcf(x, fs)
66.666
```

### onebone.feature.gear

Condition metrics for gear.

- Author: Kyunghwan Kim, Kangwhi Kim
- Contact: kyunghwan.kim@onepredict.com, kangwhi.kim@onepredict.com

onebone.feature.gear.**na4**(*x_tsa: ndarray*, *prev_info: Tuple[int, float]*, *fs: Union[int, float]*, *rpm: float*, *freq_list: Optional[Tuple[float]] = None*, *n_harmonics: int = 2*) → Tuple[float, Tuple[int, float]]

Calculate NA4 metric.

NA4 indicates the onset of damage and continues to react to the damage as it spreads and increases in magnitude.

$$NA4 = \frac{N\sum_{i=1}^{N}(r_i - \bar{r})^4}{\left\{\frac{1}{M}\sum_{j=1}^{M}\left[\sum_{i=1}^{N}(r_{ij} - \bar{r_j})^2\right]\right\}^2} = \frac{(\mu_4)_M}{\frac{1}{M}\sum_{j=1}^{M}(\mu_2)_j}$$ [1]

Where $r$ is residual signal, $\bar{r}$ is mean value of residual signal, $N$ is total number of data points in time record, $M$ is current time record in run ensemble, $i$ is data point number in time record, $j$ is time record number in run ensemble, $\mu_2$ is the 2th central moment of $r$, and $\mu_4$ is the 4th central moment of $r$.

**Parameters**

- **x** (numpy.ndarray of shape (length of *x*,)) – Time-synchronous-averaged signal (recommended)
- **prev_info** (*tuple of (int, float)*) – The information for the 'previous time record number' in run ensemble. The first element of *prev_info* is the 'previous time record number'(M-1). The second element of *prev_info* is the average of each 2th central moment of (M-1) previous residual signals. If the current time record number is 1, the *prev_info* is (0, 0).
- **fs** (*int or float*) – Sampling rate
- **rpm** (*float*) – Revolution per minute. The unit of 'rpm' is 'rev/min'.
- **freq_list** (*None or tuple of floats, default=None*) – The frequencies of gear-meshing components. They are filtered from the 'x' signal.
- **n_harmonics** (*int, default=2*) – A positive integer specifying the number of shaft and gear meshing frequency harmonics to remove.

**Returns**

- **na4** (*float*) – A metric to not only detect the onset of damage, but also to continue to react to the damage as it increases.
- **cur_info** (*tuple of (int, float)*) – The information for the 'current time record number' in run ensemble. The first element of *cur_info* is the 'current time record number'(M). The second element of *cur_info* is the average of each 2th central moment of M residual signals.

**References**

**Examples**

```
>>> rpm = 180                                        # Revolution per minute
>>> fs = 50e3                                        # Sampling rate
>>> t = np.arange(0, (1 / 3) - 1 / fs, 1 / fs)    # Sample times
>>> freq_list = (51, 153)                            # Gear mesh frequencies
>>> f = (rpm/60,) + freq_list                        # Frequencies of signals
>>> prev_info = (0, 0)          # The information for the 'previous time record number
↪' in run ensemble.
>>> n_harmonics = 2
>>> na4_list = []
```

# Assume that the gear condition is getting worse.

```
>>> for k in range(1, 11):
        # Motor shaft rotation and harmonic
        shaft_signal = np.sin(2 * np.pi * f[0] * t) + np.sin(2 * np.pi * 2 * f[0] *␣
↪t)
        # Gear mesh vibration and harmonic for a pair of gears
        gm1_signal = 3 * np.sin(2 * np.pi * f[1] * t) + 3 * np.sin(2 * np.pi * 2 *␣
↪f[1] * t)
        # Gear mesh vibration and harmonic for a pair of gears
        gm2_signal = 4 * np.sin(2 * np.pi * f[2] * t) + 4 * np.sin(2 * np.pi * 2 *␣
↪f[2] * t)
        # Fault component signal
        fault_signal = 2 * (k / 6) * np.sin(2 * np.pi * 10 * f[0] * t)
        # New signal is the sum of gm1_signal, gm2_signal, and fault_signal.
        new_signal = shaft_signal + gm1_signal + gm2_signal + fault_signal
```

# Calculate NA4 **na4_**, cur_info = na4(new_signal, prev_info, fs, rpm, freq_list, n_harmonics) prev_info = cur_info na4_list.append(**na4_**)

```
>>> print(na4_list)
[1.536982703280907, 3.857738227803903, 5.590250485891509, 6.835250547872656, 7.
↪755227746173137,
 8.457654233606695, 9.009683995782796, 9.454160950683573, 9.819352835339927, 10.
↪12454304712786]
```

## onebone.feature.snr

onebone.feature.snr.**snr**(*x: ndarray*, *fs: Union[int, float] = 1000*, *nperseg: int = 256*, *noverlap: int = 32*) → ndarray

Extract the SNR(Signal-to-Noise-Ratio) feature from the signal using the 'STFT'. SNR is the ratio between max power intensity of frequency and power of other frequencies at time t in the STFT spectrum.

$$P_{signal}(t) = max(|STFT(t, f)|)$$

$$SNR(t) = \frac{P_{signal}(t)}{\sum_f |STFT(t, f)| - P_{signal}(t)}$$

**Parameters**

- **x** (*numpy.ndarray*) – 1d-signal data. Must be real.

- **fs** (*int or float*) – Sampling rate.
- **nperseg** (*int, default=256*) – Length of each segment.
- **noverlap** (*int, default=32*) – Number of points to overlap between segments.

**Returns**

**snr** (*numpy.ndarray*) – SNR of the *x*, 1d-array.

**Examples**

