PyEMD Documentation

Release 0.2.13

Dawid Laszuk

Apr 24, 2021

Table of Content

Writing documentation is hard. If more clarifications are needed, or you think others might benefit from extra explanation, don't hesitate to contact me through contact page.

CHAPTER 1

Intro

1.1 General

PyEMD is a Python implementation of Empirical Mode Decomposition (EMD) and its variations. One of the most popular expansion is Ensemble Empirical Mode Decomposition (EEMD), which utilises an ensemble of noise-assisted executions.

As the name suggests, methods in this package take data (signal) and decompose it into a set of component. All these methods theoretically should decompose a signal into the same set of components but in practise there are plenty of nuances and different ways to handle noise. Regardless of the method, obtained components are often called *Intrinsic Mode Functions* (IMF) to highlight that they contain an intrinsic (self) property which is a specific oscillation (mode). These are generic oscillations; their frequency and amplitude can change, however, no they are distinct within analyzed signal.

1.2 Installation

1.2.1 Simplest (pip)

Using *pip* to install is the quickest way to try and play. The package has had plenty of time to mature and at this point there aren't that many changes, especially nothing breaking. In the end, the basic EMD is the same as it was published in 1998.

The easiest way is to either add EMD-signal to your *requirements.txt* file, or use the command line:

```
$ pip install EMD-signal
```

And, yes, the package is updated every time there is a new algorithm or other cools features added.

1.2.2 Research (github)

Do you want to see the code by yourself? Update it? Make it better? Or worse (no judgement)? Then you likely want to check out the package and install it manually. **Don't worry, installation is simple**.

PyEMD is an open source project hosted on the GitHub on the main author's account, i.e. https://github.com/laszukdawid/PyEMD. This github page is where all changes are done first and where all issues should be reported. The page should have clear instructions on how to download the code. Currently that's a (only) green button and then following options.

In case you like using command line and want a copy-paste line

```
$ git clone https://github.com/laszukdawid/PyEMD
```

Once the code is download, enter package's directory and execute

```
$ python setup.py install
```

This will download all required dependencies and will install PyEMD in your environment. Once it's done do a sanity check with quick import and version print:

```
$ python -c "import PyEMD; print(PyEMD.__version__)"
```

It should print out some value concluding that you're good to go. In case of troubles, don't hesitate to submit an issue ticket via the link provided a bit earlier.

CHAPTER 2

Usage

2.1 Typical Usage

Majority, if not all, methods follow the same usage pattern:

- Import method
- Initiate method
- Apply method on data

On vanilla EMD this is as

```
from PyEMD import EMD
emd = EMD()
imfs = emd(s)
```

2.2 Parameters

The decomposition can be changed by adjusting parameters related to either sifting or stopping conditions.

2.2.1 Sifting

The sifting depends on the used method so these parameters ought to be looked within the methods. However, the typical parameters relate to spline method or the number of mirroring points.

2.2.2 Stopping conditions

All methods have the same two conditions, *FIXE* and *FIXE_H*, for stopping which relate to the number of sifting iterations. Setting parameter *FIXE* to any positive value will fix the number of iterations for each IMF to be exactly *FIXE*.

Example:

```
emd = EMD()
emd.FIXE = 10
imfs = emd(s)
```

Parameter *FIXE_H* relates to the number of iterations when the proto-IMF signal fulfils IMF conditions, i.e. number of extrema and zero-crossings differ at most by one and the mean is close to zero. This means that there will be at least *FIXE_H* iteration per IMF.

Example:

```
emd = EMD()
emd.FIXE_H = 5
imfs = emd(s)
```

When both *FIXE* and *FIXE_H* are 0 then other conditions are checked. These can be checking for convergence between consecutive iterations or whether the amplitude of output is below acceptable range.

CHAPTER 3

Speedup tricks

EMD is inherently slow with little chances on improving its performance. This is mainly due to it being a serial method. That's both on within IMF stage, i.e. iterative sifting, or between IMFs, i.e. the next IMF depends on the previous. On top of that, the common configuration of the EMD uses the natural cubic spline to span envelopes, which in turn additionally decreases performance since it depends on all extrema in the signal.

Since the EMD is the basis for other methods like EEMD and CEEMDAN these will also suffer from the same problem. What's more, these two methods perform the EMD many (hundreds) times which significantly increases any imperfections. It is expected that when it'll take more than a minute to perform an EEMD/CEEMDAN with default settings on a 10k+ samples long signal with a "medium complexity". There are, however, a couple of tweaks one can do to do make the computation finish sooner.

