pyrem Documentation

Release 1.0

Quentin Geissmann

September 07, 2014

CONTENTS

CHAPTER

ONE

MODULE CONTENTS

Pyrem is a python package to help feature extraction from physiological time series such as electroencephalograms(EEGs) and such.

It provides:

1. Data structures for time series (`pyrem.time_series`) based on numpy arrays. This extends functionality of numpy arrays by:
   (a) Providing extra attributes such as sampling frequency and metadata
   (b) Allow new functionality such as time-string indexing (e.g. `signal["28m":"2h22.4s"]`)
2. A data structure for annotations (`pyrem.time_series.Annotation`) based on numpy recarrays. This allows to describe time series of discrete states linked to a confidence/probability of observation of each states.
3. A data structure for collection of time series (`pyrem.polygram`). It features:
   (a) Support for heterogeneous sampling rates between time series.
   (b) An iterator through arbitrary epochs (temporal slices)
   (c) Slicing using time strings and seamless merging between polygrams
4. Implementations of algorithms often used in analysis of EEGs (see `pyrem.univariate`). They are essentially faster and curated reimplementation of the function available in PyEEG.
5. Utilities to load, save and visualize the data (still in development).
6. Wrappers around samplerate and pywt libraries, to efficiently resample, and compute discrete wavelet decomposition on signals.

CHAPTER

TWO

SUBMODULES

2.1 pyrem.time_series module

2.1.1 Biological time series

This module provides data structure for two types of time series: `Signal` and `Annotation`.

Both structures extend `ndarray` by providing attributes, such as *sampling frequency*, *metadata*, name. More importantly, time series provide specific features such as indexing with time strings.

```
>>> import pyrem as pr
>>> import numpy as np
>>> # generate white noise:
>>> noise = np.random.normal(size=int(1e6))
>>> # a million point sampled at 256 Hz
```

Then, to create a time series from this vector of random numbers:

```
>>> sig = pr.time_series.Signal(noise, 256.0,
>>>                             type="noise", name="channel_1",
>>>                             metadata={"patient": "John_Doe"})
```

Display information about this time series:

```
>>> sig
```

To resample at 100 Hz:

```
>>> sig_short =  sig.resample(100.0)
>>> sig_short
```

Time series derive from numpy arrays, so you can just use them as such:

```
>>> sig_norm = sig - np.mean(sig)
Note that the resulting signal is conveniently a time series (not a regular numpy array)

>>> np.diff(sig)
```

2.1.2 Indexing time series

As numpy arrays

Since time series are derived from numpy array, the numpy indexing rule apply:

```
>>> sig[1:1000:3] # one to 999, every 3 values
>>> sig[: -100] # from start to 0 to 100 before the end
```

See numpy documentation for more info.

With strings and time-deltas

It is common to have to extract a signal between two different time points. Instead of having to tediously calculate index from time, *pyrem* offers the possibility to use stings and `datetime.timedelta`

Time strings are represented with the following format:

"29h33m1.02s"

Where:

- h is for hour
- m for minutes
- s for seconds

Example:

```
>>> # string indexing:
>>> print sig.duration
>>> sig2 = sig["1h2m2s":]  # everything after 1 hour, 2 min and 2 seconds
>>> print sig2.duration
>>> # this should be exactly 1h2m2s
>>> print sig.duration - sig2.duration
>>> print sig["1h2m2s":"1h2m2.1s"]
```

Note: When indexing a signal with time strings, we query the values of a *discrete representation of a continuous signal*. Therefore, it makes no sense to obtain a signal of length zero. For instance, imagine a signal of 10 seconds sampled at 1Hz. If we query the value between 1.5 and 1.6s, no points fall in this interval. However, the signal does have a value. In this case, *pyrem* returns a signal of length 1 where the unique value is the value of the former neighbour.

```
>>> sig = pr.time_series.Signal([3,4,2,6,4,7,4,5,7,9], 10.0,)
>>> sig["0s":"0.001s"])
>>> sig["0s":"0.011s"])
```

Epoch iteration

A common task is to extract successive temporal slices (i.e. epochs) of a signal, for instance, in order to compute features. `iter_window()` iterator facilitates this.