```
>>> fs = 1000.0
>>> t = np.linspace(0, 1, int(fs))
>>> x = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> snr_array = snr_feature(x, fs)
```

**Notes**

1. Get a snr array by using snr_feature for one of given signals.
2. Make the segments of snr array and get the mean of each segment.
3. Compare the mean of SNR between normal states and fault states. e.g. SNR_fault = np.mean(SNR_fault_array), SNR_normal = np.mean(SNR_normal_array)
4. Typically, SNR_fault is smaller than SNR_normal.

### onebone.feature.tacho

Convert tacho to angle or RPM.

- Author: Kangwhi Kim
- Contact: kangwhi.kim@onepredict.com

onebone.feature.tacho.**tacho_to_angle**(*x: ndarray*, *fs: Union[int, float]*, *state_levels_trh: Union[int, float]*, *indices_trh: int = 2*, *pulses_per_rev: int = 1*, *output_fs: Optional[Union[int, float]] = None*, *fit_type: str = 'linear'*) → Tuple[ndarray, ndarray, ndarray]

Extract angle signal from tachometer pulses.

**Parameters**

- **x** (*numpy.ndarray*) – Tachometer pulse signal(1-D).
- **fs** (*int or float*) – Sample rate.
- **state_levels_trh** (*int or float*) – The difference between state levels used to identify pulses. (The state levels used to identify pulses.)
- **indices_trh** (*int, default=2*) – The difference between indices of the first samples of high-level-state of pulses.
- **pulses_per_rev** (*int, default=1*) – Number of tachometer pulses per revolution.
- **output_fs** (*int or float, default=None*) – Output sample rate. When the default is None, the *output_fs* is the *fs*.
- **fit_type** (*str, default="linear"*) – Fitting method

**Returns**

- **angle** (*numpy.ndarray*) – Rotational angle(1-D).
- **t** (*numpy.ndarray*) – Time(1-D) expressed in seconds.
- **tp** (*numpy.ndarray*) – Pulse locations(1-D) expressed in seconds.

---

**Examples**

```
>>> x = np.array([0,1,0,1,0,0,1,0,0,1])
>>> fs = 10
>>> state_levels_trh = 0.5
```

```
>>> angle, t, tp = tacho_to_angle(x, fs, state_levels_trh)
```

```
>>> angle
array([-6.28318531e+00, -4.18879020e+00, -2.09439510e+00,  1.16262283e-15,
       2.09439510e+00,  4.18879020e+00,  6.28318531e+00,  8.37758041e+00,
       1.04719755e+01,  1.25663706e+01])
>>> t
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> tp
array([0.3, 0.6, 0.9])
```

---

onebone.feature.tacho.**tacho_to_rpm**(*x: ndarray*, *fs: Union[int, float]*, *state_levels_trh: Union[int, float]*, *indices_trh: int = 2*, *pulses_per_rev: int = 1*, *output_fs: Optional[Union[int, float]] = None*, *fit_type: str = 'linear'*) → Tuple[ndarray, ndarray, ndarray]

Extract RPM signal from tachometer pulses.

**Parameters**

- **x** (*numpy.ndarray*) – Tachometer pulse signal(1-D).
- **fs** (*int or float*) – Sample rate.
- **state_levels_trh** (*int or float*) – The difference between state levels used to identify pulses. (The state levels used to identify pulses.)
- **indices_trh** (*int, default=2*) – The difference between indices of the first samples of high-level-state of pulses.
- **pulses_per_rev** (*int, default=1*) – Number of tachometer pulses per revolution.
- **output_fs** (*int or float, default=None*) – Output sample rate. When the default is None, the *output_fs* is the *fs*.
- **fit_type** (*str, default="linear"*) – Fitting method.

**Returns**

- **rpm** (*numpy.ndarray*) – Rotational speed(1-D).
- **t** (*numpy.ndarray*) – Time(1-D) expressed in seconds.
- **tp** (*numpy.ndarray*) – Pulse locations(1-D) expressed in seconds.

---

**Examples**

```
>>> x = np.array([0,1,0,1,0,0,1,0,0,1])
>>> fs = 10
>>> state_levels_trh = 0.5
```

```
>>> rpm, t, tp = tacho_to_rpm(x, fs, state_levels_trh)
```

```
>>> rpm
array([200., 200., 200., 200., 200., 200., 200., 200., 200., 200.])
>>> t
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> tp
array([0.3, 0.6, 0.9])
```

---

### onebone.feature.tacholess

Track and extract a instantaneous frequency(IF) profile from vibration signal

- Author: Kangwhi Kim
- Contact: kangwhi.kim@onepredict.com

onebone.feature.tacholess.**two_step_if**(*x: ndarray*, *fs: Union[int, float]*, *f_start: Union[int, float]*, *f_tol: Union[int, float]*, *filter_bw: Union[int, float]*, *window: Union[str, Tuple, ndarray] = 'hann'*, *nperseg: int = 256*, *noverlap: Optional[int] = None*, ***kwargs*) → ndarray

Track and extract a instantaneous frequency(IF) profile from vibration signal, based on Two-step method.

---

**Note:** If you have a tachometer pulse signal, use *tacho_to_rpm* function.

---

*two_step_if* uses the local maxima technique [1] for IF estimation, as follows;

$$f_{max}(t) = \underset{f}{Argmax}|X(t,f)|^2, \quad \text{for } f \in \Delta f_t$$

$$\Delta f_t \subset \{f_{max}(t - d\tau) - \delta f,\ f_{max}(t - d\tau) + \delta f\}$$

where, $\delta f$ is the given frequency tolerance for maxima detection, $X(t, f)$ is the STFT of signal $x(t)$ computed for frequency values in set $\Delta f_t$, specified from the previous estimate $f_{max}(t - d\tau)$. tau is time defined as a window position. Note that for t=0, $\Delta f_t$ should be given by the user.

**Parameters**

- **x** (numpy.ndarray of shape (length of *x*, )) – A vibration 1-D signal.
- **fs** (*int or float*) – Sampling rate. [Hz]
- **f_start** (*int or float*) – Starting frequency point for the IF estimation. [Hz]
- **f_tol** (*int or float*) – Frequency tolerance for maxima detection. [Hz]
- **filter_bw** (*int or float*) – frequency bandwidth for filtration. [Hz]
- **window** (*str or tuple or numpy.ndarray, default="hann"*) – Desired window to use for *scipy.signal.stft*. If window is a string or tuple, it is passed to get_window to generate

the window values, which are DFT-even by default. See get_window for a list of windows and required parameters. If window is array_like it will be used directly as the window and its length must be nperseg. Defaults to a Hann window.

- **nperseg** (*int, default=256*) – Length of each segment for *scipy.signal.stft*.
- **noverlap** (*int, default=None*) – Number of points to overlap between segments. If None, noverlap = nperseg // 2. Defaults to None. When specified, the COLA constraint must be met; i.e. (x.shape[axis] - nperseg) % (nperseg-noverlap) == 0. For more information, see Notes of *scipy.signal.stft*.
- ****kwargs** (*dict*) – Additional parameters for *scipy.signal.stft*.

**Returns**

**inst_freq** (numpy.ndarray of shape (*x.size* - 1,)) – A instantaneous frequency(IF) profile. For improved results try to manipulate *f_tol* and *f_tol* parameters. You might also change spectrogram options.

**References**

**Examples**

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Generate a test signal, sin wave whose frequency is modulated around 3kHz, corrupted by white noise.