Sections below describe a tweaks one can do to improve performance of the EMD. In short, these changes are:

- *Change data type* (downscale)
- *Change spline method* to piecewise
- *Decrease number of trials*
- *Limit numer of output IMFs*

3.1 Change data type

Many programming frameworks by default casts numerical values to the largest data type it has. In case of Python's Numpy that's going to be numpy.float64. It's unlikely that one needs such resolution when using EMD*[0]. A suggestion is to downcast your data, e.g. to float16. The PyEMD should handle the same data type without upcasting but it can be additionally enforce a specific data type. To enable data type enforcement one needs to pass the DTYPE, i.e.

[0] I, the PyEMD's author, will go even a bit further. If one needs such large resolution then the EMD is not suitable for them. The EMD is not robust. Hundreds of iterations make any small difference to be emphasised and potentially leading to a significant change in final decomposition. This is the reason for creating EEMD and CEEMDAN which add small perturbation in a hope that the ensemble provides a robust solution.

```
from PyEMD import EMD

emd = EMD(DTYPE=np.float16)
```

3.2 Change spline method

EMD was presented with the natural cubic spline method to span envelopes and that's the default option in the PyEMD. It's great for signals with not many extrema but its not suggested for longer/more complex signals. The suggestion is to change the spline method to some piecewise splines like 'Akima' or 'piecewise cubic'.

Example:

```
from PyEMD import EEMD

eemd = EEMD(spline_kind='akima')
```

3.3 Decrease number of trials

This relates more to EEMD and CEEMDAN since they perform an EMD a multiple times with slightly modified signal. It's difficult to choose a correct number of iterations. This definitely relates to the signal in question. The more iterations the more certain that the solution is convergent but there is likely a point beyond which more evaluations change little. On the other side, the quicker we can get output the quicker we can use it.

In the PyEMD, the number of iterations is referred to by *trials* and it's an explicit parameter to EEMD and CEEMDAN. The default value was selected arbitrarily and it's most likely wrong. An example on updating it:

```
from PyEMD import CEEMDAN

ceemdan = CEEMAN(trials=20)
```

3.4 Limit numer of output IMFs

Each method, by default, will perform decomposition until all components are returned. However, many use cases only require the first component. One can limit the number of returned components by setting up an implicit variable *max_imf* to the desired value.

Example:

```
from PyEMD import EEMD

eemd = EEMD(max_imfs=2)
```

CHAPTER 4

Example

Some examples can be found in PyEMD/example directory.

4.1 EMD

4.1.1 Quick start

In most cases default settings are enough. Simply import `EMD` and pass your signal to *emd* method.

```
from PyEMD import EMD

s = np.random.random(100)
emd = EMD()
IMFs = emd.emd(s)
```

Something more

Here is a complete script on how to create and plot results.

```
from PyEMD import EMD
import numpy  as np
import pylab as plt

# Define signal
t = np.linspace(0, 1, 200)
s = np.cos(11*2*np.pi*t*t) + 6*t*t

# Execute EMD on signal
IMF = EMD().emd(s,t)
N = IMF.shape[0]+1
```

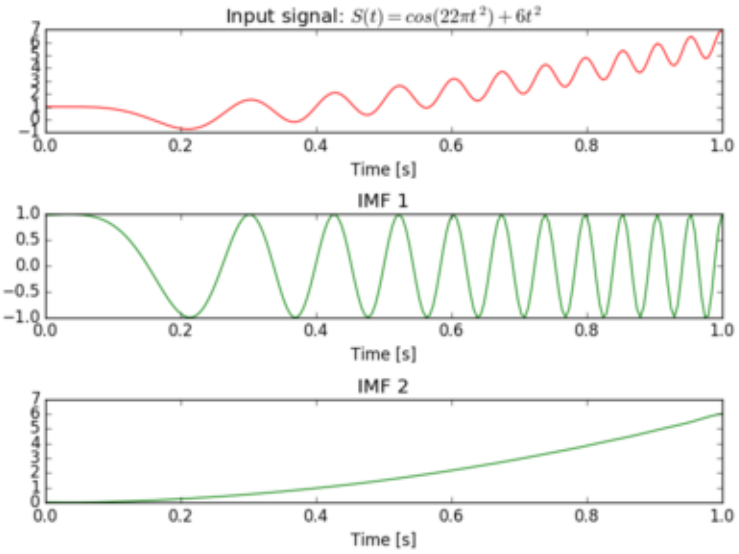
(continues on next page)

(continued from previous page)

```
# Plot results
plt.subplot(N,1,1)
plt.plot(t, s, 'r')
plt.title("Input signal: $S(t)=cos(22\pi t^2) + 6t^2$")
plt.xlabel("Time [s]")

for n, imf in enumerate(IMF):
    plt.subplot(N,1,n+2)
    plt.plot(t, imf, 'g')
    plt.title("IMF "+str(n+1))
    plt.xlabel("Time [s]")

plt.tight_layout()
plt.savefig('simple_example')
plt.show()
```

