brainpipe Documentation

Release 0.1.6

Etienne Combrisson

Oct 09, 2016

Brainpipe is a python toolbox dedicated for neuronal signals analysis and machine-learning. The aim is to provide a variety of tools to extract informations from neural activities (features) and use machine-learning to validate hypothesis. The machine-learning used the excellent scikit-learn library. Brainpipe can also perform parallel computing and try to optimize RAM usage for our 'large' datasets. If you want to have a quick overview of what you can actually do, checkout the *References*.

It's evolving every day! So if you have problems, bugs or if you want to collaborate and add your own tools, contact me at e.combrisson@gmail.com

CHAPTER

ONE

REQUIREMENT

brainpipe is developed on Python 3, so the compatibility with python 2 is not guaranted! (not tested yet)

Please, check if you have this toolbox installed and already up-to-date:

- matplotlib (visualization)
- scikit-learn (machine learning)
- joblib (parallel computing)
- scipy
- numpy

CHAPTER

TWO

INSTALLATION

For instance, the easiest way of installing brainpipe is to use github (brainpipe).

Go to your python site-package folder (ex: anaconda3/lib/python3.5/site-packages) and in a terminal run

```
git clone git@github.com:EtienneCmb/brainpipe.git
```

CHAPTER

THREE

WHAT'S NEW

3.1 v0.1.6

- Fix scikit-learn v0.18 compatibility
- Add mutli-features pipelines

3.2 v0.1.5

- classify: fit_stat() has been removed, there is only fit() now. confusion_matrix() has been rename to cm()
- LeavePSubjectOut: new leave p-subjects out cross validation
- classification: embedded visualization (daplot, cmplot), statistics (classify.stat.), dataframe for quick settings summarize (classify.info.) with excel exportation
- New folder: ipython notebook examples
- tools: unique ordered function

3.3 v0.1.4

- PAC: new surrogates method (shuffle amplitude time-series) + phase synchro
- New features (phase-locked power (coupling), ERPAC (coupling), pfdphase (coupling), Phase-Locking Value (coupling), Phase-locking Factor (PLF in spectral) and PSD based features)
- New tools for physiological bands definition
- New plotting function (addPval, continuouscol)
- Start to make the python adaptation of circstat Matlab toolbox
- Add contour to plot2D() and some other parameters
- New doc ! Checkout the *References*
- Bug fix: pac phase shuffling method, coupling time vector, statictical evaluation of permutations

3.4 v0.1.0

- **Statistics:**
  - Permutations: array optimized permutation module
  - p-values on permutations can be compute on 1 tail (upper and lower part) or two tails
  - metric:
  - Multiple comparison: maximum statistique, Bonferroni, FDR
- **Features:**
  - sigfilt//amplitude//power//TF: wilcoxon/Kruskal-Wallis/permutations stat test (comparison with baseline)
  - PAC: new Normalized Direct PAC method (Ozkurt, 2012)
- **Visualization:**
  - tilerplot() with plot1D() and plot2D() with automatic control of subplot
- **Tools:**
  - Array: ndsplit and ndjoin method which doesn't depend on odd/even size (but return list)
  - squarefreq() generate a square frequency vector

CHAPTER

FOUR

ORGANIZATION

Todo

Add trials rejection (MATLAB code adaptation)

```
from brainpipe.preprocessing import *
```

4.1 Bipolarization

preprocessing.**bipolarization**(*data*, *channel*, *dim=0*, *xyz=None*, *sep='.'*, *unbip=None*, *rmchan=None*, *keepchan='all'*, *rmspace=True*, *rmalone=True*)

Bipolarize data

Args:

data: array Data to bipolarize

channel: list List of channels name

Kwargs:

dim: integer, optional, [def: 0] Specify where is the channel dimension of data

xyz: array, optional, [def: None] Electrode coordinates. Must be a n_channel x 3

sep: string, optional, [def: '.'] Separator to simplify electrode names by removing undesired name after the sep. For example, if channel = ['h1.025', 'h2.578'] and sep='.', the final name will be 'h2-h1'.

unbip: list, optional, [def: None] Channel that don't need a bipolarization but to keep. This list can either be the index or the name of the channel.

rmchan: list, optional, [def: None] Channel to remove. This list can either be the index or the name of the channel.

keepchan: list, optional, [def: 'all'] Channel to keep. This list can either be the index or the name of the channel.

rmspace: bool, optional, [def: True] Remove undesired space in channel names.

rmalone: bool, optional, [def: True] Remove electrodes that cannot be bipolarized.

Returns:

data_b: array Bipolarized data.

channel_b: list List of the bipolrized channels name.

xyz_b: array Array of the new xyz coordinates.

Example :

```
>>> x = 47
>>> f = np.array(47, 54, 85)
```

4.2 Physiology

class preprocessing.**xyz2phy**(*nearest=True*, *r=5*, *rm_unfound=False*)
Transform coordinates to physiological informations.

Args:

nearest: bool, optional, [def: True] If no physiological is found, use this parameter to force to search in sphere of interest.

r: integer, optional, [def: 5] Find physiological informations inside a sphere of interest with a radius of r.

rm_unfound: bool, optional, [def: False] Remove un-found structures

search(*df*, **keep*)
Search in a pandas dataframe.

Args:

df: pandas dataframe The dataframe to filter

keep: tuple/list Control the informations to search. See the syntax definition

Return: List of row index for informations found.

keep(*df*, **keep*, ***, *keep_idx=True*)
Filter a pandas dataframe and keep only interresting rows.

Args:

df: pandas dataframe The dataframe to filter

keep: tuple/list Control the informations to keep. See the syntax definition

keep_idx: bool, optional [def [True]] Add a column to df to check what are the rows that has been kept.

Return: A pandas Dataframe with only the informations to keep.

remove(*df*, **rm*, ***, *rm_idx=True*)
Filter a pandas dataframe and remove only interresting rows.

Args:

df: pandas dataframe The dataframe to be filter

rm: tuple/list Control the informations to remove. See the syntax definition

rm_idx: bool, optional [def [True]] Add a column to df to check what are the rows that has been removed.

Return: A pandas Dataframe without the removed informations.

get(*xyz*, *channel=[]*)
Get physiological informations from (x,y,z) coordinates.

Args:

xyz: array Array of coordinates. The shape of xyz must be (n_electrodes x 3).

channel: list, optional, [def: []] List of channels name.

Return: A pandas DataFrame with physiological informations;

Import features:

```
from brainpipe.features import *
```

Todo

Coherence // permutation entropy // wavelet filtering (numpy.wlt)

4.3 Presentation

In order to classify diffrent conditions, you can extract from your neural signals, a large variety of features. The aim of a feature is to verify if it contains the main information you want to classify. For example, let's say you search if an electrode is accurate to differenciate resting state from motor behavior. You can for example extract beta or gamma power from this electrode and classify your resting state versus motor using power features. Here's the list of all the implemented features:

- *Filtered signal*
- *Amplitude*
- *Power*
- *Time-Frequency*
- *Phase*
- *Phase-Locking Factor*
- *Phase-Amplitude Coupling*
- *Prefered-phase*
- *Event Related Phase-Amplitude Coupling*
- *Phase-locked power*
- *Phase-Locking Value*
- *Power Spectrum Density*
- *Power PSD*
- *Spectral entropy*

4.4 Filtering based

Those following features use filtering method to extract informations in specifics frequency bands

4.4.1 Filtered signal

class feature.**sigfilt**(*sf, npts, f=[60, 200], baseline=None, norm=None, window=None, width=None, step=None, split=None, time=None, **kwargs*)

Extract the filtered signal. This class is optimized for 3D arrays.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] None and the width and step parameters will be considered

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

norm: int, optional [def: None]

Number to choose the normalization method

- 0: No normalisation
- 1: Substraction
- 2: Division
- 3: Substract then divide
- 4: Z-score

baseline: tuple/list of int [def: None] Define a window to normalize the power

split: int or list of int, optional [def: None] Split the frequency band f in "split" band width. If f is a list which contain couple of frequencies, split is a list too.

kwargs: supplementar arguments for filtering

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get (*x*, *statmeth=None*, *tail=2*, *n_perm=200*, *metric='m_center'*, *maxstat=False*, *n_jobs=-1*)
Get the spectral feature of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

statmeth: string, optional, [def: None] Method to evaluate the statistical significiancy. To get p-values, the program will compare real values with a defined baseline. As a consequence, the 'norm' and 'baseline' parameter should not be None.

- 'permutation': randomly shuffle real data with baseline.Control the number of permutations with the n_perm parameter. For example, if n_perm = 1000, this mean that minimum p-valueswill be 0.001.
- 'wilcoxon': Wilcoxon signed-rank test
- 'kruskal': Kruskal-Wallis H-test

n_perm: integer, optional, [def: 200] Number of permutations for assessing statistical significiancy.

tail: int, optional, [def: 2] For the permutation method, get p-values from one or two tails of the distribution. Use -1 for testing A<B, 1 for A>B and 2 for A~=B.

metric: string/function type, optional, [def: 'm_center'] Use diffrent metrics to normalize data and permutations by the defined baseline. Use:

- None: compare directly values without transformation
- 'm_center': (A-B)/mean(B) transformation
- 'm_zscore': (A-B)/std(B) transformation
- 'm_minus': (A-B) transformation
- function: user defined function [def myfcn(A, B): return array_like]

maxstat: bool, optional, [def: False] Correct p-values with maximum statistique. If maxstat is True, the correction will be applied only trhough frequencies.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

xF: array The un/normalized feature of x, with a shape of (n_frequency x n_electrodes x n_window x n_trials)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

4.4.2 Amplitude

class feature.**amplitude**(*sf, npts, f=[60, 200], baseline=None, norm=None, method='hilbert1', window=None, width=None, step=None, split=None, time=None, **kwargs*)
Extract the amplitude of the signal. This class is optimized for 3D arrays.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] None and the width and step parameters will be considered

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

norm: int, optional [def: None]

Number to choose the normalization method

- 0: No normalisation
- 1: Substraction
- 2: Division
- 3: Substract then divide
- 4: Z-score

baseline: tuple/list of int [def: None] Define a window to normalize the power

split: int or list of int, optional [def: None] Split the frequency band f in "split" band width. If f is a list which contain couple of frequencies, split is a list too.

method: string

Method to transform the signal. Possible values are:

- 'hilbert': apply a hilbert transform to each column
- 'hilbert1': hilbert transform to a whole matrix
- 'hilbert2': 2D hilbert transform
- 'wavelet': wavelet transform

kwargs: supplementar arguments for filtering

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get (*x*, *statmeth=None*, *tail=2*, *n_perm=200*, *metric='m_center'*, *maxstat=False*, *n_jobs=-1*)
Get the spectral feature of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

statmeth: string, optional, [def: None] Method to evaluate the statistical significiancy. To get p-values, the program will compare real values with a defined baseline. As a consequence, the 'norm' and 'baseline' parameter should not be None.

- 'permutation': randomly shuffle real data with baseline.Control the number of permutations with the n_perm parameter. For example, if n_perm = 1000, this mean that minimum p-valueswill be 0.001.
- 'wilcoxon': Wilcoxon signed-rank test
- 'kruskal': Kruskal-Wallis H-test

n_perm: integer, optional, [def: 200] Number of permutations for assessing statistical significiancy.

tail: int, optional, [def: 2] For the permutation method, get p-values from one or two tails of the distribution. Use -1 for testing A<B, 1 for A>B and 2 for A~=B.

metric: string/function type, optional, [def: 'm_center'] Use diffrent metrics to normalize data and permutations by the defined baseline. Use:

- None: compare directly values without transformation
- 'm_center': (A-B)/mean(B) transformation
- 'm_zscore': (A-B)/std(B) transformation
- 'm_minus': (A-B) transformation
- function: user defined function [def myfcn(A, B): return array_like]

maxstat: bool, optional, [def: False] Correct p-values with maximum statistique. If maxstat is True, the correction will be applied only trhough frequencies.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

xF: array The un/normalized feature of x, with a shape of (n_frequency x n_electrodes x n_window x n_trials)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

4.4.3 Power

class feature.**power**(*sf, npts, f=[60, 200], baseline=None, norm=None, method='hilbert1', window=None, width=None, step=None, split=None, time=None, **kwargs*)
Extract the power of the signal. This class is optimized for 3D arrays.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] None and the width and step parameters will be considered

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

norm: int, optional [def: None]

Number to choose the normalization method

- 0: No normalisation
- 1: Substraction
- 2: Division
- 3: Substract then divide
- 4: Z-score

baseline: tuple/list of int [def: None] Define a window to normalize the power

split: int or list of int, optional [def: None] Split the frequency band f in "split" band width. If f is a list which contain couple of frequencies, split is a list too.

method: string

Method to transform the signal. Possible values are:

- 'hilbert': apply a hilbert transform to each column
- 'hilbert1': hilbert transform to a whole matrix
- 'hilbert2': 2D hilbert transform
- 'wavelet': wavelet transform

kwargs: supplementar arguments for filtering

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get (*x*, *statmeth=None*, *tail=2*, *n_perm=200*, *metric='m_center'*, *maxstat=False*, *n_jobs=-1*)
Get the spectral feature of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

statmeth: string, optional, [def: None] Method to evaluate the statistical significiancy. To get p-values, the program will compare real values with a defined baseline. As a consequence, the 'norm' and 'baseline' parameter should not be None.