```
>>> fs = 1e4
>>> n = 1e5
>>> time = np.arange(n) / fs
>>> mod = 500 * np.cos(2 * np.pi * 0.1 * time)
>>> carrier = 3 * np.sin(2 * np.pi * 3e3 * time + mod)
>>> x = carrier + np.random.rand(carrier.size) / 5  # test signal
```

Extract the instantaneous frequency from the signal.

```
>>> inst_freq = two_step_if(x, fs, f_start=3e3, f_tol=50, filter_bw=5,
window='hann', nperseg=4096, noverlap=3985)
```

Plot the instantaneous frequency(IF) profile.

```
>>> time = np.arange(x.size) / fs
>>> time = time[:-1]
>>> plt.plot(time, inst_freq)
>>> plt.title('Estimated the instantaneous frequency(IF) profile')
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.show()
```

### onebone.feature.time

Signal analysis for the time domain.

- Author: Kyunghwan Kim
- Contact: kyunghwan.kim@onepredict.com

onebone.feature.time.**crest_factor**(*x: ndarray*, *axis: Optional[int] = None*) → ndarray

Peak to average ratio along an axis.

$$crestfactor = \frac{|x_{peak}|}{x_{rms}}$$

**Parameters**

- **x** (*numpy.ndarray*) – The data.
- **axis** (*None or int, default=None*) – Axis along which to calculate crest factor. By default, flatten the array.

**Returns**

**crest_factor** (*numpy.ndarray*) – Peak to average ratio of x.

---

**Examples**

```
>>> x = np.array([[4, 9, 2, 10],
...               [6, 9, 7, 12]])
>>> crest_factor(x, axis=0)
array([0.39223219, 0.        , 0.97128567, 0.18107148])
>>> crest_factor(x, axis=1)
array([1.12855283, 0.68155412])
>>> crest_factor(x)
1.2512223376239555
```

---

onebone.feature.time.**kurtosis**(*x: ndarray*, *axis: int = 0*, *fisher: bool = True*, *bias: bool = True*) → ndarray

---

**Note:** This method uses scipy.stats.kurtosis method as it is.

---

Compute the kurtosis (Fisher or Pearson) of a signal.

**Parameters**

- **x** (*numpy.ndarray*) – The data.
- **axis** (*None or int, default=0*) – Axis along which the kurtosis is calculated. If None, compute over the whole array a.
- **fisher** (*bool, default=True*) – If True, Fisher's definition is used (normal ==> 0.0). If False, Pearson's definition is used (normal ==> 3.0).
- **bias** (*bool, default=True*) –
- **False** (*If*) –
- **bias.** (*then the calculations are corrected for statistical*) –

**Returns**

**kurtosis** (*numpy.ndarray*) – The kurtosis of values along an axis. If all values are equal, return -3 for Fisher's definition and 0 for Pearson's definition.

**Examples**

```
>>> x = np.array([[4, 9, 2, 10],
...               [6, 9, 7, 12]])
>>> kurtosis(x)
array([-2., -3., -2., -2.])
```

onebone.feature.time.**peak2peak**(*x*, *axis: Optional[int] = None*) → ndarray

**Note:** This method uses numpy.ptp method as it is.

Maximum to minimum difference along an axis.

**Parameters**

- **x** (*array_like*) – The data.
- **axis** (*None or int, default=None*) – Axis along which to find the peaks. By default, flatten the array.

**Returns**

**p2p** (*numpy.ndarray*) – The difference between the maximum and minimum values in x.

**Examples**

```
>>> x = np.array([[4, 9, 2, 10],
...               [6, 9, 7, 12]])
>>> ptp(x, axis=1)
array([8, 6])
>>> ptp(x, axis=0)
array([2, 0, 5, 2])
>>> ptp(x)
10
```

onebone.feature.time.**rms**(*x: ndarray*, *axis: Optional[int] = None*) → ndarray

Root mean square along an axis.

**Parameters**

- **x** (*numpy.ndarray*) – The data.
- **axis** (*None or int, default=None*) – Axis along which to calculate rms. By default, flatten the array.

**Returns**

**rms** (*numpy.ndarray*) – Root mean square value of x.

**Examples**