Let us work wit a one minute signal as an example:

```
>>> sig1m = sig[:"1m"]
```

Get every 5 seconds of a the first minutes of a signal:

```
>>> for time, sub_signal in sig1m.iter_window(5,1.0):
>>>     print time, sub_signal.duration
```

Get 10 second epochs, overlapping of 50% (5s):

```
>>> for time, sub_signal in sig1m.iter_window(10,0.5):
>>>     print time, sub_signal.duration
```

Get 1 second epochs, skipping every other epoch

```
>>> for time, sub_signal in sig1m.iter_window(1,2.0):
>>>     print time, sub_signal.duration
```

class pyrem.time_series.**Annotation**(*data*, *fs*, *observation_probabilities=None*, ***kwargs*)

Bases: pyrem.time_series.BiologicalTimeSeries

Annotations are time series of discrete values associated with a probability/confidence of observing this value. BiologicalTimeSeries indexing rules apply to them.

Parameters

- **data** – a vector representing different states. It should be a uint8 one-d array like structure (typically, a ndarray)
- **fs** (*float*) – the sampling frequency
- **observation_probabilities** – an array of confidence/ probability of observation of the states. it should be a one-d array-like of floats of length *len(data)*. If *None*, confidence are assumed to be equal to one for all observations.
- **kwargs** – key word arguments to be passed to BiologicalTimeSeries

probas

The probabilities/confidences associated to the annotation values.

Returns an array of float32

Return type ndarray

resample(*target_fs*)

Resample annotations to a new sampling frequency. Values are resampled with nearest neighbour interpolation, while associated probabilities are linearly interpolated.

Parameters **target_fs** (*float*) – The target sampling frequency

Returns a new Annotation object

values

The values of each annotations.

Returns an array of uint8

Return type ndarray

class pyrem.time_series.**BiologicalTimeSeries**(*data*, *fs*, *type=None*, *name=None*, *metadata=None*)

Bases: numpy.ndarray

An abstract class for time series.

Parameters

- **data** – an one-d array like structure (typically, a ndarray)
- **fs** (*float*) – the sampling frequency
- **type** (*str*) – the type of time series (e.g. “eeg”, “temperature”, “blood_pH”)
- **name** (*str*) – a unique name to identify a time series contained in a Polygram
- **metadata** (*dict*) – a dictionary of additional information (e.g. experimental variables)

copy()
Deep copy a time series.

Returns A new time series with identical values and attributes

Return type BiologicalTimeSeries

duration

Returns the total duration of the time series

Return type datetime

fs

Returns the sampling frequency of the time series

Return type float

iter_window(*length*, *lag*)
Iterate through an array by successive (possibly overlapping) slices (i.e. epochs). Conveniently, the central time ot the epoch is also returned.

Parameters

- **lag** (*float*) – the ratio of overlap (1 = no overlap, 0 = completely overlapped, 2 = skip every other epoch)
- **length** (*float*) – duration of the epochs (in second)

Returns (centre_of_window, BiologicalTimeSeries)

metadata

Returns a dictionary of metadata (i.e. information about data acquisition)

Return type dict

name
The name of the signal. It is expected to be unique.

Returns the user-defined name for this signal

Return type str

rename(*name*)
Rename the signal

Parameters **name** (*str*) – the new name

resample(*new_fs*)
Abstract method for resampling a time series (behaves differently according to the type of time series)

Note: Because time series are digital (i.e. discrete) the resulting sampling rate is expected to be slightly different from the target sampling rate.

Parameters **new_fs** (*float*) – the target time series

Returns a new BiologicalTimeSeries

save(*filename*, *compression_level=5*)
Efficiently save a time series using joblib

Parameters

- **filename** (*str*) – the output file name

- **compression_level** (*int*) – an integer between 1 and 9. More is better, but slower. 5 is generally a good compromise

type

Returns the user-defined type of time series (e.g. "eeg" or "ecg")

Return type str

class pyrem.time_series.**Signal**(*data*, *fs*, ***kwargs*)

Bases: pyrem.time_series.BiologicalTimeSeries

resample(*target_fs*, *mode='sinc_best'*)

Resample the signal. One implication of the signal being digital, is that the resulting sampling frequency is not guaranteed to be exactly at *target_fs*. This method wraps `resample()`

Parameters

- **target_fs** (*float*) – The new sampling frequency
- **mode** (*str*) – to be passed to `resample()`

Returns

2.2 pyrem.polygram module

2.2.1 Polygram

This module provides `Polygram`; a container for biological time series such as `Signal` and `Annotation`. In this respect, it is inspired from pandas `TimeSeries` and `DataFrame`. You can think about it as a dataframe where each column is a signal, or an annotation, and each row a time point.