- 'permutation': randomly shuffle real data with baseline.Control the number of permutations with the n_perm parameter. For example, if n_perm = 1000, this mean that minimum p-valueswill be 0.001.
- 'wilcoxon': Wilcoxon signed-rank test
- 'kruskal': Kruskal-Wallis H-test

n_perm: integer, optional, [def: 200] Number of permutations for assessing statistical significiancy.

tail: int, optional, [def: 2] For the permutation method, get p-values from one or two tails of the distribution. Use -1 for testing A<B, 1 for A>B and 2 for A~=B.

metric: string/function type, optional, [def: 'm_center'] Use diffrent metrics to normalize data and permutations by the defined baseline. Use:

- None: compare directly values without transformation
- 'm_center': (A-B)/mean(B) transformation
- 'm_zscore': (A-B)/std(B) transformation
- 'm_minus': (A-B) transformation
- function: user defined function [def myfcn(A, B): return array_like]

maxstat: bool, optional, [def: False] Correct p-values with maximum statistique. If maxstat is True, the correction will be applied only trhough frequencies.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

xF: array The un/normalized feature of x, with a shape of (n_frequency x n_electrodes x n_window x n_trials)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

4.4.4 Time-Frequency

class feature.**TF** (*sf*, *npts*, *f=(2*, *200*, *10*, *5)*, *baseline=None*, *norm=None*, *method='hilbert1'*, *window=None*, *width=None*, *step=None*, *time=None*, ***kwargs*)
Extract the time-frequency map of the signal. This class is optimized for 3D arrays.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] None and the width and step parameters will be considered

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

norm: int, optional [def: None]

Number to choose the normalization method

- 0: No normalisation
- 1: Substraction
- 2: Division
- 3: Substract then divide
- 4: Z-score

baseline: tuple/list of int [def: None] Define a window to normalize the power

split: int or list of int, optional [def: None] Split the frequency band f in "split" band width. If f is a list which contain couple of frequencies, split is a list too.

method: string

Method to transform the signal. Possible values are:

- 'hilbert': apply a hilbert transform to each column
- 'hilbert1': hilbert transform to a whole matrix
- 'hilbert2': 2D hilbert transform
- 'wavelet': wavelet transform

kwargs: supplementar arguments for filtering

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get (*x*, *statmeth=None*, *tail=2*, *n_perm=200*, *metric='m_center'*, *maxstat=False*, *n_jobs=-1*)
Get the spectral feature of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

statmeth: string, optional, [def: None] Method to evaluate the statistical significiancy. To get p-values, the program will compare real values with a defined baseline. As a consequence, the 'norm' and 'baseline' parameter should not be None.

- 'permutation': randomly shuffle real data with baseline.Control the number of permutations with the n_perm parameter. For example, if n_perm = 1000, this mean that minimum p-valueswill be 0.001.
- 'wilcoxon': Wilcoxon signed-rank test
- 'kruskal': Kruskal-Wallis H-test

n_perm: integer, optional, [def: 200] Number of permutations for assessing statistical significiancy.

tail: int, optional, [def: 2] For the permutation method, get p-values from one or two tails of the distribution. Use -1 for testing A<B, 1 for A>B and 2 for A~=B.

metric: string/function type, optional, [def: 'm_center'] Use diffrent metrics to normalize data and permutations by the defined baseline. Use:

- None: compare directly values without transformation
- 'm_center': (A-B)/mean(B) transformation
- 'm_zscore': (A-B)/std(B) transformation
- 'm_minus': (A-B) transformation
- function: user defined function [def myfcn(A, B): return array_like]

maxstat: bool, optional, [def: False] Correct p-values with maximum statistique. If maxstat is True, the correction will be applied only trhough frequencies.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

xF: array The un/normalized feature of x, with a shape of (n_frequency x n_electrodes x n_window x n_trials)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

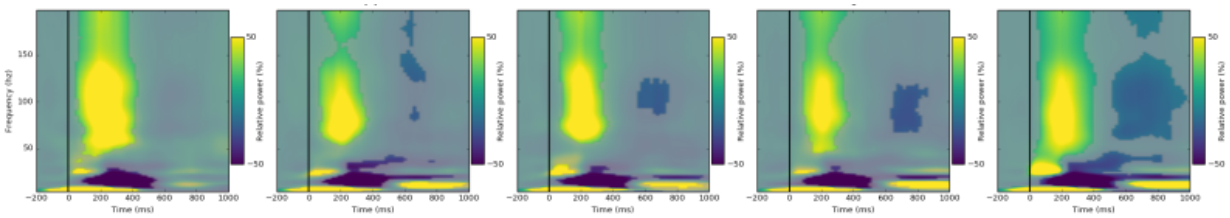

Fig. 4.1: Example of a Time-frequency map

4.4.5 Phase

class feature.**phase**(*sf, npts, f=[2, 4], method='hilbert', window=None, width=None, step=None, time=None, **kwargs*)

Extract the phase of a signal. This class is optimized for 3D arrays.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] None and the width and step parameters will be considered

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

method: string

Method to transform the signal. Possible values are:

- 'hilbert': apply a hilbert transform to each column
- 'hilbert1': hilbert transform to a whole matrix
- 'hilbert2': 2D hilbert transform

kwargs: supplementar arguments for filtering

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get(*x, getstat=True, n_jobs=-1*)

Get the spectral phase of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

getstat: bool, optional, [def: True] Set it to True if p-values should be computed. Satistical p-values are comptuted using Rayleigh test.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

xF: array The phase of x, with a shape of (n_frequency x n_electrodes x n_window x n_trials)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

get(*x*, *getstat=True*, *n_jobs=-1*)
Get the spectral phase of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

getstat: bool, optional, [def: True] Set it to True if p-values should be computed. Satistical p-values are comptuted using Rayleigh test.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

xF: array The phase of x, with a shape of (n_frequency x n_electrodes x n_window x n_trials)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

4.4.6 Phase-Locking Factor

class feature.**PLF**(*sf, npts, f=[2, 4], method='hilbert', window=None, width=None, step=None, time=None, **kwargs*)
Extract the phase-locking factor of a signal. This class is optimized for 3D arrays.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] None and the width and step parameters will be considered

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

method: string

Method to transform the signal. Possible values are:

- 'hilbert': apply a hilbert transform to each column
- 'hilbert1': hilbert transform to a whole matrix
- 'hilbert2': 2D hilbert transform

kwargs: supplementar arguments for filtering

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get (*x*, *getstat=True*, *n_jobs=-1*)
Get the phase-locking factor of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

getstat: bool, optional, [def: True] Set it to True if p-values should be computed. Satistical p-values are comptuted using Rayleigh test.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

plf: array The PLF of x, with a shape of (n_frequency x n_electrodes x n_window)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

get (*x*, *getstat=True*, *n_jobs=-1*)
Get the phase-locking factor of the signal x.

Args:

x: array Data with a shape of (n_electrodes x n_pts x n_trials)

Kargs:

getstat: bool, optional, [def: True] Set it to True if p-values should be computed. Satistical p-values are comptuted using Rayleigh test.

n_jobs: integer, optional, [def: -1] Control the number of jobs to extract features. If n_jobs = -1, all the jobs are used.

Return:

plf: array The PLF of x, with a shape of (n_frequency x n_electrodes x n_window)

pvalues: array p-values with a shape of (n_frequency x n_electrodes x n_window)

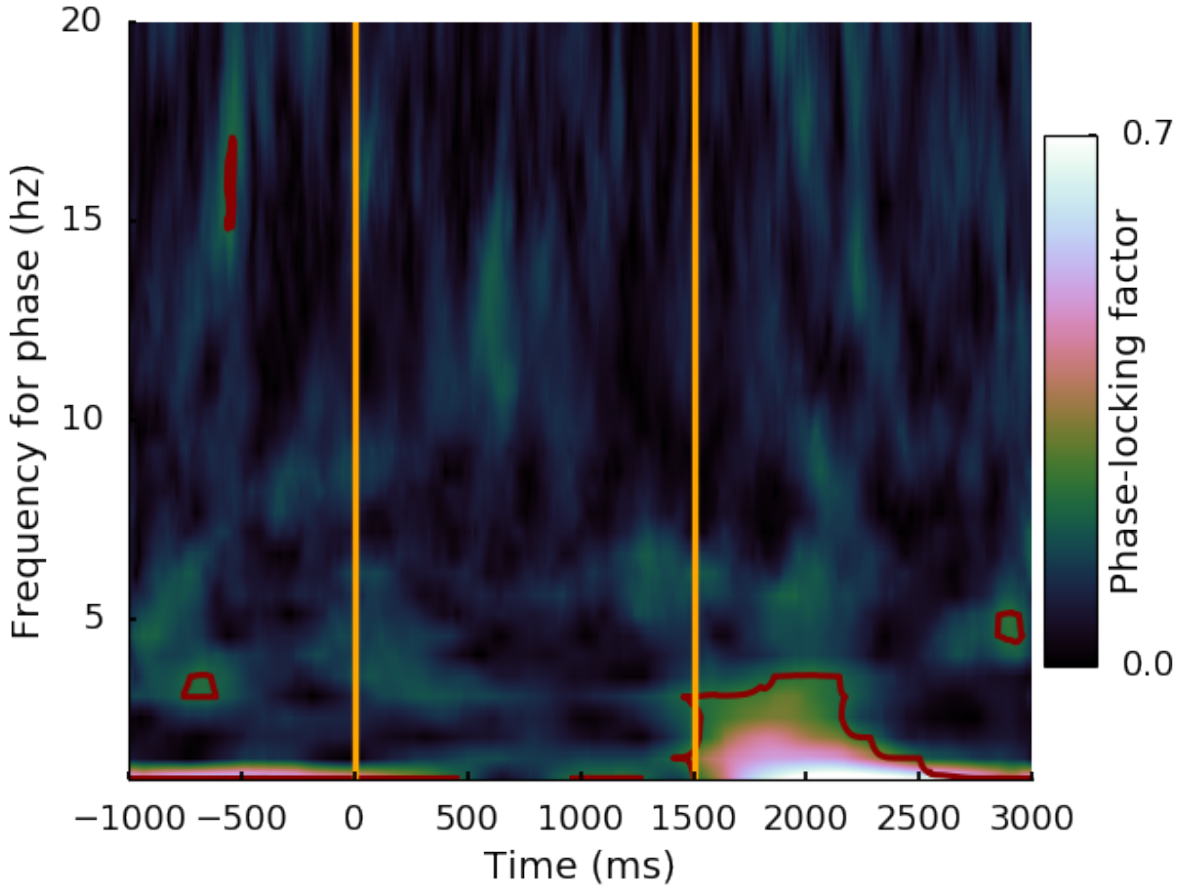

Fig. 4.2: Example of a Phase-Locking Factor map

4.5 Coupling features

Those following features use coupling (either distant or locals coupling)

4.5.1 Phase-Amplitude Coupling

class `feature.pac`(*sf, npts, Id='113', pha_f=[2, 4], pha_meth='hilbert', pha_cycle=3, amp_f=[60, 200], amp_meth='hilbert', amp_cycle=6, nbins=18, window=None, width=None, step=None, time=None, **kwargs*)

Compute the phase-amplitude coupling (pac) either in local or distant coupling. PAC require three things:

•Main method to compute it

•Surrogates to correct the true pac estimation

•A normalization method to correct pas by surrogates

Contributor: Juan LP Soto.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

Kargs:

Id: string, optional, [def: '113'] The Id correspond to the way of computing pac. Id is composed of three digits [ex : Id='210']

- First digit: refer to the pac method:
  - '1': Mean Vector Length [1]
  - '2': Kullback-Leibler Divergence [2]
  - '3': Heights Ratio
  - '4': Phase synchrony (or adapted PLV) [5]
  - '5': ndPAC [3]
- Second digit: refer to the method for computing surrogates:
  - '0': No surrogates
  - '1': Swap trials phase/amplitude [2]
  - '2': Swap trials amplitude [4]
  - '3': Shuffle phase time-series
  - '4': Shuffle amplitude time-series
  - '5': Time lag [1] [NOT IMPLEMENTED YET]
  - '6': Circular shifting [NOT IMPLEMENTED YET]
- Third digit: refer to the normalization method for correction:

[1] Canolty et al, 2006
[2] Tort et al, 2010
[5] Penny et al, 2008
[3] Ozkurt et al, 2012
[4] Bahramisharif et al, 2013

- '0': No normalization
- '1': Substract the mean of surrogates
- '2': Divide by the mean of surrogates
- '3': Substract then divide by the mean of surrogates
- '4': Z-score

So, if Id='143', this mean that pac will be evaluate using the Modulation Index ('1'), then surrogates are computing by randomly shuffle amplitude values ('4') and finally, the true pac value will be normalized by substracting then dividing by the mean of surrogates.

pha_f: tuple/list, optional, [def: [2,4]] List containing the couple of frequency bands for the phase. Example: f=[[2,4], [5,7], [60,250]]

pha_meth: string, optional, [def: 'hilbert'] Method for the phase extraction.

pha_cycle: integer, optional, [def: 3] Number of cycles for filtering the phase.

amp_f: tuple/list, optional, [def: [60,200]] List containing the couple of frequency bands for the amplitude. Each couple can be either a list or a tuple.

amp_meth: string, optional, [def: 'hilbert'] Method for the amplitude extraction.

amp_cycle: integer, optional, [def: 6] Number of cycles for filtering the amplitude.

nbins: integer, optional, [def: 18] Some pac method (like Kullback-Leibler Distance or Heights Ratio) need a binarization of the phase. nbins control the number of bins.

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] Width and step parameters will be ignored.

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

filtname: string, optional [def: 'fir1']

Name of the filter. Possible values are:

- 'fir1': Window-based FIR filter design
- 'butter': butterworth filter
- 'bessel': bessel filter

cycle: int, optional [def: 3] Number of cycle to use for the filter. This parameter is only avaible for the 'fir1' method

order: int, optional [def: 3] Order of the 'butter' or 'bessel' filter

axis: int, optional [def: 0] Filter accross the dimension 'axis'

get (*xpha*, *xamp*, *n_perm=200*, *p=0.05*, *matricial=False*, *n_jobs=-1*)
Get the normalized cfc mesure between an xpha and xamp signals.

Args:

xpha: array Signal for phase. The shape of xpha should be : (n_electrodes x n_pts x n_trials)

xamp: array Signal for amplitude. The shape of xamp should be : (n_electrodes x n_pts x n_trials)

Kargs:

n_perm: integer, optional, [def: 200] Number of permutations for normalizing the cfc mesure.

p: float, optional, [def: 0.05] p-value for the statistical method of Ozkurt 2012.

matricial: bool, optional, [def: False] Some methods can work in matricial computation. This can lead to a 10x or 30x time faster. But, please, monitor your RAM usage beacause this parameter can use a lot of RAM. So, turn this parameter in case of small computation.

n_jobs: integer, optional, [def: -1] Control the number of jobs for parallel computing. Use 1, 2, .. depending of your number or cores. -1 for all the cores.

If the same signal is used (example : xpha=x and xamp=x), this mean the program compute a local cfc.