```
>>> x = np.array([[4, 9, 2, 10],
...               [6, 9, 7, 12]])
>>> rms(x, axis=0)
array([ 5.09901951,  9.        ,  5.14781507, 11.04536102])
>>> rms(x, axis=1)
array([7.08872344, 8.80340843])
>>> rms(x)
7.99218368157289
```

### 2.1.2 onebone.math

#### onebone.math.integrate

Frequency domain feature.

- Author: Kangwhi Kim
- Contact: kangwhi.kim@onepredict.com

onebone.math.integrate.**integrate_trapezoid**(*y*, *x=None*, *dx: float = 1.0*, *axis: int = -1*) → Union[float, ndarray]

**Note:** This method uses numpy.trapz method as it is.

Integrate along the given axis using the composite trapezoidal rule.

If *x* is provided, the integration happens in sequence along its elements - they are not sorted.

Integrate $y$ ($x$) along each 1d slice on the given axis, compute $\int y(x)dx$. When $x$ is specified, this integrates along the parametric curve, computing $\int_t y(t)dt = \int_t y(t)\left.\frac{dx}{dt}\right|_{x=x(t)} dt$.

**Parameters**

- **y** (*array_like*) – Input array to integrate.
- **x** (*array_like, optional, default=None*) – The sample points corresponding to the $y$ values. If $x$ is None, the sample points are assumed to be evenly spaced $dx$ apart.
- **dx** (*float, optional, default=1*) – The spacing between sample points when $x$ is None.
- **axis** (*int, optional, default=-1*) – The axis along which to integrate.

**Returns**

**trapz** (*float or ndarray*) – Definite integral of ‘y’ = n-dimensional array as approximated along a single axis by the trapezoidal rule. If ‘y’ is a 1-dimensional array, then the result is a float. If ‘n’ is greater than 1, then the result is an ‘n-1’ dimensional array.

**See also:**

numpy.sum, numpy.cumsum

**Notes**

Image[2] illustrates trapezoidal rule – y-axis locations of points will be taken from *y* array, by default x-axis distances between points will be 1.0, alternatively they can be provided with *x* array or with *dx* scalar. Return value will be equal to combined area under the red lines.

**References**

**Examples**

```
>>> np.trapz([1,2,3])
4.0
>>> np.trapz([1,2,3], x=[4,6,8])
8.0
>>> np.trapz([1,2,3], dx=2)
8.0
```

Using a decreasing *x* corresponds to integrating in reverse:

```
>>> np.trapz([1,2,3], x=[8,6,4])
-8.0
```

More generally *x* is used to integrate along a parametric curve. This finds the area of a circle, noting we repeat the sample which closes the curve:

```
>>> theta = np.linspace(0, 2 * np.pi, num=1000, endpoint=True)
>>> np.trapz(np.cos(theta), x=np.sin(theta))
3.141571941375841
```

```
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.trapz(a, axis=0)
array([1.5, 2.5, 3.5])
>>> np.trapz(a, axis=1)
array([2.,  8.])
```

### 2.1.3 onebone.preprocessing

#### onebone.preprocessing.feature_selection

Feature Selection methods.

- Author: Junha Jeon
- Contact: junha.jeon@onepredict.com

onebone.preprocessing.feature_selection.**fs_crosscorrelation**(*x: ndarray*, *refer: ndarray*, *output_col_num: int*) → ndarray

[2] Illustration image: https://en.wikipedia.org/wiki/File:Composite_trapezoidal_rule_illustration.png

**Note:** This method uses scipy.signal.correlate.

Reduce the dimension of input data by removing the signals which have small cross correlation with reference signal.

**Parameters**

- **x** (*numpy.ndarray of shape (data_length, n_features)*) – The data.
- **refer** (*numpy.ndarray of shape (data_length,)*) – The reference data.
- **output_col_num** (*int*) – Number of columns after dimension reduction.

**Returns**

**x_tr** (*numpy.ndarray of shape (data_length, n_features)*) – The data after dimension reduction.

**Examples**

```
>>> t = np.linspace(0, 1, 1000)
>>> a = 1.0 * np.sin(2 * np.pi * 30.0 * t)
>>> b = 5.0 * np.sin(2 * np.pi * 30.0 * t)
>>> x = np.stack([a, b], axis=1)
>>> x.shape
(1000, 2)
>>> refer = 1.0 * np.sin(2 * np.pi * 10.0 * t)
>>> x_dimreduced = fs_crosscorrelation(x, refer, output_col_num=1)
>>> x_dimreduced.shape
(1000, 1)
```

### onebone.preprocessing.pd

Transform PRPS(Phase Resolved Pulse Sequence) format pd data to PRPD(Phase Resolved Partial Discharge) format.

- Author: Hyunjae Kim
- Contact: hyunjae.kim@onepredict.com

onebone.preprocessing.pd.**ps2pd**(*ps*, *range_amp: Tuple[int, int] = (0, 256)*, *resol_amp: int = 128*) → ndarray

Transform prps(phase resolved pulse sequance) to a prpd(phaes resolved partial discharge) by marginalizing time dimension.

**Parameters**

- **ps** (*array_like of shape (n_resolution_phase, n_timestep)*) – The data. Ex: kepco standard=(3600, 128)
- **range_amp** (*tuple (min, max), default=(0, 256)*) – Measurement range of PD DAQ. Refers to DAQ manufacture.
- **resol_amp** (*int, default=128*) – Desired resolution of amplitude resolution for transformd prpd.

**Returns**

**pd** (*numpy.ndarray of shape (n_resolution_phase, n_resolution_amplitude)*) – The transformed prpd.

**Examples**

```
>>> ps = np.random.random([3600,128])
>>> ps2pd(ps)
array([[0., 0., 0., ..., 0., 0., 0.],
    [0., 0., 0., ..., 0., 0., 0.],
    ...,
    [0., 0., 0., ..., 0., 0., 0.]])
```

### onebone.preprocessing.scaling

Data scaling methods.

- Author: Kyunghwan Kim
- Contact: kyunghwan.kim@onepredict.com

onebone.preprocessing.scaling.**minmax_scaling**(*x*, *feature_range: Tuple[int, int] = (0, 1)*, *axis: int = 0*) → ndarray

**Note:** This method uses sklearn.preprocessing.minmax_scale method as it is.

Transform features by scaling each feature to a given range.

$$x' = \frac{(x - x_{min})}{(x_{max} - x_{min})}$$

**Parameters**

- **x** (*array_like of shape (n_samples, n_features)*) – The data.
- **feature_range** (*tuple (min, max), default=(0, 1)*) – Desired range of transformed data.
- **axis** (*int, default=0*) – Axis used to scale along.

**Returns**

**x_tr** (*numpy.ndarray of shape (n_samples, n_features)*) – The transformed data.

**Examples**