The originality of :~*pyrem.polygram.Polygram* is to be able to deal with **heterogeneous (between signals) sampling rates**. It contains time series with the same approximate duration, but different number of points. This is typical when dealing with physiological time series because different variable will be recorded at different sampling rates (see for instance, the [EDF] data format). Another situation it which this could be useful, is when performing a wavelet decomposition of a signal. Indeed, one would obtain a set of time series (coefficients) of the same duration, but with different sampling rates (i.e. $fs_{D_N} = 2fs_{D_{N+1}}$).

Systematically resampling signals, and annotations, to the maximal sampling rate is not trivial, and would impact significantly computational efficiency.

Constructing a Polygram

First, let us create a couple of `BiolgicalTimeSeries`:

```
>>> import numpy as np
>>> from pyrem.time_series import Annotation, Signal
>>> from pyrem.polygram import Polygram
>>>
>>> # create an Annotation with 1000 random values, sampled at 1.0Hz
>>> probs = np.random.random_sample(1000)
>>> vals = (np.random.random_sample(1000) * 4 +1).astype(np.int)
>>> annot = Annotation(vals,fs=1.0, observation_probabilities=probs, type="vigilance_state", name="st
>>>
>>> # now a random walk signal of 100000 points at 100.0Hz
>>> rw = np.cumsum(np.random.normal(0,1,100000))
>>> sig = Signal(rw, fs=100.0,type="eeg", name="eeg1")
```

```
>>>
>>> # Once we have our time series, we can just do:
>>> pol = Polygram([annot, sig])
>>> #printing the object shows the characteristic of each channels
>>> pol
Polygram
-----
Duration:       0:16:40 (HH:mm:ss)
N signals:      1
N annotations:  1
Metadata:
                None
----
Channel information:
    Name             Type fs(Hz) Duration
0   eeg1              eeg  100.0  0:16:40
1  state  vigilance_state    1.0  0:16:40
```

Note: Slightly different durations are allowed

The constructor will raise an error if the provided channels do not have the same duration:

```
>>> Polygram([annot[:"11m"], sig[:"10m"]])
ValueError
'Channels must have approximately the same length.
The durations of the input channels are:['0:10:00', '0:11:00']'
```

However, in practice, it is almost impossible to obtain discrete signal of the exact same duration. Imagine, for instance that you have a first signal of 14 points at 3Hz (~ 4.667s), and a second signal of 5 points at 1Hz (5.0s). In this case, it is impossible to have exactly 14/3s of signal form a 1Hz signal. This could be represented by:

```
>>> 0123456789abcd-   #  3Hz => one symbol/point
>>> AAABBBCCCDDDEEE   # 1Hz => one LETTER/point
>>> AAABBBCCCDDD---   # 1Hz => one LETTER/point
```

Here, neither the second nor the third signal match, exactly, the duration of the first, but bot are approximately the same duration as the first.

A Polygram will tolerate this sort of mismatch if and only if all pairs of channels are within one period of the time series with the channel longest period.

Accessing channels

Often, you will want to extract a channel by name:

```
>>> pol.channel_names
['eeg1', 'state']
>>> pol['eeg1']
Signal
----
Name:   eeg1
Duration:       0:16:40 (HH:mm:ss)
Sampling freq:  100.000000 Hz
Type:   eeg
N points:       100000
Metadata:
                None
```

```
>>> # this is equivalent to
>>> pol[0]
```

You can also iterate through channels:

```
>>> [c.size for c in pol.channels]
[100000, 1000]
```

With strings and time-deltas

Because time series are potentially at different sampling rates, it makes no sense to index a polygram by range of integers:

```
>>> #does NOT work
>>> # pol[10:20]
```

Instead, time string and `datetime.timedelta` can be used for extracting a sub_polygram:

```
>>> pol["1m":"2m"]
```

Indexing rules are similar to `time_series`

Note: Indexing does NOT deep copy

When getting an epoch (temporal slice), of a polygram, the channel in the new polygram are *views* to the underlying data of the original channel. Like for numpy arrays, *modifying the data in a sub-polygram will modify the parent polygram*. To avoid this behaviour, one can call `copy()`