Returns:

ncfc: array The cfc mesure of size : (n_amplitude x n_phase x n_electrodes x n_windows x n_trials)

pvalue: array The associated p-values of size : (n_amplitude x n_phase x n_electrodes x n_windows)

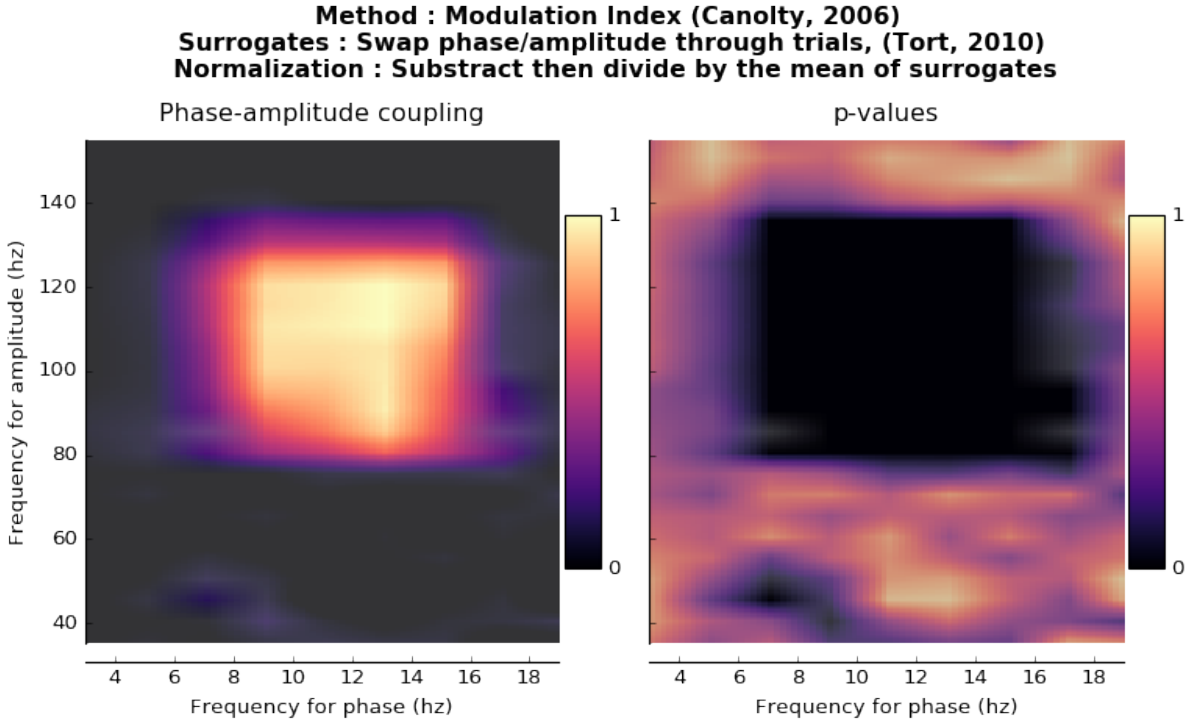

Fig. 4.3: Example of PAC maps for a synthetic coupled signal

4.5.2 Prefered-phase

class `feature.`**`pfdphase`**(*sf, npts, nbins=18, pha_f=[2, 4], pha_meth='hilbert', pha_cycle=3, amp_f=[60, 200], amp_meth='hilbert', amp_cycle=6, window=None, width=None, step=None, time=None, **kwargs*)

Get the preferred phase of a phase-amplitude coupling

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

Kargs:

nbins: integer, optional, [def: 18] Number of bins to binarize the amplitude.

pha_f: tuple/list, optional, [def: [2,4]] List containing the couple of frequency bands for the phase. Example: f=[[2,4], [5,7], [60,250]]

pha_meth: string, optional, [def: 'hilbert'] Method for the phase extraction.

pha_cycle: integer, optional, [def: 3] Number of cycles for filtering the phase.

amp_f: tuple/list, optional, [def: [60,200]] List containing the couple of frequency bands for the amplitude. Each couple can be either a list or a tuple.

amp_meth: string, optional, [def: 'hilbert'] Method for the amplitude extraction.

amp_cycle: integer, optional, [def: 6] Number of cycles for filtering the amplitude.

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] Width and step parameters will be ignored.

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

get (*xpha*, *xamp*, *n_jobs=-1*)
Get the preferred phase

Args:

xpha: array Signal for phase. The shape of xpha should be : (n_electrodes x n_pts x n_trials)

xamp: array Signal for amplitude. The shape of xamp should be : (n_electrodes x n_pts x n_trials)

Kargs:

n_jobs: integer, optional, [def: -1] Control the number of jobs for parallel computing. Use 1, 2, .. depending of your number or cores. -1 for all the cores.

If the same signal is used (example : xpha=x and xamp=x), this mean the program compute a local cfc.

Returns:

pfp: array The preferred phase extracted from the mean of trials of size : (n_amplitude x n_phase x n_electrodes x n_windows)

prf: array The preferred phase extracted from each trial of size : (n_amplitude x n_phase x n_electrodes x n_windows x n_trials)

ambin: array The binarized amplitude of size : (n_amplitude x n_phase x n_electrodes x n_windows x n_bins x n_trials)

pvalue: array The associated p-values of size : (n_amplitude x n_phase x n_electrodes x n_windows)

4.5.3 Phase-locked power

class feature.**PhaseLockedPower** (*sf, npts, f=(2, 200, 10, 5), pha=[8, 13], time=None, baseline=None, norm=None, **powArgs*)
Extract phase-locked power and visualize shifted time-frequency map according to phase peak.

Args:

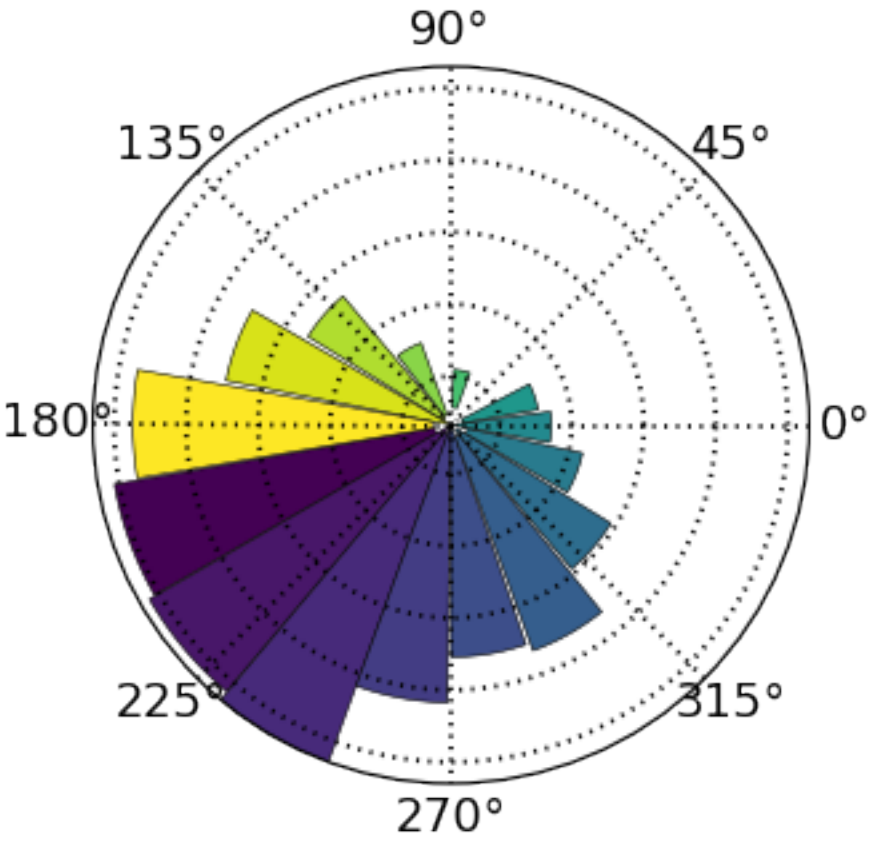

Fig. 4.4: Example of preferred phase for an alpha-gamma coupling in BA6

sf: int Sampling frequency

npts: int Number of points of the time serie

Kargs:

f: tuple/list, optional, [def: (2, 200, 10, 5)] The frequency vector (fstart, fend, fwidth, fstep)

pha: tuple/list, optional, [def: [8, 13]] Frequency for phase.

time: array/list, optional, [def: None] The time vector to use

baseline: tuple/list, optional, [def: None] Location of baseline (in sample)

norm: integer, optional, [def: None]

Normalize method

- 0: No normalisation
- 1: Substraction
- 2: Division
- 3: Substract then divide
- 4: Z-score

powArgs: any supplementar arguments are directly passed to the power function.

get (*x*, *cue*)

Get power phase locked

Args:

x: array Data of shape (npt, ntrials)

cue: integer Cue to align time-frequency maps.

Returns: xpow, xpha, xsig: repectively realigned power, phase and filtered signal

tflockedplot (*xpow*, *sig*, *cmap='viridis'*, *vmin=None*, *vmax=None*, *ylim=None*, *alpha=0.3*, *kind='std'*, *vColor='r'*, *sigcolor='slateblue'*, *fignum=0*)

Plot realigned time-frequency maps.

Args: xpow, sig: output of the get() method. sig can either be the phase or the filtered signal.

Kargs:

cmap: string, optional, [def: 'viridis'] The colormap to use

vmin, vmax: int/float, otpional, [def: None, None] Limits of the colorbar

ylim: tuple/list, optional, [def: None] Limit for the plot of the signal

alpha: float, optional, [def: 0.3] Transparency of deviation/sem

kind: string, optional, [def: 'std'] Choose between 'std' or 'sem' to either display standard deviation or standard error on the mean for the signal plot

vColor: string, optional, [def: 'r'] Color of the vertical line which materialized the choosen cue

sigcolor: string, optional, [def: 'slateblue'] Color of the signal

fignum: integer, optional, [def: 0] Number of the figure

Returns: figure, axes1 (TF plot), axes2 (signal plot), axes3 (colorbar)

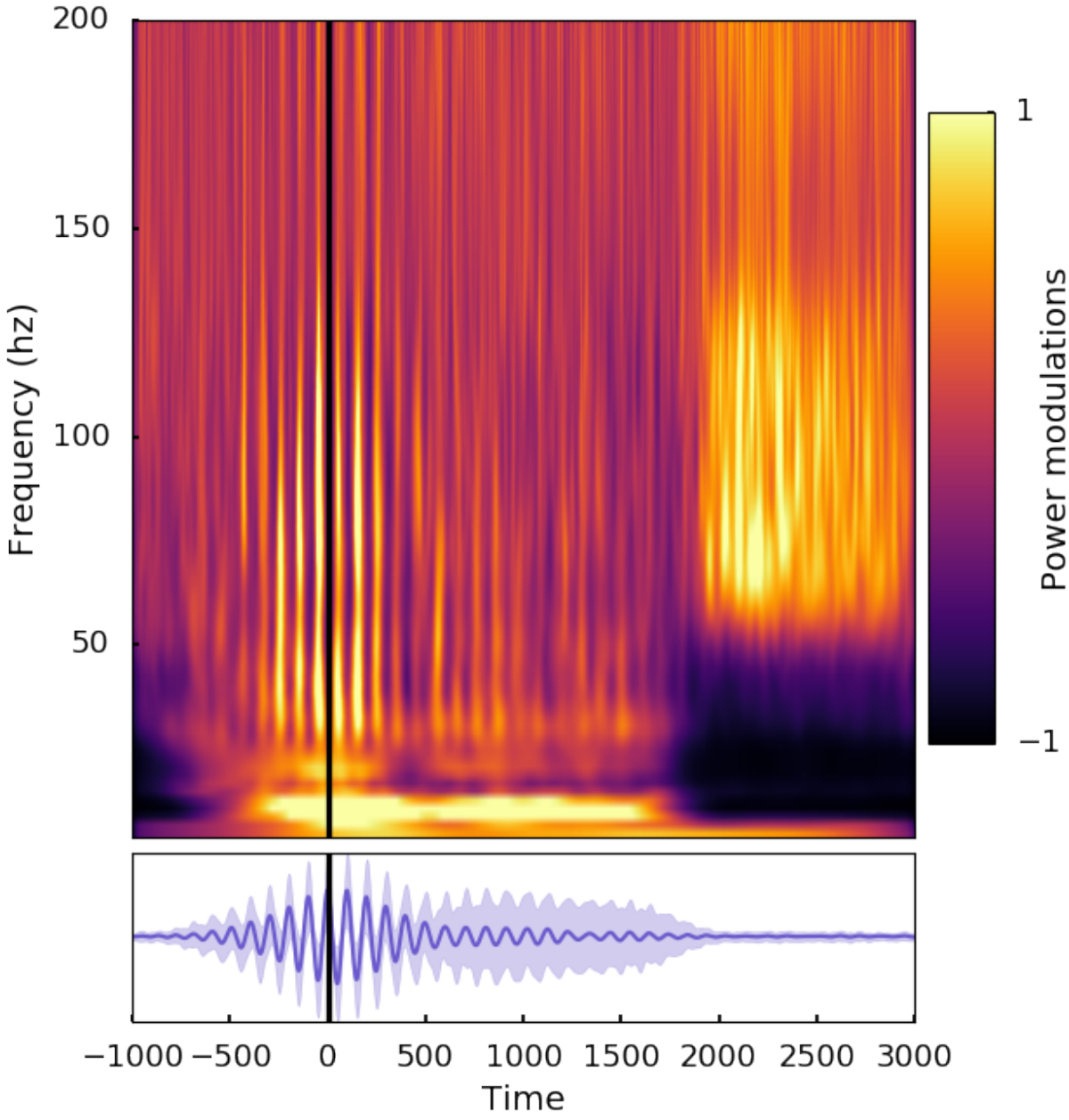

Fig. 4.5: Event Related Phase-Amplitude

4.5.4 Event Related Phase-Amplitude Coupling

class feature.**erpac**(*sf, npts, pha_f=[2, 4], pha_meth='hilbert', pha_cycle=3, amp_f=[60, 200], amp_meth='hilbert', amp_cycle=6, window=None, step=None, width=None, time=None, **kwargs*)

Compute Event Related Phase-Amplitude coupling. See [6]

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

Kargs:

pha_f: tuple/list, optional, [def: [2,4]] List containing the couple of frequency bands for the phase. Example: f=[[2,4], [5,7], [60,250]]

pha_meth: string, optional, [def: 'hilbert'] Method for the phase extraction.

pha_cycle: integer, optional, [def: 3] Number of cycles for filtering the phase.

amp_f: tuple/list, optional, [def: [60,200]] List containing the couple of frequency bands for the amplitude. Each couple can be either a list or a tuple.

amp_meth: string, optional, [def: 'hilbert'] Method for the amplitude extraction.

amp_cycle: integer, optional, [def: 6] Number of cycles for filtering the amplitude.

window: tuple/list/None, optional [def: None] List/tuple: [100,1500] List of list/tuple: [(100,500),(200,4000)] Width and step parameters will be ignored.