```
>>> a = list(range(9))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> minmax_scaling(a)
array([0.   , 0.125, 0.25 , 0.375, 0.5  , 0.625, 0.75 , 0.875, 1.   ])
```

onebone.preprocessing.scaling.**zscore_scaling**(*x*, *axis: int = 0*)

**Note:** This method uses sklearn.preprocessing.scale method as it is.

Transform input data so that they can be described as a normal distribution.

$$x' = \frac{(x - x_{mean})}{x_{std}}$$

**Parameters**

- **x** (*array_like of shape (n_samples, n_features)*) – The data.
- **axis** (*int, default=0*) – Axis used to compute the means and standard deviations along.

**Returns**

**x_tr** (*numpy.ndarray of shape (n_samples, n_features)*) – The transformed data.

**Examples**

```
>>> a = list(range(9))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> zscore_scaling(a)
array([-1.54919334, -1.161895  , -0.77459667, -0.38729833,  0.,
        0.38729833,  0.77459667,  1.161895  ,  1.54919334])
```

### 2.1.4 onebone.signal

#### onebone.signal.denoise

Signal denoising method.

- Author: Kyunghwan Kim
- Contact: kyunghwan.kim@onepredict.com

onebone.signal.denoise.**wavelet_denoising**(*signal: ndarray*, *wavelet: str*, *axis: int = -1*, *level: Optional[int] = None*) → ndarray

Denoise signal using Discrete Wavelet Transform(DWT).

1. Multilevel Wavelet Decomposition.
2. Identify a thresholding technique.
3. Threshold and Reconstruct.

**Parameters**

- **signal** (*numpy.ndarray*) – Input signal.
- **wavelet** (*str*) – Wavelet name. See this page.
- **axis** (*int, default=-1*) – Axis over which to compute the DWT.
- **level** (*int, default=None*) – If level is None, then it will be calculated using the `pywt.dwt_max_level` function.

**Returns**

**out** (*numpy.ndarray*) – Denoised signal.

**Examples**

Apply the filter to 1d signal.

```
>>> signal = np.array([10.0] * 10 + [0.0] * 10)
>>> signal += np.random.random(signal.shape)
```

```
>>> wavelet = "db1"
>>> denoised_signal = wavelet_denoising(signal, wavelet, level=2)
```

### onebone.signal.envelope

Extract envelope.

- Author: Kangwhi Kim
- Contact: kangwhi.kim@onepredict.com

onebone.signal.envelope.**envelope_hilbert**(*x*, *axis: int = -1*) → ndarray

Extract the envelope from the signal using the 'Hilbert transform'.

**Parameters**

- **x** (*array_like*) – Signal data. Must be real.
- **axis** (*int, default=-1*) – Axis along which to do the transformation.

**Returns**

**y** (*numpy.ndarray*) – Envelope of the *x*, of each 1-D array along *axis*

### onebone.signal.fft

The module about fast fourier transform.

- Author: Daeyeop Na, Kangwhi Kim
- Contact: daeyeop.na@onepredict.com, kangwhi.kim@onepredict.com

onebone.signal.fft.**positive_fft**(*signal: ndarray*, *fs: Union[int, float]*, *hann: bool = False*, *normalization: bool = False*, *axis: int = -1*) → Tuple[ndarray, ndarray]

Positive 1D fourier transformation.

**Parameters**

- **signal** (*numpy.ndarray*) – Original time-domain signal
- **fs** (*Union[int, float]*) – Sampling rate
- **hann** (*bool, default = False*) – hann function used to perform Hann smoothing. It is implemented when hann is True
- **normalization** (*bool, default = False*) – Normalization after Fourier transform
- **axis** (*int, default=-1*) – The axis of the input data array along which to apply the fourier Transformation.

**Returns**

- **freq** (*numpy.ndarray*) – frequency If input shape is [signal_length,], output shape is freq = [signal_length,]. If input shape is [n, signal_length,], output shape is freq = [signal_length,].
- **mag** (*numpy.ndarray*) – magnitude If input shape is [signal_length,], output shape is mag = [signal_length,]. If input shape is [n, signal_length,], output shape is mag = [n, signal_length,].

**Examples**

```
>>> n = 400  # array length
>>> fs = 800  # Sampling frequency
>>> t = 1 / fs  # Sample interval time
>>> x = np.linspace(0.0, n * t, n, endpoint=False) # time
>>> y = 3 * np.sin(50.0 * 2.0 * np.pi * x) + 2 * np.sin(80.0 * 2.0 * np.pi * x)
>>> signal = y
>>> freq, mag = positive_fft(signal, fs,  hann = False, normalization = False, axis␣
↪= -1)
>>> freq = np.around(freq[np.where(mag > 1)])
>>> freq
[50., 80.]
```

### onebone.signal.filter

**A frequency filter to leave only a specific frequency band.**
and a filter that replaces outlier values in data with other values.

- Author: Kyunghwan Kim, Sunjin Kim
- Contact: kyunghwan.kim@onepredict.com, sunjin.kim@onepredict.com

onebone.signal.filter.**bandpass_filter**(*signal: ndarray*, *fs: Union[int, float]*, *l_cutoff: Union[int, float]*, *h_cutoff: Union[int, float]*, *order: int = 5*, *axis: int = -1*) → ndarray

1D Butterworth bandpass filter.

**Parameters**

- **signal** (*numpy.ndarray*) – Original time-domain signal.
- **fs** (*Union[int, float]*) – Sampling rate.
- **l_cutoff** (*Union[int, float]*) – Low cutoff frequency.
- **h_cutoff** (*Union[int, float]*) – High cutoff frequency.
- **order** (*int, default=5*) – Order of butterworth filter.
- **axis** (*int, default=-1*) – The axis of the input data array along which to apply the linear filter.

**Returns**
**out** (*numpy.ndarray*) – Filtered signal. If input shape is [signal_length,], output shape is [signal_length,]. If input shape is [n, signal_length,], output shape is [n, signal_length,].

**Examples**

Apply the filter to 1d signal. And then check the frequency component of the filtered signal.

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 500.0 * t)
>>> signal.shape
(5000,)
>>> freq_x = np.fft.rfftfreq(signal.size, 1 / fs)[:-1]
>>> origin_fft_mag = abs((np.fft.rfft(signal) / signal.size)[:-1] * 2)
>>> origin_freq = freq_x[np.where(origin_fft_mag > 0.5)]
>>> origin_freq
[ 20., 100., 500.]
>>> filtered_signal = bandpass_filter(signal, fs, l_cutoff=50, h_cutoff=300)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal) / signal.size)[:-1] * 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 100.]
```

Apply the filter to 2d signal (axis=0).