Epoch iteration

If you want to extract features for each epoch and each channel, you may want to use th `iter_window()` iterator. It works like the `iter_window()`

class `pyrem.polygram.`**`Polygram`**(*channels*, *metadata=None*)

Bases: `object`

Parameters

- **channels** (list(`BiolgicalTimeSeries`)) – a list of time series with approximately the same duration
- **metadata** (*dict*) – a dictionary of additional information (e.g. experimental variables)

`annotation_channels`

An iterator through the all the *annotation* channels

`channel_names`

Returns The list of channel names

Return type list(str)

`channel_types`

Returns the types of all channels

Return type list(str)

`channels`

An iterator through the all the channels

copy()
Deep copy of an Polygram

Returns a new Polygram with the same values

Return type Polygram

duration

Returns The duration total of the polygram. That is the duration of the channel with the longest duration

Return type datetime.timedelta

iter_window(*length*, *lag*)

map_signal_channels(*fun*)
Applies a function to all signal channels and returns a new Polygram with modified channels

An example of how to normalise all signal channels

```
>>> pol_norm =  pol.map_signal_channels(
>>>          lambda x: (x - np.mean(x))/np.std(x))
>>> np.mean(pol[0])
>>> np.mean(pol_norm[0])
```

Parameters **fun** (*callable*) – a function to be applied

Returns a new polygram

Return type Polygram

merge(*obj*, *trim_channel=True*)
Adds channels from a polygram to another polygram, or append a time series to a polygram

Parameters

- **obj** (Polygram or BiologicalTimeSeries) – either a polygram or a time series to be added
- **trim_channel** (*bool*) – whether the new channel(s), if they have a longer duration, would be shortened to match the existing polygram.

metadata

Returns the metadata of this polygram

Return type dict

n_annotations

Returns The total number of *annotation* channels

Return type int

n_channels

Returns The total number of channels

Return type int

n_signals

Returns The total number of *signal* channels

Return type int

save (*filename*, *compression_level=5*)
Efficiently save a Polygram using joblib

Parameters

- **filename** (*str*) – the output file name
- **compression_level** (*int*) – an integer between 1 and 9. More is better, but slower. 5 is generally a good compromise

show ()
Interactively displays a polygram using matplotlib. **Very urresponsive and prototypical at the minute**

signal_channels
An iterator through the all the *signal* channels

2.3 pyrem.univariate module

2.3.1 Feature computation for univariate time series

This sub-module provides routines for computing features on univariate time series. Many functions are improved version of PyEEG [PYEEG] functions. Be careful, some functions will give different results compared to PyEEG as the maths have been changed to match original definitions. Have a look at the documentation notes/ source code to know more.

Here a list of the functions that were reimplemented:

- Approximate entropy `ap_entropy()` [RIC00]
- Fisher information `fisher_info()` [PYEEG]
- Higuchi fractal dimension `hfd()` [HIG88]
- Hjorth parameters `hjorth()` [HJO70]
- Petrosian fractal dimension `pfd()` [PET95]
- Sample entropy `samp_entropy()` [RIC00]
- Singular value decomposition entropy `svd_entropy()` [PYEEG]
- Spectral entropy `spectral_entropy()` [PYEEG]

pyrem.univariate.**ap_entropy** (*a*, *m*, *R*)
Compute the approximate entropy of a signal with embedding dimension "de" and delay "tau" [PYEEG]. Vectorised version of the PyEEG function. Faster than PyEEG, but still critically slow.

Parameters

- **a** (`ndarray` or `Signal`) – a one dimensional floating-point array representing a time series.
- **m** (*int*) – the scale
- **R** (*float'*) – The tolerance

Returns the approximate entropy, a scalar

Return type float

pyrem.univariate.**dfa** (*X*, *Ave=None*, *L=None*, *sampling=1*)
WIP on this function. It is basically copied and pasted from [PYEEG], without verification of the maths or unittests.

pyrem.univariate.**fisher_info**(*a*, *tau*, *de*)

Compute the Fisher information of a signal with embedding dimension "de" and delay "tau" [PYEEG]. Vectorised (i.e. faster) version of the eponymous PyEEG function.

Parameters

- **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.
- **tau** (*int*) – the delay
- **de** (*int*) – the embedding dimension

Returns the Fisher information, a scalar

Return type float

pyrem.univariate.**hfd**(*a*, *k_max*)

Compute Higuchi Fractal Dimension of a time series. Vectorised version of the eponymous [PYEEG] function.