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

get(*xpha*, *xamp*, *n_perm=200*, *n_jobs=-1*)

Get the erpac mesure between an xpha and xamp signals.

Args:

xpha: array Signal for phase. The shape of xpha should be : (n_electrodes x n_pts x n_trials)

xamp: array Signal for amplitude. The shape of xamp should be : (n_electrodes x n_pts x n_trials)

Kargs:

n_perm: integer, optional, [def: 200] Number of permutations for normalizing the cfc mesure.

n_jobs: integer, optional, [def: -1] Control the number of jobs for parallel computing. Use 1, 2, .. depending of your number or cores. -1 for all the cores.

If the same signal is used (example : xpha=x and xamp=x), this mean the program compute a local erpac.

Returns:

xerpac: array The erpac mesure of size : (n_amplitude x n_phase x n_electrodes x n_windows)

pvalue: array The associated p-values of size : (n_amplitude x n_phase x n_electrodes x n_windows)

[6] Voytek et al, 2013

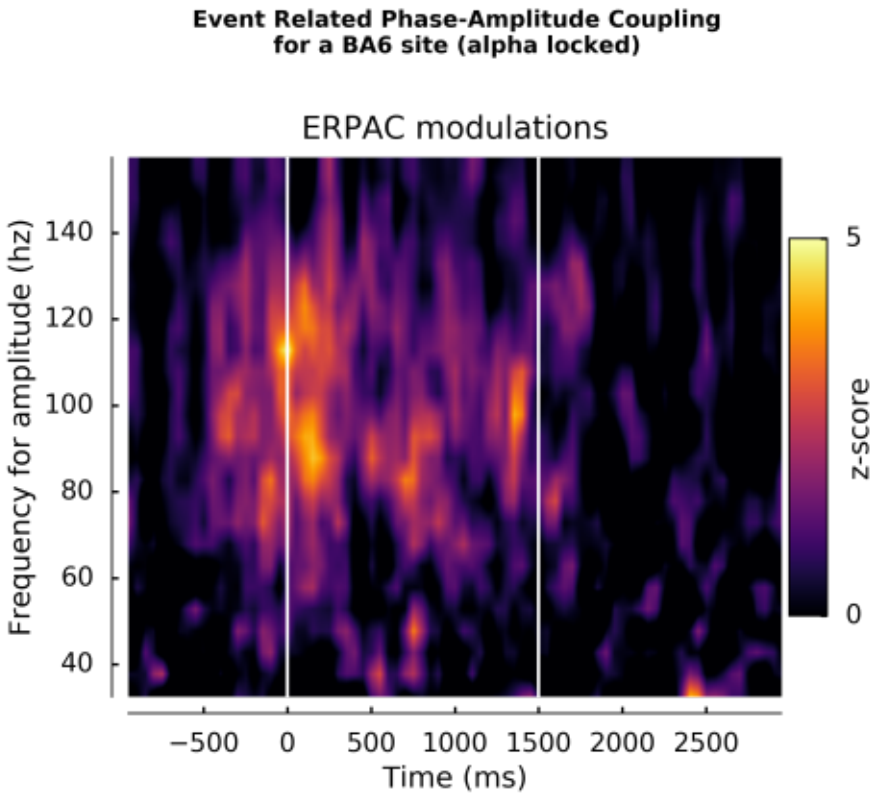

Fig. 4.6: Event Related Phase-Amplitude Coupling

4.5.5 Phase-Locking Value

class feature.**PLV**(*sf, npts, f=[2, 4], method='hilbert', cycle=3, sample=None, time=None, **kwargs*)
Compute the Phase-Locking Value [7]

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

Kargs:

f: tuple/list, optional, [def: [2,4]] List containing the couple of frequency bands for the phase. Example: f=[[2,4], [5,7], [60,250]]

method: string, optional, [def: 'hilbert'] Method for the phase extraction.

cycle: integer, optional, [def: 3] Number of cycles for filtering the phase.

sample: list, optional, [def: None] Select samples in the time series to compute the plv

time: list/array, optional [def: None] Define a specific time vector

amp_cycle: integer, optional, [def: 6] Number of cycles for filtering the amplitude.

get(*xelec1*, *xelec2*, *n_perm=200*, *n_jobs=-1*)
Get Phase-Locking Values for a set of distant sites

Args:

xelec1, xelec2: array PLV will be compute between xelec1 and xelec2. Both matrix contains times-series of each trial pear electrodes. It's not forced that both have the same size but they must have at least the same number of time points (npts) and trials (ntrials). [xelec1] = (n_elec1, npts, ntrials), [xelec2] = (n_elec2, npts, ntrials)

Kargs:

n_perm: int, optional, [def: 200] Number of permutations to estimate the statistical significiancy of the plv mesure

n_jobs: integer, optional, [def: -1] Control the number of jobs for parallel computing. Use 1, 2, .. depending of your number or cores. -1 for all the cores.

Returns:

plv: array The plv mesure for each phase and across electrodes of size: [plv] = (n_pha, n_elec1, n_elec2, n_sample)

pvalues: array The p-values with the same shape of plv

4.6 PSD based features

Those following features are extracted using a Power Spectrum Density (PSD)

4.6.1 Power Spectrum Density

class feature.**PSD**(*sf*, *npts*, *step=None*, *width=None*, *time=None*)
Compute the power spectral density of multiple electrodes.

[7] Lachaux et al, 1999

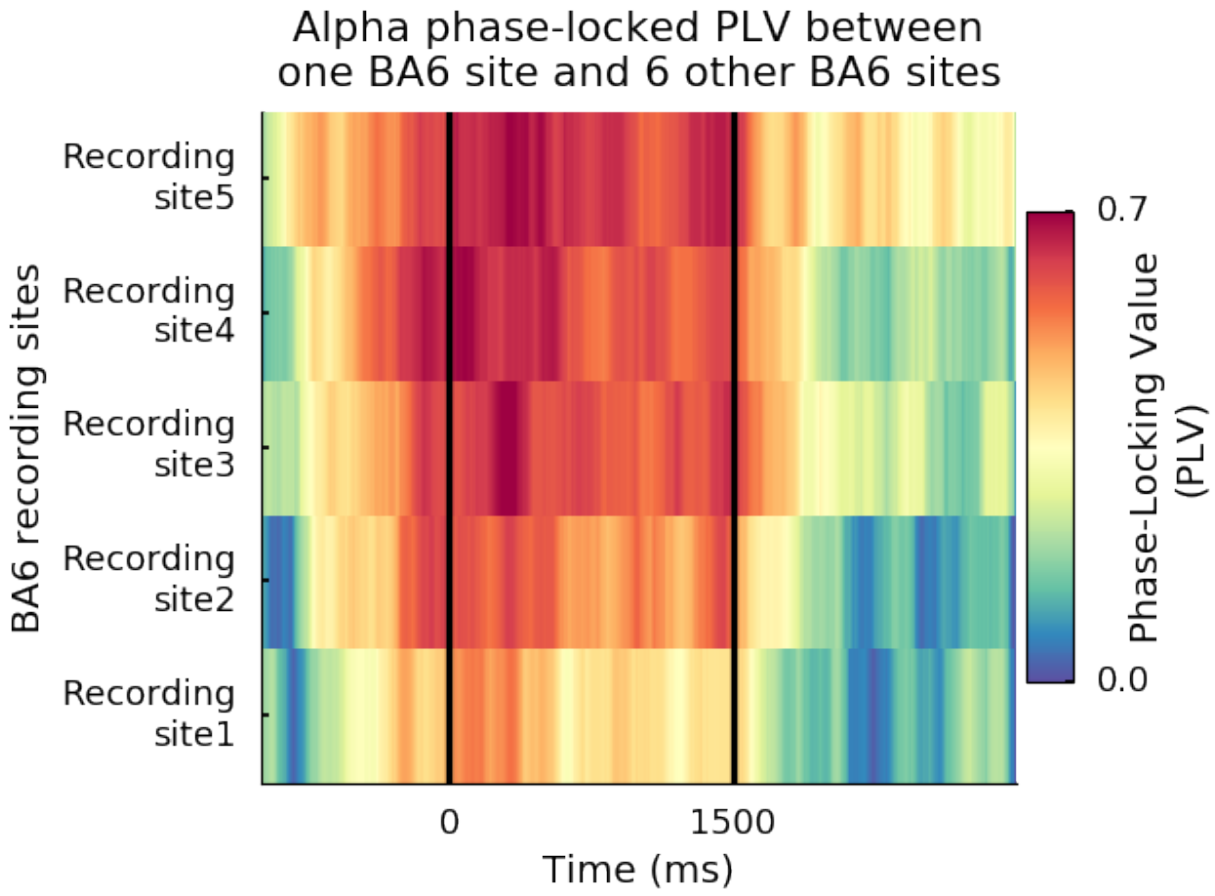

Fig. 4.7: Example of an alpha PLV

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

get (*x*, ***kwargs*)
Get the PSD

Args:

x: array Data to get PSD. x can be a vector of (npts), a matrix (npts, ntrials) or 3D array (nelectrodes, npts, ntrials).

Kargs: kwargs: any supplementar argument are directly passed to the welch function of scipy.

Returns: f: frequency vector of shape (nfce,)

amp: PSD array of shape (nelectrodes, nfce, nwin, ntrials)

4.6.2 Power PSD

class feature.**powerPSD** (*sf, npts, f=[60, 200], step=None, width=None, time=None*)
Compute the power based on psd of multiple electrodes.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

f: tuple/list List containing the couple of frequency bands to extract spectral informations. Alternatively, f can be define with the form f=(fstart, fend, fwidth, fstep) where fstart and fend are the starting and endind frequencies, fwidth and fstep are the width and step of each band.

```
>>> # Specify each band:
>>> f = [ [15, 35], [25, 45], [35, 55], [45, 60], [55, 75] ]
>>> # Second option:
>>> f = (15, 75, 20, 10)
```

get (*x*, ***kwargs*)
Get the PSD power

Args:

x: array Data to get PSD power. x can be a vector of (npts), a matrix (npts, ntrials) or 3D array (nelectrodes, npts, ntrials).

Kargs: kwargs: any supplementar argument are directly passed to the welch function of scipy.

Return: xpow: power array of shape (nfce, nelectrodes, nwin, ntrials)

4.6.3 Spectral entropy

class feature.**SpectralEntropy**(*sf*, *npts*, *step=None*, *width=None*, *time=None*)
Compute the spectral entropy based on psd of multiple electrodes.

Args:

sf: int Sampling frequency

npts: int Number of points of the time serie

width: int, optional [def: None] width of a single window.

step: int, optional [def: None] Each window will be spaced by the "step" value.

time: list/array, optional [def: None] Define a specific time vector

get(*x*, ***kwargs*)
Get the spectral entropy

Args:

x: array Data to get spectral entropy. x can be a vector of (npts), a matrix (npts, ntrials) or 3D array (nelectrodes, npts, ntrials).

Kargs: kwargs: any supplementar argument are directly passed to the welch function of scipy.

Return: xent: spectral entropy of x of shape (nelectrodes, nwin, ntrials)

4.7 Tools

4.7.1 Physiological bands

feat.featools.**bandRef**(*return_as='table'*)
Get the traditionnal physiological frequency bands of interest

Args:

return_as: string, optional, [def: 'table] Say how to return bands. Use either 'table' to get a pandas Dataframe or 'list' to have two list containing bands name and definition.

feat.featools.**findBandName**(*band*)
Find the physiological name of a frequency band

Args:

band: list List of frequency bands

Returns: List of band names

Example:

```
>>> band = [[13, 60], [0, 1], [5, 7]]
>>> findBandName(band)
>>> ['Low-gamma', 'VLFC', 'Theta']
```

feat.featools.**findBandFcy**(*band*)
Find the physiological frequency band for a given name

Args:

band: list List of frequency band name

Returns: List of freqeuncy bands

4.7.2 Cross-frequency vector

feat.featools.**cfcVec**(*pha=(2, 30, 2, 1)*, *amp=(60, 200, 10, 5)*)

Generate cross-frequency coupling vectors.

Kargs:

pha: tuple, optional, [def: (2, 30, 2, 1)] Frequency parameters for phase. Each argument inside the tuple mean (starting fcy, ending fcy, width, step)

amp: tuple, optional, [def: (60, 200, 10, 5)] Frequency parameters for amplitude. Each argument inside the tuple mean (starting fcy, ending fcy, width, step)

Returns:

pVec: array Centered-frequency vector for the phase

aVec: array Centered-frequency vector for the amplitude

pTuple: list List of tuple. Each tuple contain the (starting, ending) frequency for phase.

aTuple: list List of tuple. Each tuple contain the (starting, ending) frequency for amplitude.

4.7.3 Simulation

feat.featools.**cfcRndSignals**(*fPha=2*, *fAmp=100*, *sf=1024*, *ndatasets=10*, *tmax=1*, *chi=0*, *noise=1*, *dPha=0*, *dAmp=0*)

Generate randomly phase-amplitude coupled signals.