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal = np.stack([signal, signal]).T
>>> signal.shape
(5000, 2)
>>> filtered_signal = bandpass_filter(signal, fs, l_cutoff=50, h_cutoff=300, axis=0)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal[:, 0]) / signal.size)[:-1]␣
↪* 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 100.]
```

---

onebone.signal.filter.**bandpass_filter_ideal**(*signal: ndarray*, *fs: Union[int, float]*, *l_cutoff: Union[int, float]*, *h_cutoff: Union[int, float]*) → ndarray

**Warning:** This method **may cause distortion of signal**. Generally, this operates well on signals extracted in low resolution. In order to check the distortion of signals, it is recommended to monitor the linear transition of phase.

1D ideal bandpass filter.

**Parameters**

- **signal** (*numpy.ndarray*) – Original time-domain signal.
- **fs** (*Union[int, float]*) – Sampling rate.
- **l_cutoff** (*Union[int, float]*) – Low cutoff frequency.
- **h_cutoff** (*Union[int, float]*) – High cutoff frequency.

**Returns**

**out** (*numpy.ndarray*) – Filtered signal. Input shape is [signal_length,] and output shape is [signal_length,].

---

**Examples**

Apply the filter to 1d signal. And then check the frequency component of the filtered signal.

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 500.0 * t)
>>> signal.shape
(5000,)
>>> freq_x = np.fft.rfftfreq(signal.size, 1 / fs)[:-1]
>>> origin_fft_mag = abs((np.fft.rfft(signal) / signal.size)[:-1] * 2)
>>> origin_freq = freq_x[np.where(origin_fft_mag > 0.5)]
>>> origin_freq
[ 20., 100., 500.]
>>> filtered_signal = bandpass_filter_ideal(signal, fs, l_cutoff=50, h_cutoff=300)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal) / signal.size)[:-1] * 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 100.]
```

---

onebone.signal.filter.**bandstop_filter**(*signal: ndarray*, *fs: Union[int, float]*, *l_cutoff: Union[int, float]*, *h_cutoff: Union[int, float]*, *order: int = 5*, *axis: int = -1*) → ndarray

1D Butterworth bandstop filter.

**Parameters**

- **signal** (*numpy.ndarray*) – Original time-domain signal.
- **fs** (*Union[int, float]*) – Sampling rate.
- **l_cutoff** (*Union[int, float]*) – Low cutoff frequency.
- **h_cutoff** (*Union[int, float]*) – High cutoff frequency.
- **order** (*int, default=5*) – Order of butterworth filter.
- **axis** (*int, default=-1*) – The axis of the input data array along which to apply the linear filter.

**Returns**

**out** (*numpy.ndarray*) – Filtered signal. If input shape is [signal_length,], output shape is [signal_length,]. If input shape is [n, signal_length,], output shape is [n, signal_length,].

---

**Examples**

Apply the filter to 1d signal. And then check the frequency component of the filtered signal.

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
```

(continues on next page)

(continued from previous page)

```
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 500.0 * t)
>>> signal.shape
(5000,)
>>> freq_x = np.fft.rfftfreq(signal.size, 1 / fs)[:-1]
>>> origin_fft_mag = abs((np.fft.rfft(signal) / signal.size)[:-1] * 2)
>>> origin_freq = freq_x[np.where(origin_fft_mag > 0.5)]
>>> origin_freq
[ 20., 100., 500.]
>>> filtered_signal = bandstop_filter(signal, fs, l_cutoff=50, h_cutoff=300)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal) / signal.size)[:-1] * 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 20., 500.]
```

Apply the filter to 2d signal (axis=0).

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal = np.stack([signal, signal]).T
>>> signal.shape
(5000, 2)
>>> filtered_signal = bandstop_filter(signal, fs, l_cutoff=50, h_cutoff=300, axis=0)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal[:, 0]) / signal.size)[:-1]␣
↪* 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 20., 500.]
```

---

onebone.signal.filter.**hampel_filter**(*x: ndarray*, *window_size: int*, *n_sigma: float = 3*) → Tuple[ndarray, list]

A hampel filter removes outliers. Estimate the median and standard deviation of each sample using MAD(Median Absolute Deviation) in the window range set by the user. If the MAD > 3 * sigma condition is satisfied, the value is replaced with the median value.

$$m_i = median(x_{i-k_{left}}, x_{i-k_{left}+1}, ..., x_{i+k_{right}-1}, x_{i+k_{right}})$$

$$MAD_i = median(|x_{i-k} - m_i|, ..., |x_{i+k} - m_i|)$$

$$\sigma_i = \kappa * MAD_i$$

Where $k_{left}$ and $k_{right}$ are the number of neighbors on the left and right sides, respectively, based on x_i ($k_{left} + k_{right}$ = window samples). $m_i$ is Local median, $MAD_i$ is median absolute deviation which is the residuals (deviations) from the data's median. $\sigma_i$ is the MAD may be used similarly to how one would use the deviation for the average. In order to use the MAD as a consistent estimator for the estimation of the standard deviation $\sigma$, one takes $\kappa * MAD_i$. $\kappa$ is a constant scale factor, which depends on the distribution. For normally distributed data $\kappa$ is taken to be $\kappa$ = 1.4826

**Parameters**

- **x** (*numpy.ndarray*) – 1d-timeseries data. The shape of x must be (signal_length,) .