Note: Difference with PyEEG:

Results is different from [PYEEG] which appears to have implemented an erroneous formulae. [HIG88] defines the normalisation factor as:

$$\frac{N-1}{[\frac{N-m}{k}]k}$$

[PYEEG] implementation uses:

$$\frac{N-1}{[\frac{N-m}{k}]}$$

The latter does *not* give the expected fractal dimension of approximately *1.50* for brownian motion (see example bellow).

Parameters

- **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.
- **k_max** (*int*) – the maximal value of k

Returns Higuchi's fractal dimension; a scalar

Return type float

Example from [HIG88]. This should produce a result close to *1.50*:

```
>>> import numpy as np
>>> import pyrem as pr
>>> i = np.arange(2 ** 15) +1001
>>> z = np.random.normal(size=int(2 ** 15) + 1001)
>>> y = np.array([np.sum(z[1:j]) for j in i])
>>> pr.univariate.hfd(y,2**8)
```

pyrem.univariate.**hjorth**(*a*)

Compute Hjorth parameters [HJO70].

$$Activity = m_0 = \sigma_a^2$$

$$Complexity = m_2 = \sigma_d/\sigma_a$$

$$Morbidity = m_4 = \frac{\sigma_{dd}/\sigma_d}{m_2}$$

Where:

σ_x^2 is the mean power of a signal x. That is, its variance, if it's mean is zero.

a, d and dd represent the original signal, its first and second derivatives, respectively.

Note: Difference with PyEEG:

Results is different from [PYEEG] which appear to uses a non normalised (by the length of the signal) definition of the activity:

$$\sigma_a^2 = \sum \mathbf{x}[i]^2$$

As opposed to

$$\sigma_a^2 = \frac{1}{n} \sum \mathbf{x}[i]^2$$

Parameters **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.

Returns activity, complexity and morbidity

Return type tuple(float, float, float)

Example:

```
>>> import pyrem as pr
>>> import numpy as np
>>> # generate white noise:
>>> noise = np.random.normal(size=int(1e4))
>>> activity, complexity, morbidity = pr.univariate.hjorth(noise)
```

pyrem.univariate.**hurst**(*signal*)

Experimental/untested implementation taken from: http://drtomstarke.com/index.php/calculation-of-the-hurst-exponent-to-test-for-trend-and-mean-reversion/

Use at your own risks.

pyrem.univariate.**pfd**(*a*)

Compute Petrosian Fractal Dimension of a time series [PET95].

It is defined by:

$$\frac{log(N)}{log(N) + log(\frac{N}{N+0.4N_\delta})}$$

Note: Difference with PyEEG:

Results is different from [PYEEG] which implemented an apparently erroneous formulae:

$$\frac{log(N)}{log(N) + log(\frac{N}{N} + 0.4N_\delta)}$$

Where:

N is the length of the time series, and

N_δ is the number of sign changes.

Parameters **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.

Returns the Petrosian Fractal Dimension; a scalar.

Return type float

Example:

```
>>> import pyrem as pr
>>> import numpy as np
>>> # generate white noise:
>>> noise = np.random.normal(size=int(1e4))
>>> pr.univariate.pdf(noise)
```

pyrem.univariate.**samp_entropy**(*a*, *m*, *r*, *tau=1*, *relative_r=True*)

Compute the sample entropy [RIC00] of a signal with embedding dimension *de* and delay *tau* [PYEEG]. Vectorised version of the eponymous PyEEG function. In addition, this function can also be used to vary tau and therefore compute Multi-Scale Entropy(MSE) [COS05] by coarse grainning the time series (see example bellow). By default, r is expressed as relatively to the standard deviation of the signal.

Parameters

- **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.
- **m** (*int*) – the scale
- **r** (*float*) – The tolerance
- **tau** (*int*) – The scale for coarse grainning.
- **relative_r** (*true*) – whether the argument r is relative to the standard deviation. If false, an absolute value should be given for r.