Kargs:

fPha: int/float, optional, [def: 2] Frequency for phase

fAmp: int/float, optional, [def: 100] Frequency for amplitude

sf: int, optional, [def: 1024] Sampling frequency

ndatasets [int, optional, [def: 10]] Number of datasets

tmax: int/float (1<=tmax<=3), optional, [def: 1] Length of the time vector. If tmax=2 and sf=1024, the number of time points npts=1024*2=2048

chi: int/float (0<=chi<=1), optional, [def: 0] Amout of coupling. If chi=0, signals of phase and amplitude are strongly coupled.

noise: int/float (1<=noise<=3), optional, [def: 1] Amount of noise

dPha: int/float (0<=dPha<=100), optional, [def: 0] Introduce a random incertitude on the phase frequency. If fPha is 2, and dPha is 50, the frequency for the phase signal will be between : [2-0.5*2, 2+0.5*2]=[1,3]

dAmp: int/float (0<=dAmp<=100), optional, [def: 0] Introduce a random incertitude on the amplitude frequency. If fAmp is 60, and dAmp is 10, the frequency for the amplitude signal will be between : [60-0.1*60, 60+0.1*60]=[54,66]

Return:

data: array The randomly coupled signals. The shape of data will be (ndatasets x npts)

time: array The corresponding time vector

```
from brainpipe.classification import *
```

4.8 Presentation

Ok, let's say you already have extracted features from your neural activity and now, you want to use machine-learning to verify if your features can discriminate some conditions. For example, you want to discriminate conscious versus unconscious people using alpha power on 64 EEG electrodes. Your data can be organized like this:

```
# Consider that conscious data have 150 trials and 130 for unconscious. So if you
# print the shape of both, you'll have :
print(conscious_data.shape, unconscious_data.shape)
# ( (150, 67), (130, 67) )
# Let's build your data matrix by concatenating along the trial dimension:
x = np.concatenate( (conscious_data, unconscious_data), axis=0 )
print('New shape of x: ', x.shape)
# New shape of x: (280, 67)
# Now, build your label vector to indicate to machine learning
# which trial belong to which condition. We are going to use
# 0 for conscious / 1 for unconscious. Finally, the label vector
# will have the same length as the number of trials in x :
y = [0]*conscious_data.shape[0] + [1]*unconscious_data.shape[0]
```

Now, we have the concatenated data and the label vector. To start using machine learning, we need two things:

- a classifier
- a cross-validation

In brainpipe, use defClf() to construct your classifier. Use defCv() to construct the cross-validation. Finally, the classify() function will linked this two objectsin order to classify your conditions.

```
# Define a 50 times 5-folds cross-validation :
cv = defCv(y, cvtype='kfold', rep=50, n_folds=5)
# Define a Random Forest with 200 trees :
clf = defClf(y, clf='rf', n_tree=200, random_state=100)
# Past the two objects inside classify :
clfObj = classify(y, clf=clf, cvtype=cv)
# Evaluate the classifier on data:
da = clfObj.fit(x)
```

4.9 Define a classifier

class classification.**defClf**(*y*, *clf='lda'*, *kern='rbf'*, *n_knn=10*, *n_tree=100*, *priors=False*, ***kwargs*)

Choose a classifier and switch easyly between classifiers implemented in scikit-learn.

Args:

y: array The vector label

clf: int or string, optional, [def: 0] Define a classifier. Use either an integer or a string Choose between:

- 0 / 'lda': Linear Discriminant Analysis (LDA)
- 1 / 'svm': Support Vector Machine (SVM)
- 2 / 'linearsvm' : Linear SVM

- 3 / 'nusvm': Nu SVM
- 4 / 'nb': Naive Bayesian
- 5 / 'knn': k-Nearest Neighbor
- 6 / 'rf': Random Forest
- 7 / 'lr': Logistic Regression
- 8 / 'qda': Quadratic Discriminant Analysis

kern: string, optional, [def: 'rbf'] Kernel of the 'svm' classifier

n_knn: int, optional, [def: 10] Number of neighbors for the 'knn' classifier

n_tree: int, optional, [def: 100] Number of trees for the 'rf' classifier

Kargs: optional arguments. To define other parameters, see the description of scikit-learn.

Return: A scikit-learn classification objects with two supplementar arguments :

- lgStr : long description of the classifier
- shStr : short description of the classifier

4.10 Define a cross-validation

class `classification.`**`defCv`**(*y*, *cvtype='skfold'*, *n_folds=10*, *rndstate=0*, *rep=10*, ***kwargs*)

Choose a cross_validation (CV) and switch easyly between CV implemented in scikit-learn.

Args:

y: array The vector label

kargs:

cvtype: string, optional, [def: skfold] Define a cross_validation. Choose between :

- 'skfold': Stratified k-Fold
- 'kfold': k-fold
- 'sss': Stratified Shuffle Split
- 'ss': Shuffle Split
- 'loo': Leave One Out
- 'lolo': Leave One Label Out

n_folds: integer, optional, [def: 10] Number of folds

rndstate: integer, optional, [def: 0] Define a random state. Usefull to replicate a result

rep: integer, optional, [def: 10] Number of repetitions

kwargs: optional arguments. To define other parameters, see the description of scikit-learn.

Return: A list of scikit-learn cross-validation objects with two supplementar arguments:

- lgStr: long description of the cross_validation
- shStr: short description of the cross_validation

4.11 Classify

class classification.**classify**(*y*, *clf='lda'*, *cvtype='skfold'*, *clfArg={}*, *cvArg={}*)

Define a classification object and apply to classify data. This class can be consider as a centralization of scikit-learn tools, with a few more options.

To classify data, two objects are necessary : - A classifier object (lda, svm, knn...) - A cross-validation object which is used to validate a classification performance. This two objects can either be defined before the classify object with defCv and defClf, or they can be directly defined inside the classify class.

Args:

y: array The vector label

Kwargs:

clf: int / string / classifier object, optional, [def: 0] Define a classifier. If clf is an integer or a string, the classifier will be defined inside classify. Otherwise, it is possible to define a classifier before with defClf and past it in clf.

cvtype: string / cross-validation object, optional, [def: 'skfold'] Define a cross-validation. If cvtype is a string, the cross-validation will be defined inside classify. Otherwise, it is possible to define a cross-validation before with defCv and past it in cvtype.

clfArg: dictionnary, optional, [def: {}] This dictionnary can be used to define supplementar arguments for the classifier. See the documentation of defClf.

cvArg: dictionnary, optional, [def: {}] This dictionnary can be used to define supplementar arguments for the cross-validation. See the documentation of defCv.

Example:

```
>>> # 1) Define a classifier and a cross-validation before classify():
>>> # Define a 50 times 5-folds cross-validation :
>>> cv = defCv(y, cvtype='kfold', rep=50, n_folds=5)
>>> # Define a Random Forest with 200 trees :
>>> clf = defClf(y, clf='rf', n_tree=200, random_state=100)
>>> # Past the two objects inside classify :
>>> clfObj = classify(y, clf=clf, cvtype=cv)
```

```
>>> # 2) Define a classifier and a cross-validation inside classify():
>>> clfObj = classify(y, clf = 'rf', cvtype = 'kfold',
>>>        clfArg = {'n_tree':200, 'random_state':100},
>>>                  cvArg = {'rep':50, 'n_folds':5})
>>> # 1) and 2) are equivalent. Then use clfObj.fit() to classify data.
```

cm(*normalize=True*)

Get the confusion matrix of each feature.

Kargs:

normalize: bool, optional, [def: True] Normalize or not the confusion matrix

update: bool, optional, [def: True] If update is True, the data will be re-classified. But, if update is set to False, and if the methods .fit() or .fit_stat() have been run before, the data won't we re-classified. Instead, the labels previously found will be used to get confusion matrix.

Return:

CM: array Array of confusion matrix of shape (n_features x n_class x n_class)

fit (*x*, *mf=False*, *center=False*, *grp=None*, *method='bino'*, *n_perm=200*, *rndstate=0*, *n_jobs=-1*)
Apply the classification and cross-validation objects to the array x.

Args:

x: array Data to classify. Consider that x.shape = (N, M), N is the number of trials (which should be the length of y). M, the number of colums, is a supplementar dimension for classifying data. If M = 1, the data is consider as a single feature. If M > 1, use the parameter mf to say if x should be consider as a single feature (mf=False) or multi-features (mf=True)

Kargs:

mf: bool, optional, [def: False] If mf=False, the returned decoding accuracy (da) will have a shape of (1, rep) where rep, is the number of repetitions. This mean that all the features are used together. If mf=True, da.shape = (M, rep), where M is the number of columns of x.

center: optional, bool, [def: False] Normalize fatures with a zero mean by substracting then dividing by the mean. The center parameter should be set to True if the classifier is a svm.

grp: array, optional, [def: None] If mf=True, the grp parameter allow to define group of features. If x.shape = (N, 5) and grp=np.array([0,0,1,2,1]), this mean that 3 groups of features will be considered : (0,1,2)

method: string, optional, [def: 'bino'] Four methods are implemented to test the statistical significance of the decoding accuracy :

- 'bino': binomial test
- 'label_rnd': randomly shuffle the labels
- 'full_rnd': randomly shuffle the whole array x
- 'intra_rnd': randomly shuffle x inside each class and each feature

Methods 2, 3 and 4 are based on permutations. The method 2 and 3 should provide similar results. But 4 should be more conservative.

n_perm: integer, optional, [def: 200] Number of permutations for the methods 2, 3 and 4

rndstate: integer, optional, [def: 0] Fix the random state of the machine. Usefull to reproduce results.

n_jobs: integer, optional, [def: -1] Control the number of jobs to cumpute the decoding accuracy. If n_jobs = -1, all the jobs are used.

Return:

da: array The decoding accuracy of shape n_repetitions x n_features

pvalue: array Array of associated pvalue of shape n_features

daPerm: array Array of all the decodings obtained for each permutations of shape n_perm x n_features

4.12 Leave p-subjects out

class classification.**LeavePSubjectOut** (*y*, *nsuj*, *pout=1*, *clf='lda'*, ***clfArg*)
Leave p-subbject out cross-validation

Args:

y: list List of label vectors for each subject.

nsuj: int Number of subjects

Kargs:

pout: int Number of subjects to leave out for testing. If pout=1, this is a leave one-subject out

clf: int / string / classifier object, optional, [def: 0] Define a classifier. If clf is an integer or a string, the classifier will be defined inside classify. Otherwise, it is possible to define a classifier before with defClf and past it in clf.

clfArg: supplementar arguments This dictionnary can be used to define supplementar arguments for the classifier. See the documentation of defClf.

change_clf(*clf='lda'*, ***clfArg*)
Change the classifier

fit(*x*, *mf=False*, *center=False*, *grp=None*, *method='bino'*, *n_perm=200*, *rndstate=0*, *n_jobs=-1*)
Apply the classification and cross-validation objects to the array x.

Args:

x: list List of dataset for each subject. All the dataset in the list should have the same number of columns but the number of lines could be diffrent for each subject and must correspond to the same number of lines each each label vector of y.

Kargs:

mf: bool, optional, [def: False] If mf=False, the returned decoding accuracy (da) will have a shape of (1, rep) where rep, is the number of repetitions. This mean that all the features are used together. If mf=True, da.shape = (M, rep), where M is the number of columns of x.

center: optional, bool, [def: False] Normalize fatures with a zero mean by substracting then dividing by the mean. The center parameter should be set to True if the classifier is a svm.

grp: array, optional, [def: None] If mf=True, the grp parameter allow to define group of features. If x.shape = (N, 5) and grp=np.array([0,0,1,2,1]), this mean that 3 groups of features will be considered : (0,1,2)

method: string, optional, [def: 'bino'] Four methods are implemented to test the statistical significance of the decoding accuracy :

- 'bino': binomial test
- 'label_rnd': randomly shuffle the labels

Methods 2 and 3 are based on permutations. They should provide similar results. But 4 should be more conservative.

n_perm: integer, optional, [def: 200] Number of permutations for the methods 2, 3 and 4

rndstate: integer, optional, [def: 0] Fix the random state of the machine. Usefull to reproduce results.

n_jobs: integer, optional, [def: -1] Control the number of jobs to cumpute the decoding accuracy. If n_jobs = -1, all the jobs are used.

Return:

da: array The decoding accuracy of shape n_repetitions x n_features

pvalue: array Array of associated pvalue of shape n_features

daPerm: array Array of all the decodings obtained for each permutations of shape n_perm x n_features

4.13 Generalization

class classification.**generalization**(*time*, *y*, *x*, *clf='lda'*, *cvtype=None*, *clfArg={}*, *cvArg={}*)

Generalize the decoding performance of features. The generalization consist of training and testing at diffrents moments. The use is to see if a feature is consistent and performant in diffrents period of time.

Args:

time: array/list The time vector of dimension npts

y: array The vector label of dimension ntrials

x: array The data to generalize. If x is a 2D array, the dimension of x should be (ntrials, npts). If x is 3D array, the third dimension is consider as multi-features. This can be usefull to do time generalization in multi-features.

Kargs:

clf: int / string / classifier object, optional, [def: 0] Define a classifier. If clf is an integer or a string, the classifier will be defined inside classify. Otherwise, it is possible to define a classifier before with defClf and past it in clf.

cvtype: string / cross-validation object, optional, [def: None] Define a cross-validation. If cvtype is None, the diagonal of the matrix of decoding accuracy will be set at zero. If cvtype is defined, a cross-validation will be performed on the diagonal. If cvtype is a string, the cross-validation will be defined inside classify. Otherwise, it is possible to define a cross-validation before with defCv and past it in cvtype.

clfArg: dictionnary, optional, [def: {}] This dictionnary can be used to define supplementar arguments for the classifier. See the documentation of defClf.

cvArg: dictionnary, optional, [def: {}] This dictionnary can be used to define supplementar arguments for the cross-validation. See the documentation of defCv.

Return: An array of dimension (npts, npts) containing the decoding accuracy. The y axis is the training time and the x axis is the testing time (also known as "generalization time")

4.14 Multi-features

class classification.**mf**(*y*, *Id='0'*, *p=0.05*, *n_perm=200*, *stat='bino'*, *threshold=None*, *nbest=10*, *direction='forward'*, *occurence='i%'*, *clfIn={'clf': 'lda'}*, *clfOut={'clf': 'lda'}*, *cvIn={'rep': 1, 'n_folds': 10, 'cvtype': 'skfold'}*, *cvOut={'rep': 10, 'n_folds': 10, 'cvtype': 'skfold'}*)

Compute multi-features (mf) with the possibility of using methods in cascade and run the mf on particular groups.