- **window_size** (*int*) – Lenght of the sliding window. Only integer types are available, and the window size must be adjusted according to your data.
- **n_sigma** (*float, defalut=3*) – Coefficient of standard deviation.

**Returns**

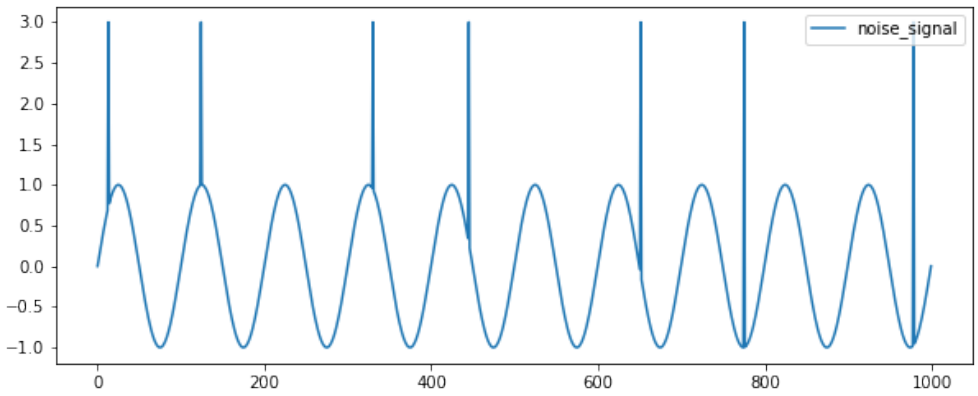
- **filtered_x** (*numpy.ndarray*) – A value from which outlier or NaN has been removed by the filter.
- **index** (*list*) – Returns the index corresponding to outlier.

**References**

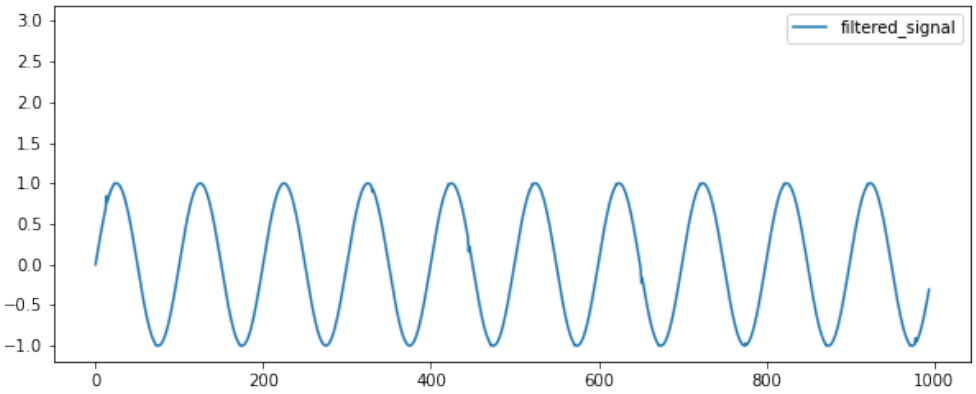
[1] Pearson, Ronald K., et al. "Generalized hampel filters." EURASIP Journal on Advances in Signal Processing 2016.1 (2016): 1-18. DOI: 10.1186/s13634-016-0383-6

**Examples**

```
>>> fs = 1000.0
>>> t = np.linspace(0, 1, int(fs))
>>> y = np.sin(2 * np.pi * 10.0 * t)
>>> np.put(y, [13, 124, 330, 445, 651, 775, 978], 3)
>>> plt.plot(y) # noise_signal
```




```
>>> filtered_signal = hampel_filter.hampel_filter(y, window_size=5)[0]
>>> plt.plot(filtered_signal) # filtered_signal
```




---

onebone.signal.filter.**highpass_filter**(*signal: ndarray*, *fs: Union[int, float]*, *cutoff: Union[int, float]*, *order: int = 5*, *axis: int = -1*) → ndarray

1D Butterworth highpass filter.

**Parameters**

- **signal** (*numpy.ndarray*) – Original time-domain signal.
- **fs** (*Union[int, float]*) – Sampling rate.
- **cutoff** (*Union[int, float]*) – Cutoff frequency.
- **order** (*int, default=5*) – Order of butterworth filter.
- **axis** (*int, default=-1*) – The axis of the input data array along which to apply the linear filter.

**Returns**

**out** (*numpy.ndarray*) – Filtered signal.

---

**Examples**

Apply the filter to 1d signal. And then check the frequency component of the filtered signal.

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal.shape
(5000,)
>>> freq_x = np.fft.rfftfreq(signal.size, 1 / fs)[:-1]
>>> origin_fft_mag = abs((np.fft.rfft(signal) / signal.size)[:-1] * 2)
>>> origin_freq = freq_x[np.where(origin_fft_mag > 0.5)]
>>> origin_freq
[ 20., 100.]
>>> filtered_signal = highpass_filter(signal, fs, cutoff=50)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal) / signal.size)[:-1] * 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 100.]
```

Apply the filter to 2d signal (axis=0).

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal = np.stack([signal, signal]).T
>>> signal.shape
(5000, 2)
>>> filtered_signal = highpass_filter(signal, fs, cutoff=50, axis=0)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal[:, 0]) / signal.size)[:-1]␣
↪* 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 100.]
```

---

onebone.signal.filter.**lowpass_filter**(*signal: ndarray*, *fs: Union[int, float]*, *cutoff: Union[int, float]*, *order: int = 5*, *axis: int = -1*) → ndarray

1D Butterworth lowpass filter.

**Parameters**

- **signal** (*numpy.ndarray*) – Original time-domain signal.
- **fs** (*Union[int, float]*) – Sampling rate.
- **cutoff** (*Union[int, float]*) – Cutoff frequency.
- **order** (*int, default=5*) – Order of butterworth filter.
- **axis** (*int, default=-1*) – The axis of the input data array along which to apply the linear filter.