The Figure below was produced with input:

$S(t) = cos(22\pi t^2) + 6t^2$



4.2 EEMD

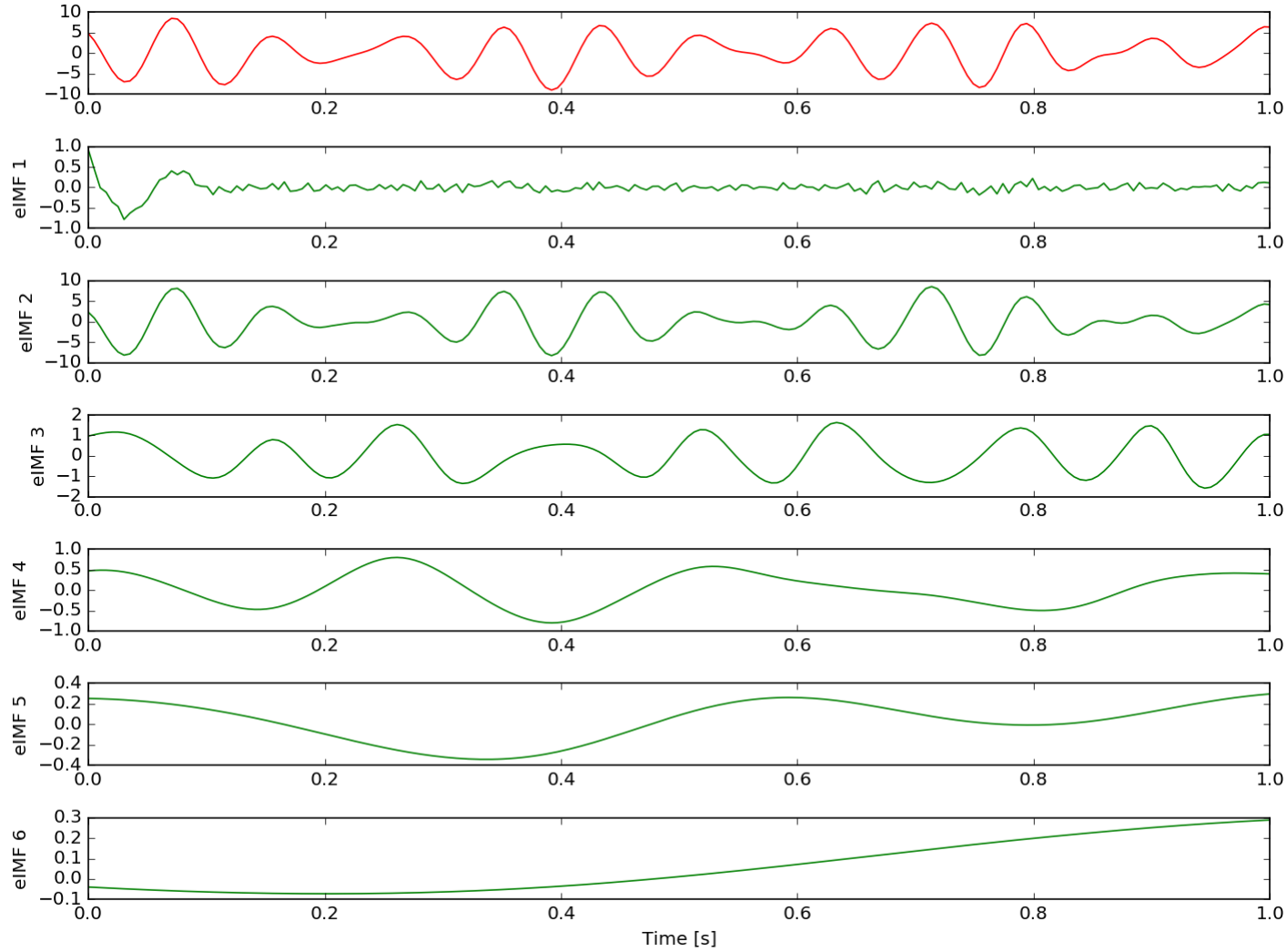
Simplest case of using Ensemble EMD (EEMD) is by importing *EEMD* and passing your signal to *eemd* method.

```
from PyEMD import EEMD
import numpy as np
import pylab as plt

# Define signal
t = np.linspace(0, 1, 200)

sin = lambda x,p: np.sin(2*np.pi*x*t+p)
S = 3*sin(18,0.2)*(t-0.2)**2
S += 5*sin(11,2.7)
S += 3*sin(14,1.6)
```