Args:

y: array-like The target variable to try to predict in the case of supervised learning

Kargs:

Id: string, optional, [def: '0'] Use this parameter to define a cascade of methods. Here is the list of the current implemented methods:

- '0': No selection. All the features are used
- '1': Select <p significant features using either a binomial law or permutations
- '2': select 'nbest' features

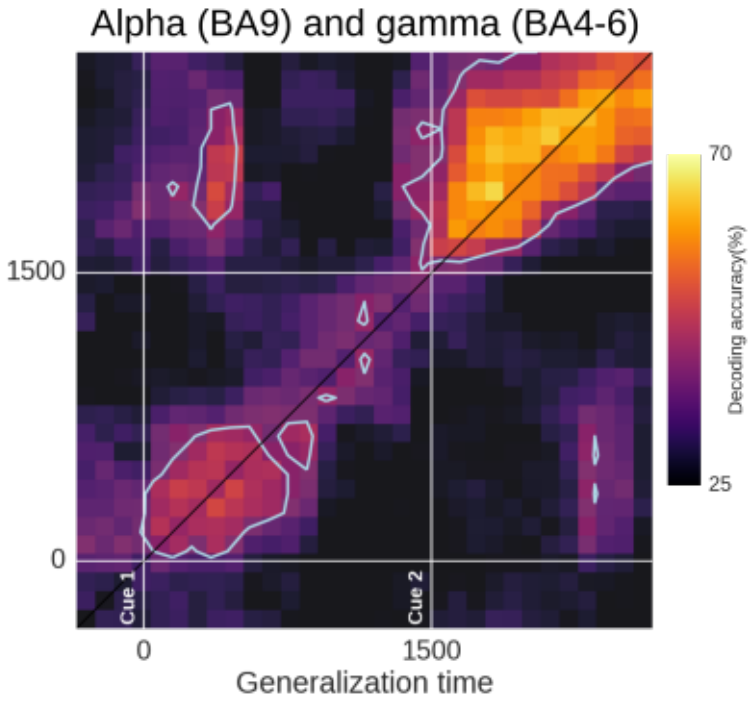

Fig. 4.8: Time-generalization using two features (alpha and gamma power)

- '3': use 'forward'/'backward'/'exhaustive' to select features

If is for example Id='12', the program will first select significants features, then, on this subset, it will find the nbest features. All the methods can be serialized.

p: float, optional, [def: 0.05] The pvalue to select features for the Id='1' method

n_perm: integer, optional, [def: 200] Number of permutations for the Id='1' method

stat [string, optional, [def: 'bino']] Statisical test for selecting features for the Id='1' method. Choose between:

- 'bino': binomial test
- 'label_rnd': randomly shuffle the labels
- 'full_rnd': randomly shuffle the whole array x
- 'intra_rnd': randomly shuffle x inside each class and each feature

Methods 2, 3 and 4 are based on permutations. The method 2 and 3 should provide similar results. But 4 should be more conservative.

threshold: integer/float, optional, [def: None] Define a decoding accuracy for thresholding features. equivalent to the p parameter.

nbest: integer, optional, [def: 10] For the Id='2', use this parameter to control the number of features to select. If nbest=10, the program will classify each feature and then select the 10 best of them.

direction: string, optional, [def: 'forward'] For the method Id='3', use:

- 'forward'
- 'backward'
- 'exhaustive'

to control the direction of the feature selection.

occurence: string, optional, [def: 'i%'] Use this parameter to modify the way of visualizing the occurence of each feature apparition. Choose between :

- '%' : in percentage (float)
- 'i%' : in integer percentage (int)
- 'c' : count (= number of times the feature has been selected)

clfIn // clfOut [dictionnary, optional] Use those dictionnaries to control the classifier to use.

- clfIn : the classifier use for the training [def: LDA]
- clfOut : the classifier use for the testing [def: LDA]

To have more controlable classifiers, see the defClf() class inside the classification module.

cvIn // cvOut [dictionnary, optional] Use those dictionnaries to control the cross-validations (cv) to use.

- **cvIn** [the cv to use for the training [def: 1 time stratified] 10-folds]
- **cvOut** [the more extern cv, to separate training and testing and] to avoid over-fitting [def: 10 time stratified 10-folds]

To have more controlable cross-validation, see the defCv() class inside the classification module.

Return A multi-features object with a fit() method to apply to model to the data.

fit (*x*, *grp=[]*, *center=False*, *combine=False*, *grpas='single'*, *grplen=[]*, *display=True*, *n_jobs=-1*)
Run the model on the matrix of features x

Args:

x: array-like The features. Dimension [n trials x n features]

Kargs:

grp: list of strings, optional, [def: []] Group features by using a list of strings. The length of grp must be the same as the number of features. If grp is not empty, the program will run the feature selection inside each group.

center: optional, bool, [def: False] Normalize fatures with a zero mean by substracting then dividing by the mean. The center parameter should be set to True if the classifier is a svm.

combine: boolean, optional, [def: False] If a group of features is specified using the grp parameter, combine give the access of combining or not groups. For example, if there is three unique groups, combining them will compute the mf model on each combination : [[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]

grpas: string, optional, [def: 'single'] Specify how to consider features inside each group. If the parameter grpas ("group as") is:

- 'single': inside each combination of group, the features are considered as independant.
- 'group': inside each combination of group, the features are going to be associated. So the mf model will search to add a one by one feature, but it will add groups of features.

grplen: list, optional, [def: []] Control the number of combinations by specifying the number of elements to associate. If there is three unique groups, all possible combinations are : [[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]] but if grplen is specify, for example grplen=[1,3], this will consider combinations of groups only with a length of 1 and 3 and remove combinations of 2 elements: [[1],[2],[3],[1,2,3]]

display: boolean, optional, [def: True] Display informations for each step of the mf selection. If n_jobs is -1, it is advise to set the display to False.

n_jobs: integer, optional, [def: -1] Control the number of jobs to cumpute the decoding accuracy. If n_jobs=-1, all the jobs are used.

Returns:

da: list The decoding accuracy (da) for each group with the selected number of repetitions, which by default is set to 10 (see : cvOut // rep)

prob: list The appearance probability of each feature. The size of prob is the same as da.

groupinfo: pandas Dataframe Dataframe to resume the mf feature selection.

class classification.**MFpipe**(*y*, *cv*, *random_state=0*)

Define a pipeline of multi-features

Args:

y: ndarray Vector label

cv: sklearn.cross_validation, optional, (def: default) An external cross-validation to split in training and testing to validate the pipeline without overfiting.

random_state: ineteger, optional, (def: 0) Fix the random state of the machine for reproducibility.

Return Multi-features pipeline object

custom_pipeline(*pipeline=None*, *grid=None*)

Send a custom pipeline

Kargs:

pipeline: sklearn.Pipeline, optional, (def: None) The pipeline to use

grid: sklearn.GridCv, optional, (def: None) A grid for parameters optimisation

default_pipeline(*name, n_pca=10, n_best=10, lda_shrink=10, svm_C=10, svm_gamma=10, fdr_alpha=[0.05], fpr_alpha=[0.05]*)

Use a default combination of parameters for building a pipeline

Args:

name: string The string for building a default pipeline (see examples below)

Kargs:

n_pca: integer, optional, (def: 10) The number of components to search

n_best: integer, optional, (def: 10) Number of best features to consider using a statistical method

lda_shrink: integer, optional, (def: 10) Fit optimisation parameter for the lda

svm_C/svm_gamma: integer, optional, (def: 10/10) Parameters to optimize for the svm

fdr/fpr_alpha: list, optional, (def: [0.05]) List of float for selecting features using a fdr or fpr

Examples:

```
>>> # Basic classifiers :
>>> name = 'lda' # or name = 'svm_linear' for a linear SVM
>>> # Combine a classifier with a feature selection method :
>>> name = 'lda_fdr_fpr_kbest_pca'
>>> # The method above will use an LDA for the features evaluation
>>> # and will combine a FDR, FPR, k-Best and pca feature seelction.
>>> # Now we can combine with classifier optimisation :
```

```
>>> name = 'lda_optimized_pca' # will try to optimize an LDA with a pca
>>> name = 'svm_kernel_C_gamma_kbest' # optimize a SVM by trying
>>> # diffrent kernels (linear/RBF), and optimize C and gamma parameters
>>> # combine with a k-Best features selection.
```

fit (*x*, *name=None*, *rep=1*, *n_iter=5*, *n_jobs=1*, *verbose=0*)
Apply the pipeline.

Args:

x: ndarray Array of features organize as (n_trials, n_features)

Kargs:

name: ndarray, optional, (def: None) Array of string with the same length as the number of features

rep: integer, optional, (def: 1) Number of repetitions for the whole pipeline.

n_iter: integer, optional, (def: 5) Number of iterations in order to find the best set of parameters

n_jobs: integer, optional, (def: 1) Number of jobs for parallel computing. Use -1 for all jobs.

verbose: integer, optional, (def: 0) Control displaying state

Return da: the final vector of decoding accuracy of shape (n_repetitions,)

get_grid (*n_splits=3*, *n_jobs=1*)
Return the cross-validated grid search object

Kargs:

n_splits: int, optional, (def: 3) The number of folds for the cross-validation

n_jobs: int, optional, (def: 1) Number of jobs to use for parallel processing

Return: grid: a sklearn.GridSearchCV object with the defined pipeline

selected_features ()
Get the number of times a feature was selected

```
from brainpipe.statistics import *
```

- *Binomial*
- *Permutations*
- *Multiple-comparisons*
- *Circular statistics toolbox*

4.15 Binomial

statistics.**bino_da2p** (*y*, *da*)
For a given vector label, get p-values of a decoding accuracy using the binomial law.

Args:

y [array] The vector label

da: int / float / list /array [0 <= da <= 100] The decoding accuracy array. Ex : da = [75, 33, 25, 17].

Return:

p: ndarray The p-value associate to each decoding accuracy

statistics.**bino_p2da**(*y*, *p*)
For a given vector label, get the decoding accuracy of p-values using the binomial law.

Args:

y: array The vector label

p: int / float / list / array [0 <= p < 1] p-value. Ex : p = [0.05, 0.01, 0.001, 0.00001]

Return:

da: ndarray The decoding accuracy associate to each p-value

statistics.**bino_signifeat**(*feat*, *th*)
Get significants features.

Args:

feat: array Array containing either decoding accuracy (da) either p-values

th: int / float The threshold in order to find significants features.

Return:

index: array The index corresponding to significants features

signi_feat: array The significants features

4.16 Permutations

4.16.1 Evaluation

statistics.**perm_2pvalue**(*data*, *perm*, *n_perm*, *threshold=None*, *tail=2*)
Get the associated p-value of a permutation distribution

Arg:

data: array Array of real data. The shape must be (d1, d2, ..., dn)

perm: array Array of permutations. The shape must be (n_perm, d1, d2, ..., dn)

n_perm: int Number of permutations

Kargs:

threshold: int / float, optional, [def: None] Every values upper to threshold are going to be set to 1.

tail: int, optional, [def: 2] Define if the calculation of p-value must take into account one or two tails of the permutation distribution

Return:

pvalue [array] Array of associated p-values

statistics.**perm_metric**(*metric*)
Get a metric for permutation normalization.

Args: metric: string/function

- None: compare directly values without transformation
- 'm_center': (A-B)/mean(B) transformation
- 'm_zscore': (A-B)/std(B) transformation

- 'm_minus': (A-B) transformation
- function: user defined function [def myfcn(A, B): return array_like]

statistics.**perm_pvalue2level**(*perm*, *p=0.05*, *maxst=False*)
Get level from which you can consider that it's p-significant;

Args:

perm: Array of permutations of shape (n_perm, d1, d2, ..., dn)

Kargs:

p: float, optional, [def: 0.05] p-value to search in permutation distribution

maxst: bool, optional, [def: False] Correct permutations with maximum statistics

Return:

level: float The level from which you can consider your results as p-significants using permutations

4.16.2 Generate

statistics.**perm_rndDatasets**(*mu=0*, *sigma=1*, *dmu=0.1*, *dsigma=0.1*, *size=(5, 5)*, *rndstate=0*)
Generate data and permutations uniformly distributed.

Kargs:

mu: int/float Center of the distribution

sigma: int/float Deviation of the distribution

dmu: int/float Introduce a shift to the mean of data

dsigma: int/float Introduce a shift to the deviation of data

size: tuple Size of the generated data and permutations (d1, d2, ..., d3)

rndstate: int Fix the random state of the machine

Returns:

data: array Simulated data of shape (n_perm, d1, d2, ..., d3)

perm: array Simulated permutations of shape (n_perm, d1, d2, ..., d3)

statistics.**perm_swap**(*a*, *b*, *n_perm=200*, *axis=-1*, *rndstate=0*)
Permute values between two arrays and generate a number of permutations.

Args:

a, b: ndarray Array to swap values. If axis=-1, the shape of a and b could be diffrent. If axis is not -1, the shape of a and b could be diffrent along the specified axis, but must be the same on all other axis.

n_pem: int Number of permutations

Kargs:

axis: int, optional, [def: -1] Axis for swapping values. If axis is -1, this mean that all values across all dimensions are going to be swap.

rndstate: int Fix the random state of the machine

Return:

ash, bsh: array Swapped arrays

statistics.**perm_rep**(*x*, *n_perm*)
Repeat a ndarray x n_perm times

Args:

x: array Data to repeat of shape (d1, d2, ..., d3)

n_perm: int Number of permutations

Returns:

xrep: array Repeated data of shape (n_perm, d1, d2, ..., d3)

4.17 Multiple-comparisons

4.17.1 Bonferroni

statistics.**bonferroni**(*p*, *axis=-1*)
Bonferroni correction

Args:

p: array Array of p-values

Kargs:

axis: int, optional, [def: -1] Axis to apply the Bonferroni correction. If axis is -1, the correction is applied through all dimensions.

Return: Corrected pvalues

4.17.2 False Discovery Rate (FDR)

statistics.**fdr**(*p*, *q*)
False Discovery Rate correction

Args:

p: array Array of p-values

q: float Thresholding p-value

Return:

pcorr: array Corrected p-values

4.17.3 Maximum statistic

statistics.**maxstat**(*perm*, *axis=-1*)
Correction by the maximum statistic

Args:

perm: array The permutations.

axis: integer, optional, [def: -1] Use -1 to correct through all dimensions. Otherwise, use d1, d2, ... or dn to correct through a specific dimension.

Kargs:

permR: array The re-aranged permutations according to the selectionned axis. Then use perm_2pvalues to get the p-value according to this distribution.