Returns the approximate entropy, a scalar

Return type float

Example:

```
>>> import pyrem as pr
>>> import numpy as np
>>> # generate white noise:
>>> noise = np.random.normal(size=int(1e4))
>>> pr.univariate.samp_entropy(noise, m=2, r=1.5)
>>> # now we can do that for multiple scales (MSE):
>>> [pr.univariate.samp_entropy(noise, m=2, r=1.5, tau=tau) for tau in range(1, 5)]
```

pyrem.univariate.**spectral_entropy**(*a*, *sampling_freq*, *bands=None*)

Compute spectral entropy of a signal with respect to frequency bands. The power spectrum is computed through fft. Then, it is normalised and assimilated to a probability density function. The entropy of the signal x can be expressed by:

$$H(x) = -\sum_{f=0}^{f=f_s/2} PSD(f) log_2[PSD(f)]$$

Where:

PSD is the normalised power spectrum (Power Spectrum Density), and

f_s is the sampling frequency

Parameters

- **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.
- **sampling_freq** (*float*) – the sampling frequency
- **bands** – a list of numbers delimiting the bins of the frequency bands. If None the entropy is computed over the whole range of the DFT (from 0 to $f_s/2$)

Returns the spectral entropy; a scalar

pyrem.univariate.**svd_entropy**(*a*, *tau*, *de*)

Compute the Singular Value Decomposition entropy of a signal with embedding dimension "de" and delay "tau" [PYEEG].

Note: Difference with PyEEG:

The result differs from PyEEG implementation because log_2 is used (as opposed to natural logarithm in PyEEG code), according to the definition in their paper [PYEEG] (eq. 9):

$$H_{SVD} = -\sum \bar{\sigma}_i log_2 \bar{\sigma}_i$$

Parameters

- **a** (ndarray or Signal) – a one dimensional floating-point array representing a time series.
- **tau** (*int*) – the delay
- **de** (*int*) – the embedding dimension

Returns the SVD entropy, a scalar

Return type float

2.4 pyrem.visualization module

This is prototypical early visualisation module. It allows visualisation of Polygram objects using matplotlib. In the future, The visualisation tool should be faster and more interactive. It is likely to become independent of matplotlib.

class pyrem.visualization.**PolygramDisplay**(*polygram*, *max_point_amplitude_plot=1000*)

Bases: object

show()

2.5 pyrem.wavelet_decomposition module

pyrem.wavelet_decomposition.**decompose_signal**(*signal*, *levels=(1, 2, 3, 4, 5)*, *wavelet='db4'*, *resample_before=None*, *mode='per'*, *keep_a=True*)

A wrapper around wavedec(). It performs discrete wavelet decomposition and return the coefficients as a

pyrem.polygram.Polygram. It allows to select the wavelet coefficients to keep and can perform preliminary resampling of the signal. In the resulting polygram, the names of the coefficients will be suffixed by an an identifier describing their respective levels (i.e. cD_1, cD_2, ..., cD_N, cA_N). The sampling frequency of coefficients will also be automatically computed.

```
>>> import numpy as np
>>> import pyrem as pr

>>> noise = np.random.normal(size=int(1e6))
>>> sig = pr.time_series.Signal(noise, 256.0,
>>>      name="channel_1")
>>>      name="channel_1")
>>> pol = pr.wavelet_decomposition decompose_signal(sig)
>>> pol
    Polygram
    ----
    Duration:       1:05:06.250000 (HH:mm:ss)
    N signals:      6
    N annotations:  0
    Metadata:
            None
    ----
    Channel information:
                 Name  Type fs(Hz)         Duration
    0  channel_1_cA_5  None    8.0  1:05:06.250000
    1  channel_1_cD_1  None  128.0  1:05:06.250000
    2  channel_1_cD_2  None   64.0  1:05:06.250000
    3  channel_1_cD_3  None   32.0  1:05:06.250000
    4  channel_1_cD_4  None   16.0  1:05:06.250000
    5  channel_1_cD_5  None    8.0  1:05:06.250000
```

Parameters

- **signal** (pyrem.time_series.Signal) – the time series to be decomposed
- **levels** (*list([int])*) – the levels to keep (e.g. [1,2,3,5]) will not return the coeficient cD_4
- **wavelet** (*str*) – the type of wavelet (see wavedec())
- **resample_before** (*float*) – the sampling frequency at which to resample the signal before the decomposition
- **mode** (*str*) – (see wavedec())
- **keep_a** – whether to keep the last coefficient (cA_N)