(continues on next page)

(continued from previous page)

```
S += 1*np.sin(4*2*np.pi*(t-0.8)**2)
S += t**2.1 -t

# Assign EEMD to `eemd` variable
eemd = EEMD()

# Say we want detect extrema using parabolic method
emd = eemd.EMD
emd.extrema_detection="parabol"

# Execute EEMD on S
eIMFs = eemd.eemd(S, t)
nIMFs = eIMFs.shape[0]

# Plot results
plt.figure(figsize=(12,9))
plt.subplot(nIMFs+1, 1, 1)
plt.plot(t, S, 'r')

for n in range(nIMFs):
    plt.subplot(nIMFs+1, 1, n+2)
    plt.plot(t, eIMFs[n], 'g')
    plt.ylabel("eIMF %i" %(n+1))
    plt.locator_params(axis='y', nbins=5)

plt.xlabel("Time [s]")
plt.tight_layout()
plt.savefig('eemd_example', dpi=120)
plt.show()
```


eIMF 1
eIMF 2
eIMF 3
eIMF 4
eIMF 5
eIMF 6
Time [s]

CHAPTER 5

EMD

Empirical Mode Decomposition (EMD) is an iterative procedure which decomposes signal into a set of oscillatory components, called *Intrisic Mode Functions* (IMFs).

class PyEMD.**EMD**(*spline_kind: str = 'cubic'*, *nbsym: int = 2*, ***kwargs*)

Empirical Mode Decomposition

Method of decomposing signal into Intrinsic Mode Functions (IMFs) based on algorithm presented in Huang et al. [Huang1998].

Algorithm was validated with Rilling et al. [Rilling2003] Matlab's version from 3.2007.

Threshold which control the goodness of the decomposition:

- *std_thr* — Test for the proto-IMF how variance changes between siftings.
- *svar_thr* – Test for the proto-IMF how energy changes between siftings.
- *total_power_thr* — Test for the whole decomp how much of energy is solved.
- *range_thr* — Test for the whole decomp whether the difference is tiny.

References

[Huang1998], [Rilling2003]

Examples

```
>>> import numpy as np
>>> T = np.linspace(0, 1, 100)
>>> S = np.sin(2*2*np.pi*T)
>>> emd = EMD(extrema_detection='parabol')
>>> IMFs = emd.emd(S)
>>> IMFs.shape
(1, 100)
```

__call__(*S: numpy.ndarray*, *T: Optional[numpy.ndarray] = None*, *max_imf: int = -1*) → numpy.ndarray
Call self as a function.

__init__(*spline_kind: str = 'cubic'*, *nbsym: int = 2*, ***kwargs*)
Initiate *EMD* instance.

Configuration, such as threshold values, can be passed as kwargs (keyword arguments).

Parameters

FIXE [int (default: 0)]

FIXE_H [int (default: 0)]

MAX_ITERATION [int (default 1000)] Maximum number of iterations per single sifting in EMD.

energy_ratio_thr [float (default: 0.2)] Threshold value on energy ratio per IMF check.

std_thr float [(default 0.2)] Threshold value on standard deviation per IMF check.

svar_thr float [(default 0.001)] Threshold value on scaled variance per IMF check.

total_power_thr [float (default 0.005)] Threshold value on total power per EMD decomposition.

range_thr [float (default 0.001)] Threshold for amplitude range (after scaling) per EMD decomposition.

extrema_detection [str (default 'simple')] Method used to finding extrema.

DTYPE [np.dtype (default np.float64)] Data type used.

Examples

```
>>> emd = EMD(std_thr=0.01, range_thr=0.05)
```

__weakref__
list of weak references to the object (if defined)

check_imf(*imf_new: numpy.ndarray*, *imf_old: numpy.ndarray*, *eMax: numpy.ndarray*, *eMin: numpy.ndarray*) → bool
Huang criteria for **IMF** (similar to Cauchy convergence test). Signal is an IMF if consecutive siftings do not affect signal in a significant manner.

emd(*S: numpy.ndarray*, *T: Optional[numpy.ndarray] = None*, *max_imf: int = -1*) → numpy.ndarray
Performs Empirical Mode Decomposition on signal S. The decomposition is limited to *max_imf* imfs. Returns IMF functions in numpy array format.

Parameters

S [numpy array,] Input signal.

T [numpy array, (default: None)] Position or time array. If None is passed or self.extrema_detection == "simple", then numpy range is created.

max_imf [int, (default: -1)] IMF number to which decomposition should be performed. Negative value means *all*.