4.18 Circular statistics toolbox

Python adaptation of the Matlab toolbox (Berens et al, 2009)

`statistics.`**`circ_corrcc`**(*alpha*, *x*)

Correlation coefficient between one circular and one linear random variable.

Args:

alpha: vector Sample of angles in radians

x: vector Sample of linear random variable

Returns:

rho: float Correlation coefficient

pval: float p-value

Code taken from the Circular Statistics Toolbox for Matlab By Philipp Berens, 2009 Python adaptation by Etienne Combrisson

`statistics.`**`circ_r`**(*alpha*, *w=None*, *d=0*, *axis=0*)

Computes mean resultant vector length for circular data.

Args:

alpha: array Sample of angles in radians

Kargs:

w: array, optional, [def: None] Number of incidences in case of binned angle data

d: radians, optional, [def: 0] Spacing of bin centers for binned data, if supplied correction factor is used to correct for bias in estimation of r

axis: int, optional, [def: 0] Compute along this dimension

Return: r: mean resultant length

Code taken from the Circular Statistics Toolbox for Matlab By Philipp Berens, 2009 Python adaptation by Etienne Combrisson

`statistics.`**`circ_rtest`**(*alpha*, *w=None*, *d=0*)

Computes Rayleigh test for non-uniformity of circular data. H0: the population is uniformly distributed around the circle HA: the populatoin is not distributed uniformly around the circle Assumption: the distribution has maximally one mode and the data is sampled from a von Mises distribution!

Args:

alpha: array Sample of angles in radians

Kargs:

w: array, optional, [def: None] Number of incidences in case of binned angle data

d: radians, optional, [def: 0] Spacing of bin centers for binned data, if supplied correction factor is used to correct for bias in estimation of r

Code taken from the Circular Statistics Toolbox for Matlab By Philipp Berens, 2009 Python adaptation by Etienne Combrisson

```
from brainpipe.visual import *
```


4.19 1-D graphics

4.19.1 Border plot

class visual.**BorderPlot**(*time*, *x*, *y=None*, *kind='sem'*, *color=''*, *alpha=0.2*, *linewidth=2*, *legend=''*, *ncol=1*, ***kwargs*)

Plot a signal with it associated deviation. The function plot the mean of the signal, and the deviation (std) or standard error on the mean (sem) in transparency.

Args:

time: array/limit The time vector of the plot (len(time)=N)

x: numpy array The signal to plot. One dimension of x must be the length of time N. The other dimension will be consider to define the deviation. For example, x.shape = (N, M)

Kargs:

y: numpy array, optional, [def: None] Label vector to separate the x signal in diffrent classes. The length of y must be M. If no y is specified, the deviation will be computed for the entire array x. If y is composed with integers Example: y = np.array([1,1,1,1,2,2,2,2]), the function will geneate as many curve as the number of unique classes in y. In this case, two curves are going to be considered.

kind: string, optional, [def: 'sem'] Choose between 'std' for standard deviation and 'sem', standard error on the mean (wich is: std(x)/sqrt(N-1))

color: string or list of strings, optional Specify the color of each curve. The length of color must be the same as the length of unique classes in y.

alpha: int/float, optional [def: 0.2] Control the transparency of the deviation.

linewidth: int/float, optional, [def: 2] Control the width of the mean curve.

legend: string or list of strings, optional, [def: ''] Specify the label of each curve and generate a legend. The length of legend must be the same as the length of unique classes in y.

ncol: integer, optional, [def: 1] Number of colums for the legend

kwargs: Supplementar arguments to control each suplot: title, xlabel, ylabel (which can be list for each subplot) xlim, ylim, xticks, yticks, xticklabels, yticklabels, style.

Return: The axes of the plot.

4.19.2 p-value plot

visual.**addPval**(*ax*, *pval*, *y=0*, *x=None*, *p=0.05*, *minsucc=1*, *color='b'*, *shape='-'*, *lw=2*, ***kwargs*)

Add significants p-value to an existing plot

Args:

ax: matplotlib axes The axes to add lines. Use for example plt.gca()

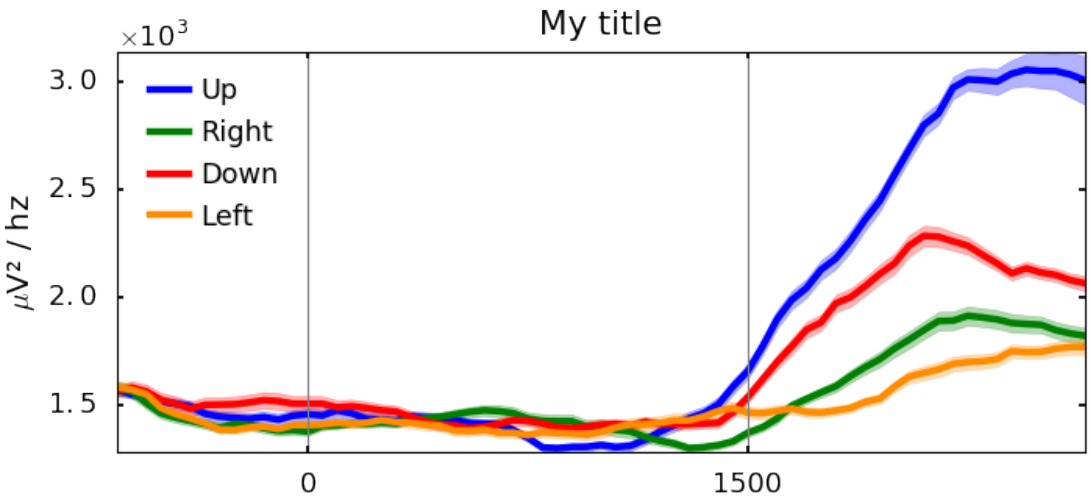

Fig. 4.9: Border plot example

pval: vector Vector of pvalues

Kargs:

y: int/float The y location of your p-values

x: vector x vector of the plot. Must have the same size as pval

p: float p-value threshold to plot

minsucc: int Minimum number of successive significants p-values

color: string Color of th p-value line

shape: string Shape of th p-value line

lw: int Linewidth of th p-value line

kwargs: Any supplementar arguments are passed to the plt.plot() function

Return: ax: updated matplotlib axes

4.19.3 Continuous color

class `visual.`**`continuouscol`**(*ax*, *y*, *x=None*, *color=None*, *cmap='inferno'*, *pltargs={}*, ***kwargs*)
Plot signal with continuous color

Args:

ax: matplotlib axes The axes to add lines. Use for example plt.gca()

y: vector Vector to plot

Kargs:

x: vector, optional, [def: None] Values on the x-axis. x should have the same length as y. By default, x-values are 0, 1, ..., len(y)

color: vector, optional, [def: None] Values to colorize the line. color should have the same length as y.

cmap: string, optional, [def: 'inferno'] The name of the colormap to use

pltargs: dict, optional, [def: {}] Arguments to pass to the LineCollection() function of matplotlib

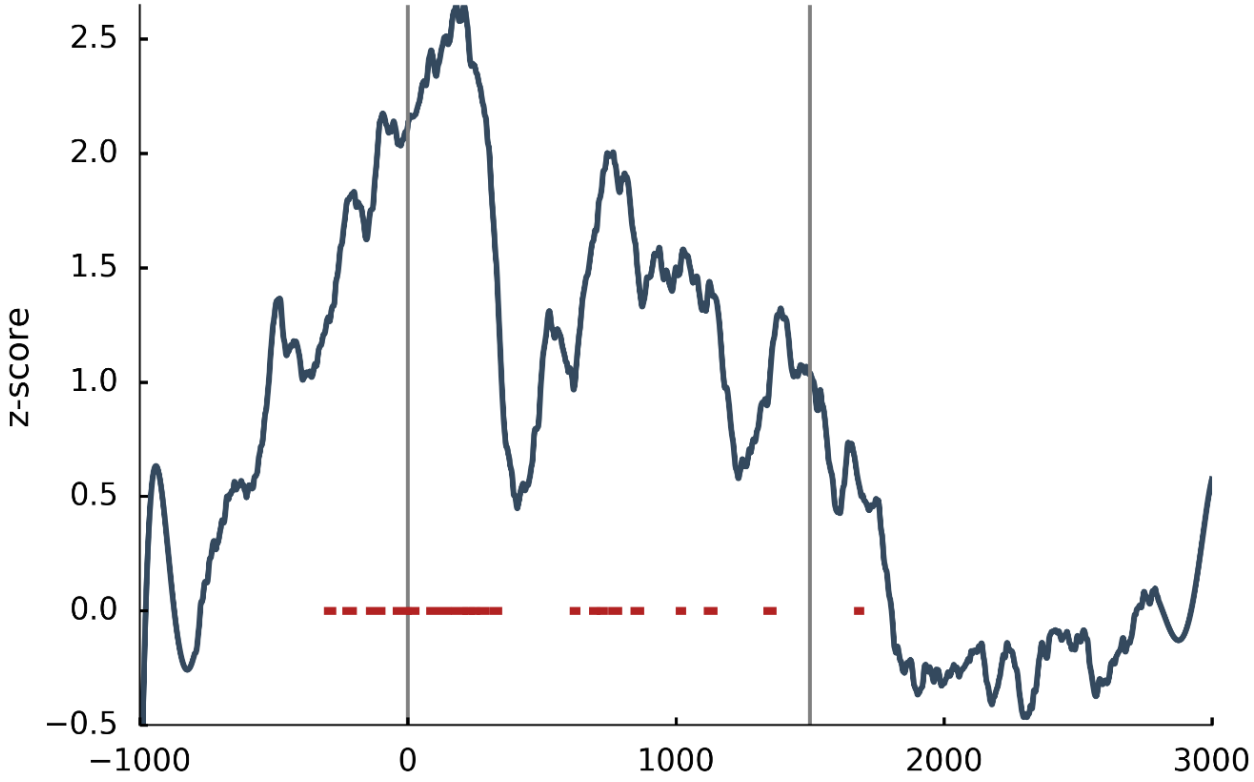

Fig. 4.10: Add p-values to an existing plot

kwargs: Supplementar arguments to control each suplot: title, xlabel, ylabel (which can be list for each subplot) xlim, ylim, xticks, yticks, xticklabels, yticklabels, style.

4.20 1-D or 2-D graphics

4.20.1 Add lines

class visual.**addLines**(*ax*, *vLines=[]*, *vColor=None*, *vShape=None*, *vWidth=None*, *hLines=[]*, *hColor=None*, *hWidth=None*, *hShape=None*)

Add vertical and horizontal lines to an existing plot.

Args:

ax: matplotlib axes The axes to add lines. USe for example plt.gca()

Kargs:

vLines: list, [def: []] Define vertical lines. vLines should be a list of int/float

vColor: list of strings, [def: ['gray']] Control the color of the vertical lines. The length of the vColor list must be the same as the length of vLines

vShape: list of strings, [def: ['–']] Control the shape of the vertical lines. The length of the vShape list must be the same as the length of vLines

hLines: list, [def: []] Define horizontal lines. hLines should be a list of int/float

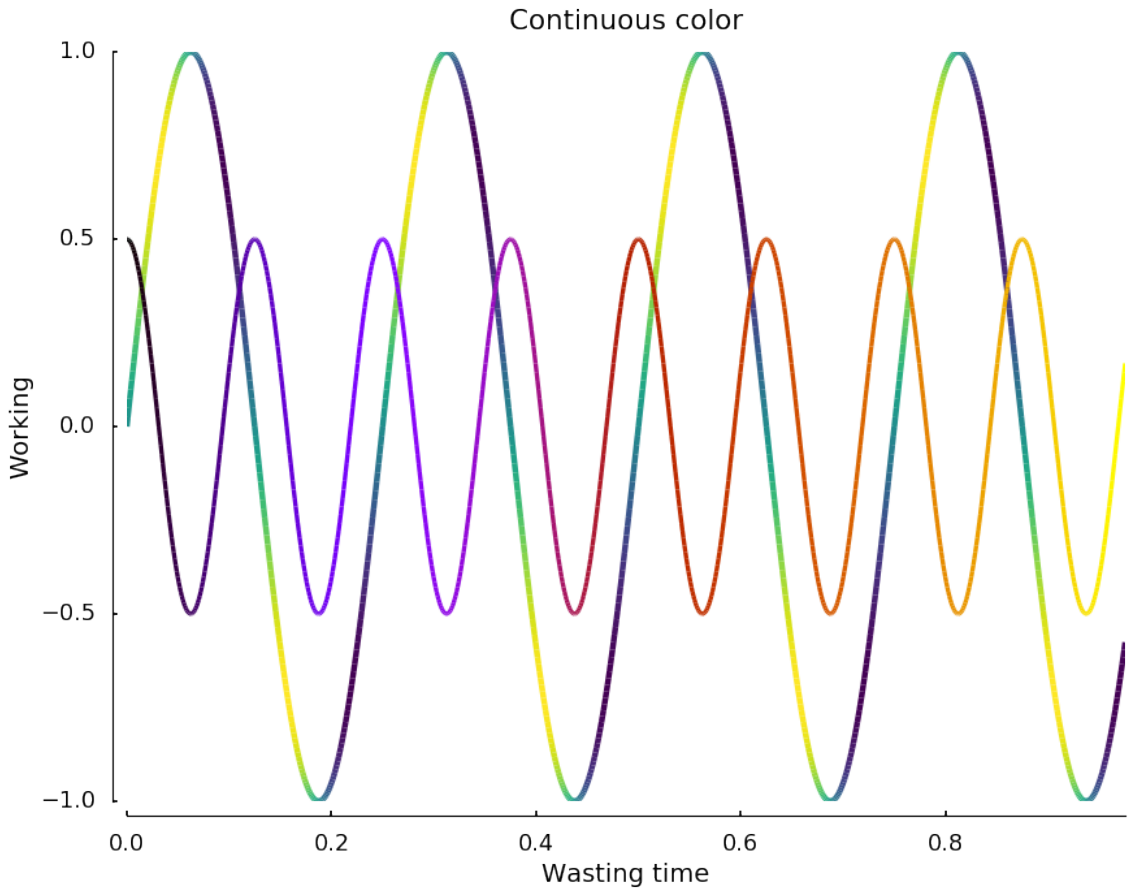

Fig. 4.11: Continuous color-line

hColor: list of strings, [def: ['black']] Control the color of the horizontal lines. The length of the hColor list must be the same as the length of hLines

hShape: list of strings, [def: ['-']] Control the shape of the horizontal lines. The length of the hShape list must be the same as the length of hLines