Returns A polygram with all the requested coefficients

Return type pyrem.polygram.Polygram

2.6 pyrem.feature_families module

The goal of this submodule is to provide a flexible interface to compute arbitrary features on each channel and epoch (temporal slices) of a multivariate time series (Polygraph). Features are grouped in families of several features (e.g. Power Features may contain mean power, variance of power, ...). Feature factory computes features for arbitrary feature families and group them in a data.frame

class pyrem.feature_families.**AbsoluteFeatures**

Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'abs'

class pyrem.feature_families.**AnnotationFeatureBase**
Bases: pyrem.feature_families.FeatureFamilyBase

class pyrem.feature_families.**EntropyFeatures**
Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'entropy'

class pyrem.feature_families.**FeatureFamilyBase**
Bases: object

A feature family object is a process returning a vector of features upon analysis of some data. Features are returned as a pandas DataFrame object, with column names for features. Each feature name is prefixed by the name of the feature family. This is an abstract class designed to be derived by:

1. Defining a `prefix` attribute. It will add the name of the family to the name of the features.
2. Overriding the `_make_feature_vec` method. It should return a dictionary of scalars, each being a feature.

make_vector(*signal*)
Compute one vector of features from polygraph.

Parameters **data** (Polygraph) – A signal

Returns a one-row dataframe

Return type DataFrame

prefix = None

class pyrem.feature_families.**FractalFeatures**
Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'fractal'

class pyrem.feature_families.**HjorthFeatures**
Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'hjorth'

class pyrem.feature_families.**NonLinearFeatures**
Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'nl'

class pyrem.feature_families.**PeriodogramFeatures**
Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'spectr'

class pyrem.feature_families.**PowerFeatures**
Bases: pyrem.feature_families.SignalFeatureBase

prefix = 'power'

class pyrem.feature_families.**SignalFeatureBase**
Bases: pyrem.feature_families.FeatureFamilyBase

class pyrem.feature_families.**VigilState**
Bases: pyrem.feature_families.AnnotationFeatureBase

prefix = 'vigil'

2.7 pyrem.io module

pyrem.io.**polygram_from_pkl**(*filename*)

pyrem.io.**polygram_from_spike_matlab_file**(*signal_filename*, *annotation_filename*, *fs*, *annotation_fs*, *channel_names*, *channel_types*, *doubt_chars*, *resample_signals*, *metadata={}*)

This function loads a matlab file exported by spike to as a polygraph.

Parameters **signal_filename** – the matlab file name

Returns a polygram

pyrem.io.**signal_from_pkl**(*filename*)

2.8 pyrem.utils module

pyrem.utils.**str_to_time**(*str*)

Parse a string describing a duration as a timedelta

Parameters **str** (*str*) – a string with the format "XhYmZs". where XYZ are integers (or floats)

Returns a timedelta corresponding to the *str* argument

Return type timedelta

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

[EDF] B. Kemp and J. Olivan, “European data format ‘plus’ (EDF+), an EDF alike standard format for the exchange of physiological data,” Clinical Neurophysiology, vol. 114, no. 9, pp. 1755-1761, Sep. 2003.

[PET95] A. Petrosian, Kolmogorov complexity of finite sequences and recognition of different preictal EEG patterns, in , Proceedings of the Eighth IEEE Symposium on Computer-Based Medical Systems, 1995, 1995, pp. 212-217.

[PYEEG] F. S. Bao, X. Liu, and C. Zhang, PyEEG: An Open Source Python Module for EEG/MEG Feature Extraction, Computational Intelligence and Neuroscience, vol. 2011, p. e406391, Mar. 2011.

[HJO70] B. Hjorth, EEG analysis based on time domain properties, Electroencephalography and Clinical Neurophysiology, vol. 29, no. 3, pp. 306-310, Sep. 1970.

[COS05] 13. Costa, A. L. Goldberger, and C.-K. Peng, “Multiscale entropy analysis of biological signals,” Phys. Rev. E, vol. 71, no. 2, p. 021906, Feb. 2005.

[RIC00] J. S. Richman and J. R. Moorman, “Physiological time-series analysis using approximate entropy and sample entropy,” American Journal of Physiology - Heart and Circulatory Physiology, vol. 278, no. 6, pp. H2039-H2049, Jun. 2000.

[HIG88] 20. Higuchi, “Approach to an irregular time series on the basis of the fractal theory,” Physica D: Nonlinear Phenomena, vol. 31, no. 2, pp. 277-283, Jun. 1988.

p

R

S

T

V