Returns

IMF [numpy array] Set of IMFs produced from input signal.

end_condition(*S: numpy.ndarray*, *IMF: numpy.ndarray*) → bool
Tests for end condition of whole EMD. The procedure will stop if:

- Absolute amplitude (max - min) is below *range_thr* threshold, or
- Metric L1 (mean absolute difference) is below *total_power_thr* threshold.

Parameters

S [numpy array] Original signal on which EMD was performed.

IMF [numpy 2D array] Set of IMFs where each row is IMF. Their order is not important.

Returns

end [bool] Whether sifting is finished.

extract_max_min_spline(*T: numpy.ndarray*, *S: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]
Extracts top and bottom envelopes based on the signal, which are constructed based on maxima and minima, respectively.

Parameters

T [numpy array] Position or time array.

S [numpy array] Input data S(T).

Returns

max_spline [numpy array] Spline spanned on S maxima.

min_spline [numpy array] Spline spanned on S minima.

max_extrema [numpy array] Points indicating local maxima.

min_extrema [numpy array] Points indicating local minima.

find_extrema(*T: numpy.ndarray*, *S: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]
Returns extrema (minima and maxima) for given signal S. Detection and definition of the extrema depends on `extrema_detection` variable, set on initiation of EMD.

Parameters

T [numpy array] Position or time array.

S [numpy array] Input data S(T).

Returns

local_max_pos [numpy array] Position of local maxima.

local_max_val [numpy array] Values of local maxima.

local_min_pos [numpy array] Position of local minima.

local_min_val [numpy array] Values of local minima.

get_imfs_and_residue() → Tuple[numpy.ndarray, numpy.ndarray]
Provides access to separated imfs and residue from recently analysed signal. :return: (imfs, residue)

get_imfs_and_trend() → Tuple[numpy.ndarray, numpy.ndarray]
Provides access to separated imfs and trend from recently analysed signal. Note that this may differ from the *get_imfs_and_residue* as the trend isn't necessarily the residue. Residue is a point-wise difference between input signal and all obtained components, whereas trend is the slowest component (can be zero). :return: (imfs, trend)

prepare_points(*T: numpy.ndarray*, *S: numpy.ndarray*, *max_pos: numpy.ndarray*, *max_val: numpy.ndarray*, *min_pos: numpy.ndarray*, *min_val: numpy.ndarray*)

Performs extrapolation on edges by adding extra extrema, also known as mirroring signal. The number of added points depends on *nbsym* variable.

Parameters

T [numpy array] Position or time array.

S [numpy array] Input signal.

max_pos [iterable] Sorted time positions of maxima.

max_val [iterable] Signal values at max_pos positions.

min_pos [iterable] Sorted time positions of minima.

min_val [iterable] Signal values at min_pos positions.

Returns

max_extrema [numpy array (2 rows)] Position (1st row) and values (2nd row) of minima.

min_extrema [numpy array (2 rows)] Position (1st row) and values (2nd row) of maxima.

spline_points(*T: numpy.ndarray*, *extrema: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Constructs spline over given points.

Parameters

T [numpy array] Position or time array.

extrema [numpy array] Position (1st row) and values (2nd row) of points.

Returns

T [numpy array] Position array (same as input).

spline [numpy array] Spline array over given positions T.

CHAPTER 6

EEMD

6.1 Info

Ensemble empirical mode decomposition (EEMD) creates an ensemble of worker each of which performs an *EMD* on a copy of the input signal with added noise. When all workers finish their work a mean over all workers is considered as the true result.

6.2 Class

class PyEMD.**EEMD**(*trials: int = 100*, *noise_width: float = 0.05*, *ext_EMD=None*, *parallel: bool = False*, ***kwargs*)

Ensemble Empirical Mode Decomposition

Ensemble empirical mode decomposition (EEMD) [Wu2009] is noise-assisted technique, which is meant to be more robust than simple Empirical Mode Decomposition (EMD). The robustness is checked by performing many decompositions on signals slightly perturbed from their initial position. In the grand average over all IMF results the noise will cancel each other out and the result is pure decomposition.