**Returns**

**out** (*numpy.ndarray*) – Filtered signal.

---

**Examples**

Apply the filter to 1d signal. And then check the frequency component of the filtered signal.

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal.shape
(5000,)
>>> freq_x = np.fft.rfftfreq(signal.size, 1 / fs)[:-1]
>>> origin_fft_mag = abs((np.fft.rfft(signal) / signal.size)[:-1] * 2)
>>> origin_freq = freq_x[np.where(origin_fft_mag > 0.5)]
>>> origin_freq
[ 20., 100.]
>>> filtered_signal = lowpass_filter(signal, fs, cutoff=50)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal) / signal.size)[:-1] * 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 20.]
```

Apply the filter to 2d signal (axis=0).

```
>>> fs = 5000.0
>>> t = np.linspace(0, 1, int(fs))
>>> signal = 10.0 * np.sin(2 * np.pi * 20.0 * t)
>>> signal += 5.0 * np.sin(2 * np.pi * 100.0 * t)
>>> signal = np.stack([signal, signal]).T
>>> signal.shape
(5000, 2)
>>> filtered_signal = lowpass_filter(signal, fs, cutoff=50, axis=0)
>>> filtered_fft_mag = abs((np.fft.rfft(filtered_signal[:, 0]) / signal.size)[:-1]␣
↪* 2)
>>> filtered_freq = freq_x[np.where(filtered_fft_mag > 0.5)]
>>> filtered_freq
[ 20.]
```

## onebone.signal.smoothing

A Moving Average (MA) which returns the weighted average of array.

- Author: Kibum Park
- Contact: kibum.park@onepredict.com

onebone.signal.smoothing.**moving_average**(*signal: ndarray*, *window_size: Union[int, float]*, *pad: bool = False*, *weights: Optional[ndarray] = None*, *axis: Union[int, float] = -1*) → ndarray

Weighted moving average. .. math:: WMA(x, w, t, n) = sum_{i=n-t+1}^{n} w_i x_i, where $x$ is the input array, $w_i$ is the weight of the $i$-th element, $t$ is the window size, $n$ is the $n$ is smaller than $t$, $i$ is set to 0.

**Parameters**

- **signal** (*numpy.ndarray of shape (signal_length,), (n, signal_length,)*) – Original time-domain signal.
- **window_size** (*Union[int, float], optional, default=10*) – Window size. One of *window_size*,`weights` must be specified.
- **pad** (*bool, default=False*) – Padding method. If True, Pads with the edge values of array is added. So the shape of output is same as *signal*.
- **weights** (*Union[numpy.ndarray of shape (window_size,), None], optional, default=None*) – Weighting coefficients. If None, the *weights* are uniform. One of *window_size*,`weights` must be specified.
- **axis** (*Union[int, float], optional, default=-1*) – The axis of the input data array along which to apply the moving average.

**Returns**

**ma** (*numpy.ndarray*) – Moving average signal. If input shape is [signal_length,], output shape is [signal_length,]. If input shape is [n, signal_length,] and axis is 1, output shape is [n, signal_length,]. If input shape is [signal_length, n] and axis is 0, output shape is [signal_length, n]. If pad is False, output shape is [signal_length - window_size + 1,]. If pad is True, output shape is [signal_length,].

**Examples**

```
>>> signal = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> window_size = 3
>>> moving_average(signal, window_size)
[2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> signal = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> window_size = 3
>>> moving_average(signal, window_size, pad=True)
[1, 1.5, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> signal = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> window_size = 3
>>> weights = np.array([1, 2, 3])
>>> moving_average(signal, window_size, weights=weights)
[2.33333333, 3.33333333, 4.33333333, 5.33333333, 6.33333333, 7.33333333, 8.33333333,
↪ 9.33333333]
```

---

### 2.1.5 onebone.utils

#### onebone.utils.slicing

onebone.utils.slicing.**slice_along_axis**(*arr: ndarray*, *s: slice*, *axis: int*) → ndarray

Slice the values of the array within a certain range on the axis.

**Parameters**

- **arr** (*numpy.ndarray*) – Input array.
- **s** (*slice*) – Range on the *axis*.
- **axis** (*int*) – Axis

**Returns**

**arr_out** (*numpy.ndarray*) – Sliced input array.

#### onebone.utils.timer

Timer function.

- Author: Kangwhi Kim
- Contact: kangwhi.kim@onepredict.com

**class** onebone.utils.timer.**Timer**(*logger: Optional[Logger] = None*)

Bases: object

Check the elapsed time of the function.

---

**Note:** Use it as a function decorator.

---

**Parameters**

**logger** (*logging.Logger, default=None*) – A logger.

**Returns**

**wrapper** (*function*) – Wrapper function. When *logger* is not *None*, the debug level message is delivered to the logger within the wrapper function. But, when *logger* is *None*, the message is delivered to the *print* function.

---

**Examples**

```
>>> import logging
>>> import time
>>> from onebone.utils import Timer
```

Create a logger.

```
>>> logger = logging.getLogger(__name__)
>>> logger.setLevel(logging.DEBUG)
>>> stream_handler = logging.StreamHandler()
>>> logger.addHandler(stream_handler)
```

Add the *Timer* Decorator to the function.

```
>>> @Timer(logger)
>>> def timer_test():
        start = time.time()
        time.sleep(1)
        duration = time.time() - start
        return duration
```

Run the function.

```
>>> timer_test()
```

---

CHAPTER

THREE

# CALL FOR CONTRIBUTE

We appreciate and welcome contributions. Small improvements or fixes are always appreciated; issues labeled as "good first issue" may be a good starting point.

Writing code isn't the only way to contribute to onebone. You can also:

- triage issues
- review pull requests
- help with outreach and onboard new contributors

If you're unsure where to start or how your skills fit in, reach out! You can ask here, on GitHub, by leaving a comment on a relevant issue that is already open.

If you want to use an code for signal analysis, but it's not in onebone, make a issue.

Please follow this guide to contribute to onebone.

# PYTHON MODULE INDEX

## O

# INDEX

## P



## R



## S



## T



## V



## W



## Z