Parameters

trials [int (default: 100)] Number of trials or EMD performance with added noise.

noise_width [float (default: 0.05)] Standard deviation of Gaussian noise ($\hat{\sigma}$). It's relative to absolute amplitude of the signal, i.e. $\hat{\sigma} = \sigma \cdot |\max(S) - \min(S)|$, where σ is noise_width.

ext_EMD [EMD (default: None)] One can pass EMD object defined outside, which will be used to compute IMF decompositions in each trial. If none is passed then EMD with default options is used.

parallel [bool (default: False)] Flag whether to use multiprocessing in EEMD execution. Since each EMD(s+noise) is independent this should improve execution speed considerably. *Note* that it's disabled by default because it's the most common problem when EEMD takes too long time to finish. If you set the flag to True, make also sure to set *processes* to some reasonable value.

processes [int or None (optional)] Number of processes harness when executing in parallel mode. The value should be between 1 and max that depends on your hardware.

separate_trends [bool (default: False)] Flag whether to isolate trends from each EMD decomposition into a separate component. If *true*, the resulting EEMD will contain ensemble only from IMFs and the mean residue will be stacked as the last element.

References

[Wu2009]

all_imfs
A dictionary with all computed imfs per given order.

eemd(*S: numpy.ndarray*, *T: Optional[numpy.ndarray] = None*, *max_imf: int = -1*) → numpy.ndarray
Performs EEMD on provided signal.

For a large number of iterations defined by *trials* attr the method performs `emd()` on a signal with added white noise.

Parameters

S [numpy array,] Input signal on which EEMD is performed.

T [numpy array or None, (default: None)] If none passed samples are numerated.

max_imf [int, (default: -1)] Defines up to how many IMFs each decomposition should be performed. By default (negative value) it decomposes all IMFs.

Returns

eIMF [numpy array] Set of ensemble IMFs produced from input signal. In general, these do not have to be, and most likely will not be, same as IMFs produced using EMD.

emd(*S: numpy.ndarray*, *T: numpy.ndarray*, *max_imf: int = -1*) → numpy.ndarray
Vanilla EMD method.

Provides emd evaluation from provided EMD class. For reference please see `PyEMD.EMD`.

ensemble_count() → List[int]
Count of imfs observed for given order, e.g. 1st proto-imf, in the whole ensemble.

ensemble_mean() → numpy.ndarray
Pointwise mean over computed ensemble. Same as the output of *eemd()* method.

ensemble_std() → numpy.ndarray
Pointwise standard deviation over computed ensemble.

generate_noise(*scale: float, size: Union[int, Sequence[int]]*) → numpy.ndarray
Generate noise with specified parameters. Currently supported distributions are:

- *normal* with std equal scale.
- *uniform* with range [-scale/2, scale/2].

Parameters

scale [float] Width for the distribution.

size [int] Number of generated samples.

Returns

noise [numpy array] Noise sampled from selected distribution.

get_imfs_and_residue() → Tuple[numpy.ndarray, numpy.ndarray]
Provides access to separated imfs and residue from recently analysed signal.

Returns

(imfs, residue) [(np.ndarray, np.ndarray)] Tuple that contains all imfs and a residue (if any).

noise_seed(*seed: int*) → None
Set seed for noise generation.

CHAPTER 7

CEEMDAN

7.1 Info

Complete ensemble EMD with adaptive noise (CEEMDAN) performs an EEMD with the difference that the information about the noise is shared among all workers.

7.2 Class

class PyEMD.**CEEMDAN**(*trials: int = 100*, *epsilon: float = 0.005*, *ext_EMD=None*, *parallel: bool = False*, ***kwargs*)

"Complete Ensemble Empirical Mode Decomposition with Adaptive Noise"

"Complete ensemble empirical mode decomposition with adaptive noise" (CEEMDAN) [Torres2011] is noise-assisted EMD technique. Word "complete" presumably refers to decomposing completely everything, even added perturbation (noise).

Provided implementation contains proposed "improvements" from paper [Colominas2014].

Any parameters can be updated directly on the instance or passed through a *configuration* dictionary.

Goodness of the decomposition can be configured by modifying threshold values. Two are *range_thr* and *total_power_thr* which relate to the value range (max - min) and check for total power below, respectively.

Parameters

trials [int (default: 100)] Number of trials or EMD performance with added noise.

epsilon [float (default: 0.005)] Scale for added noise (ϵ) which multiply std σ: $\beta = \epsilon \cdot \sigma$

ext_EMD [EMD (default: None)] One can pass EMD object defined outside, which will be used to compute IMF decompositions in each trial. If none is passed then EMD with default options is used.

parallel [bool (default: False)] Flag whether to use multiprocessing in EEMD execution. Since each EMD(s+noise) is independent this should improve execution speed considerably. *Note*

that it's disabled by default because it's the most common problem when CEEMDAN takes too long time to finish. If you set the flag to True, make also sure to set *processes* to some reasonable value.

processes [int or None (optional)] Number of processes harness when executing in parallel mode. The value should be between 1 and max that depends on your hardware.

References

[Torres2011], [Colominas2014]

__call__(*S: numpy.ndarray*, *T: Optional[numpy.ndarray] = None*, *max_imf: int = -1*) → numpy.ndarray
Call self as a function.

__init__(*trials: int = 100*, *epsilon: float = 0.005*, *ext_EMD=None*, *parallel: bool = False*, ***kwargs*)
Configuration can be passed through keyword parameters. For example, updating threshold would be through:

Example 1: >>> config = {"range_thr": 0.001, "total_power_thr": 0.01} >>> emd = EMD(**config) Example 2: >>> emd = EMD(range_thr=0.001, total_power_thr=0.01)

__weakref__
list of weak references to the object (if defined)

emd(*S: numpy.ndarray*, *T: Optional[numpy.ndarray] = None*, *max_imf: int = -1*) → numpy.ndarray
Vanilla EMD method.

Provides emd evaluation from provided EMD class. For reference please see *PyEMD.EMD*.

end_condition(*S: numpy.ndarray*, *cIMFs: numpy.ndarray*, *max_imf: int*) → bool
Test for end condition of CEEMDAN.

Procedure stops if:

- number of components reach provided *max_imf*, or
- last component is close to being pure noise (range or power), or
- set of provided components reconstructs sufficiently input.

Parameters

S [numpy array] Original signal on which CEEMDAN was performed.

cIMFs [numpy 2D array] Set of cIMFs where each row is cIMF.

max_imf [int] The maximum number of imfs to extract.

Returns

end [bool] Whether to stop CEEMDAN.

generate_noise(*scale: float, size: Union[int, Sequence[int]]*) → numpy.ndarray
Generate noise with specified parameters. Currently supported distributions are:

- *normal* with std equal scale.
- *uniform* with range [-scale/2, scale/2].

Parameters

scale [float] Width for the distribution.

size [int or shape] Shape of the noise that is added. In case of *int* an array of that len is generated.

Returns

noise [numpy array] Noise sampled from selected distribution.

get_imfs_and_residue() → Tuple[numpy.ndarray, numpy.ndarray]
Provides access to separated imfs and residue from recently analysed signal. :return: (imfs, residue)

noise_seed(*seed: int*) → None
Set seed for noise generation.

CHAPTER 8

Visualisation

A simple visualisation helper.

class PyEMD.**Visualisation**(*emd_instance=None*)

Simple visualisation helper.

This class is for quick and simple result visualisation.

plot_imfs(*imfs=None*, *residue=None*, *t=None*, *include_residue=True*)

Plots and shows all IMFs.

All parameters are optional since the *emd* object could have been passed when instantiating this object.

The residual is an optional and can be excluded by setting *include_residue=False*.

plot_instant_freq(*t*, *imfs=None*, *order=False*, *alpha=None*)

Plots and shows instantaneous frequencies for all provided imfs.

The necessary parameter is *t* which is the time array used to compute the EMD. One should pass *imfs* if no *emd* instances is passed when creating the Visualisation object.

Parameters

order [bool (default: False)] Represents whether the finite difference scheme is low-order (1st order forward scheme) or high-order (6th order compact scheme). The default value is False (low-order)

alpha [float (default: None)] Filter intensity. Default value is None, which is equivalent to *alpha* = 0.5, meaning that no filter is applied. The *alpha* values must be in between -0.5 (fully active) and 0.5 (no filter).

CHAPTER 9

Experimental

Also known as **not supported**.

Methods discussed and provided here have no guarantee to work or provide any meaningful results. These are somehow abandoned projects and unfortunately mid-way through. They aren't completely discarded simply because of hope that maybe someday someone will come and help fix them. We all know that the best motivation to do something is to be annoyed by the current state. Seriously though, mode decomposition in 2D and multi-dim is an interesting topic. Please?

9.1 BEMD

> **Warning:** Important This is an experimental module. Please use it with care as no guarantee can be given for obtaining reasonable results, or that they will be computed index the most computation optimal way.

9.1.1 Info

BEMD performed on bidimensional data such as images. This procedure uses morphological operators to detect regional maxima which are then used to span surface envelope with a radial basis function.

9.1.2 Class

9.2 EMD2D

> **Warning:** Important This is an experimental module. Please use it with care as no guarantee can be given for obtaining reasonable results, or that they will be computed index the most computation optimal way.

9.2.1 Info

EMD performed on images. This version uses for envelopes 2D splines, which are span on extrema defined through maximum filter.

9.2.2 Class

class PyEMD.EMD2d.**EMD2D**(***config*)

Empirical Mode Decomposition on images.

Important This is an experimental module. Experiments performed using this module didn't provide acceptable results, either in actual output nor in computation performance. The author is not an expert in image processing so it's very likely that the code could have been improved. Take your best shot.

Method decomposes images into 2D representations of loose Intrinsic Mode Functions (IMFs).

The current version of the algorithm detects local extrema, separately minima and maxima, and then connects them to create envelopes. These are then used to create a mean trend and subtracted from the input.

Threshold values that control goodness of the decomposition:

- *mse_thr* — proto-IMF check whether small mean square error.
- *mean_thr* — proto-IMF chekc whether small mean value.

__call__(*image*, *max_imf=-1*)

Call self as a function.

__init__(***config*)

Initialize self. See help(type(self)) for accurate signature.

__weakref__

list of weak references to the object (if defined)

check_proto_imf(*proto_imf*, *proto_imf_prev*, *mean_env*)

Check whether passed (proto) IMF is actual IMF. Current condition is solely based on checking whether the mean is below threshold.

Parameters

proto_imf [numpy 2D array] Current iteration of proto IMF.

proto_imf_prev [numpy 2D array] Previous iteration of proto IMF.

mean_env [numpy 2D array] Local mean computed from top and bottom envelopes.

Returns

boolean Whether current proto IMF is actual IMF.

emd(*image*, *max_imf=-1*)

Performs EMD on input image with specified parameters.

Parameters

image [numpy 2D array,] Image which will be decomposed.

max_imf [int, (default: -1)] IMF number to which decomposition should be performed. Negative value means *all*.

Returns

IMFs [numpy 3D array] Set of IMFs in form of numpy array where the first dimension relates to IMF's ordinary number.

classmethod end_condition(*image*, *IMFs*)
Determins whether decomposition should be stopped.

Parameters

image [numpy 2D array] Input image which is decomposed.

IMFs [numpy 3D array] Array for which first dimensions relates to respective IMF, i.e. (numIMFs, imageX, imageY).

extract_max_min_spline(*image*)
Calculates top and bottom envelopes for image.

Parameters

image [numpy 2D array]

Returns

min_env [numpy 2D array] Bottom envelope in form of an image.

max_env [numpy 2D array] Top envelope in form of an image.

classmethod find_extrema(*image*)
Finds extrema, both mininma and maxima, based on local maximum filter. Returns extrema in form of two rows, where the first and second are positions of x and y, respectively.

Parameters

image [numpy 2D array] Monochromatic image or any 2D array.

Returns

min_peaks [numpy array] Minima positions.

max_peaks [numpy array] Maxima positions.

classmethod prepare_image(*image*)
Prepares image for edge extrapolation. Method bloats image by mirroring it along all axes. This turns extrapolation on edges into interpolation within bigger image.

Parameters

image [numpy 2D array] Image for which interpolation is required,

Returns

image [numpy 2D array] Big image based on the input. Grid 3x3 where the center block is input and neighbouring panels are respective mirror images.

classmethod spline_points(*X*, *Y*, *Z*, *xi*, *yi*)
Interpolates for given set of points

CHAPTER 10

Contact

The best way to reach out is through creating an issue on the GitHub page. That way anyone can help or chime in on the severity of the issue. Although that might seem "slower" than email, it is honestly much faster.

Tickets https://github.com/laszukdawid/PyEMD/issues

Email \pyemd@\dawid.lasz.uk (remove slashes \).

Homepage http://www.laszukdawid.com

GitHub You can also visit PyEMD GitHub project page for this project.

CHAPTER 11

Indices and tables

- genindex
- search

Bibliography

[Huang1998] N. E. Huang et al., “The empirical mode decomposition and the Hilbert spectrum for non-linear and non stationary time series analysis”, Proc. Royal Soc. London A, Vol. 454, pp. 903-995, 1998

[Rilling2003] G. Rilling, P. Flandrin and P. Goncalves, “On Empirical Mode Decomposition and its algorithms”, IEEE-EURASIP Workshop on Nonlinear Signal and Image Processing NSIP-03, Grado (I), June 2003

[Wu2009] Z. Wu and N. E. Huang, “Ensemble empirical mode decomposition: A noise-assisted data analysis method”, Advances in Adaptive Data Analysis, Vol. 1, No. 1 (2009) 1-41.

[Torres2011] M.E. Torres, M.A. Colominas, G. Schlotthauer, P. Flandrin A complete ensemble empirical mode decomposition with adaptive noise. Acoustics, Speech and Signal Processing (ICASSP), 2011, pp. 4144–4147

[Colominas2014] M.A. Colominas, G. Schlotthauer, M.E. Torres, Improved complete ensemble EMD: A suitable tool for biomedical signal processing, In Biomed. Sig. Proc. and Control, V. 14, 2014, pp. 19–29

Index

Symbols

A

C

E

F

G

N

P

S

V