Return: The current axes

Example:

```
>>> # Create an empty plot:
>>> plt.plot([])
>>> plt.ylim([-1, 1]), plt.xlim([-10, 10])
>>> addLines(plt.gca(), vLines=[0, -5, 5, -7, 7], vColor=['k', 'r', 'g', 'y', 'b'],
>>>          vWidth=[5, 4, 3, 2, 1], vShape=['-', '-', '--', '-', '--'],
>>>          hLines=[0, -0.5, 0.7], hColor=['k', 'r', 'g'], hWidth=[5, 4, 3],
>>>          hShape=['-', '-', '--'])
```

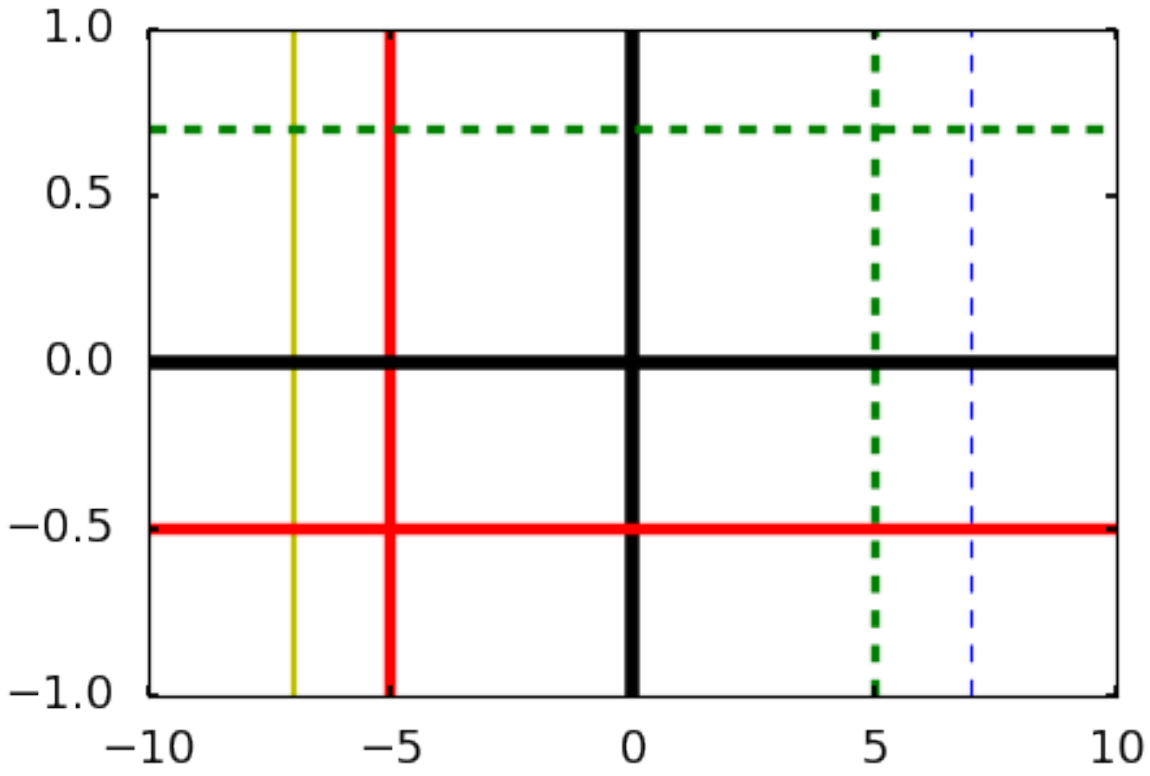

Fig. 4.12: Quickly add some lines to your plot

4.20.2 tilerplot

class `visual.`**`tilerplot`**

Automatic tiler plot for 1, 2 and 3D data.

`plot1D`(*fig*, *y*, *x=None*, *maxplot=10*, *figtitle=''*, *sharex=False*, *sharey=False*, *subdim=None*, *transpose=False*, *color='b'*, *subspace=None*, ***kwargs*)

Simple one dimentional plot

Args:

y: array Data to plot. y can either have one, two or three dimensions. If y is a vector, it will be plot in a simple window. If y is a matrix, all values inside are going to be superimpose. If y is a 3D matrix, the first dimension control the number of subplots.

x: array, optional, [def: None] x vector for plotting data.

Kargs:

figtitle: string, optional, [def: ''] Add a name to your figure

subdim: tuple, optional, [def: None] Force subplots to be subdim=(n_colums, n_rows)

maxplot: int, optional, [def: 10] Control the maximum number of subplot to prevent very large plot. By default, maxplot is 10 which mean that only 10 subplot can be defined.

transpose: bool, optional, [def: False] Invert subplot (row <-> column)

color: string, optional, [def: 'b'] Color of the plot

subspace: dict, optional, [def: None] Control the distance in subplots. Use 'left', 'bottom', 'right', 'top', 'wspace', 'hspace'. Example: {'top':0.85, 'wspace':0.8}

kwargs: Supplementar arguments to control each suplot: title, xlabel, ylabel (which can be list for each subplot) xlim, ylim, xticks, yticks, xticklabels, yticklabels, style, dpax, rmax.

plot2D(*fig*, *y*, *xvec=None*, *yvec=None*, *cmap='inferno'*, *colorbar=True*, *cbticks='minmax'*, *ycb=-10*, *cblabel=''*, *under=None*, *over=None*, *vmin=None*, *vmax=None*, *sharex=False*, *sharey=False*, *textin=False*, *textcolor='w'*, *textype='%.4f'*, *subdim=None*, *mask=None*, *interpolation='none'*, *resample=(0, 0)*, *figtitle=''*, *transpose=False*, *maxplot=10*, *subspace=None*, *contour=None*, *pltargs={}*, *pltype='pcolor'*, *ncontour=10*, *polar=False*, ***kwargs*)

Plot y as an image

Args:

fig: figure A matplotlib figure where plotting

y: array Data to plot. y can either have one, two or three dimensions. If y is a vector, it will be plot in a simple window. If y is a matrix, all values inside are going to be superimpose. If y is a 3D matrix, the first dimension control the number of subplots.

Kargs:

xvec, yvec: array, optional, [def: None] Vectors for y and x axis of each picture

cmap: string, optional, [def: 'inferno'] Choice of the colormap

colorbar: bool/string, optional, [def: True] Add or not a colorbar to your plot. Alternatively, use 'center-max' or 'center-dev' to have a centered colorbar

cbticks: list/string, optional, [def: 'minmax'] Control colorbar ticks. Use 'auto' for [min,(min+max)/2,max], 'minmax' for [min, max] or your own list.

ycb: int, optional, [def: -10] Distance between the colorbar and the label.

cblabel: string, optional, [def: ''] Label for the colorbar

under, over: string, optional, [def: ''] Color for everything under and over the colorbar limit.

vmin, vmax: int/float, optional, [def: None] Control minimum and maximum of the image

sharex, sharey: bool, optional, [def: False] Define if subplots should share x and y

textin: bool, optional, [def: False] Display values inside the heatmap

textcolor: string, optional, [def: 'w'] Color of values inside the heatmap

textype: string, optional, [def: '%.4f'] Way of display text inside the heatmap

subdim: tuple, optional, [def: None] Force subplots to be subdim=(n_colums, n_rows)

interpolation: string, optional, [def: 'none'] Plot interpolation

resample: tuple, optional, [def: (0, 0)] Interpolate the map for a specific dimension. If (0.5, 0.1), this mean that the programme will insert one new point on x-axis, and 10 new points on y-axis. Pimp you map and make it sooo smooth.

figtitle: string, optional, [def: ''] Add a name to your figure

maxplot: int, optional, [def: 10] Control the maximum number of subplot to prevent very large plot. By default, maxplot is 10 which mean that only 10 subplot can be defined.

transpose: bool, optional, [def: False] Invert subplot (row <-> column)

subspace: dict, optional, [def: None] Control the distance in subplots. Use 'left', 'bottom', 'right', 'top', 'wspace', 'hspace'. Example: {'top':0.85, 'wspace':0.8}

contour: dict, optional, [def: None] Add a contour to your 2D-plot. In order to use this parameter, define contour={'data':yourdata, 'label':[yourlabel], kwargs} where yourdata must have the same shape as y, level must float/int from smallest to largest. Use kwargs to pass other arguments to the contour function

kwargs: Supplementar arguments to control each suplot: title, xlabel, ylabel (which can be list for each subplot) xlim, ylim, xticks, yticks, xticklabels, yticklabels, style dpax, rmax.

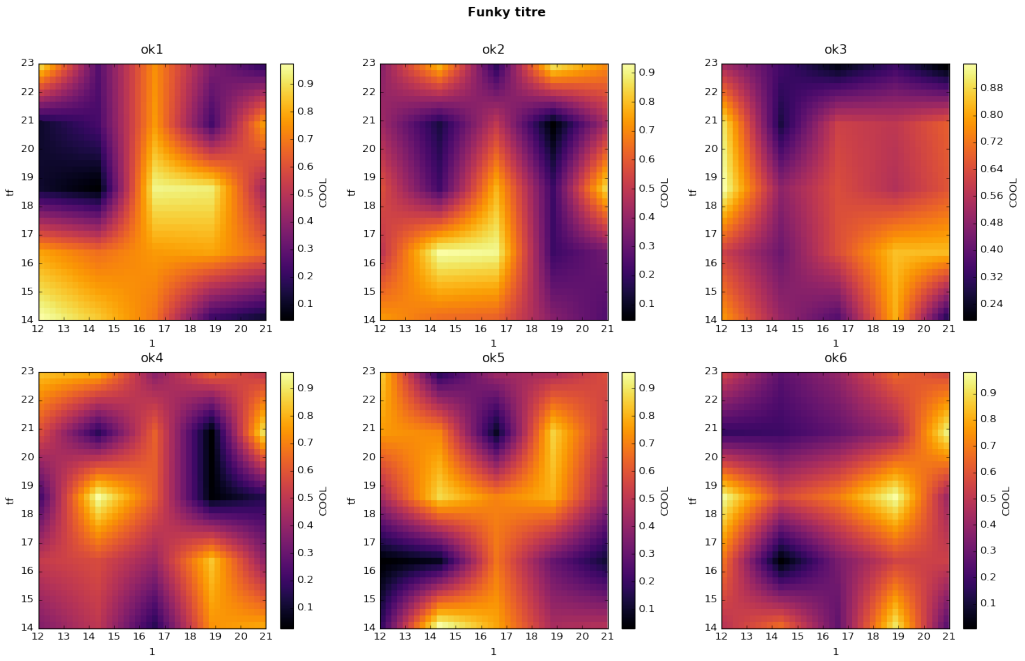

Fig. 4.13: Automatic 1D and 2D plot

4.21 Tools

visual.**rmaxis**(*ax*, *rmax*)
Remove ticks and axis of a existing plot

Args:

ax: matplotlib axes Axes to remove axis

rmax: list of strings List of axis name to be removed. For example, use ['left', 'right', 'top', 'bottom']

visual.**despine**(*ax*, *dpax*, *outward=10*)
Despine axis of a existing plot

Args:

ax: matplotlib axes Axes to despine axis

dpax: list of strings List of axis name to be despined. For example, use ['left', 'right', 'top', 'bottom']

Kargs:

outward: int/float, optional, [def: 10] Distance of despined axis from the original position.

4.22 Tools

4.22.1 Bpstudy

class system.**study**(*name=None*)
Create and manage a study with a files database.

Args:

name: string, optional [def [None]] Name of the study. If this study already exists, this will load the path with the associated database.

Example:

```
>>> # Define variables:
>>> path = '/home/Documents/database'
>>> studyName = 'MyStudy'
```

```
>>> # Create a study object :
>>> studObj = study(name=studyName)
>>> studObj.add(path)    # Create the study
>>> studObj.studies()    # Print the list of studies
```

```
>>> # Manage files in your study :
>>> fileList = studObj.search('filter1', 'filter2', folder='features')
>>> # Let say that fileList contain two files : ['file1.mat', 'file2.pickle']
>>> # Load the second file :
>>> data = studObj.load('features', 'file2.pickle')
```

add(*path*)
Add a new study

Args:

path: string path to your study.

The following folders are going to be created :

- name: the root folder of the study. Same name as the study
- /database: datasets of the study
- /feature: features extracted from the diffrents datasets
- /classified: classified features
- /multifeature: multifeatures files
- /figure: figures of the study
- /physiology: physiological informations
- /backup: backup files
- /setting: study settings
- /other: any other kind of files

delete()
Delete the current study

load(*folder*, *name*)
Load a file. The file can be a .pickle or .mat

Args:

folder: string Specify where the file is located

file: string the complete name of the file

Return: A dictionary containing all the variables.

search(**args*, ***, *folder=''*, *lower=True*)
Get a list of files

Args:

args: string, optional Add some filters to get a restricted list of files, according to the defined filters

folder: string, optional [def: ''] Define a folder to search. By default, no folder is specified so the search will focused on the root folder.

lower: bool, optional [def: True] Define if the search method have to take care of the case. Use False if case is important for searching.

Return: A list containing the files found in the folder.

static studies()
Get the list of all defined studies

static update()
Update the list of studies

4.22.2 Pandas complements

class system.**pdTools**
Tools for pandas DataFrame

Syntax:

- To search if arg1 is in column1:

```
>>> keep = ('column1', arg1)
```

- AND condition for ar1 and arg2 in column1:

```
>>> keep = ('column1', [arg1, 2])
```

- AND condition for arg1 in column1 and arg2 in column2:

```
>>> keep = [('column1', arg1), ('column2', arg2)]
```

- OR condition:

```
>>> keep = ('column1', arg1), ('column2', arg2), ...
```

keep (*df*, **keep*, ***, *keep_idx=True*)
Filter a pandas dataframe and keep only interresting rows.

Args:

df: pandas dataframe The dataframe to filter

keep: tuple/list Control the informations to keep. See the syntax definition

keep_idx: bool, optional [def [True]] Add a column to df to check what are the rows that has been kept.

Return: A pandas Dataframe with only the informations to keep.

remove (*df*, **rm*, ***, *rm_idx=True*)
Filter a pandas dataframe and remove only interresting rows.

Args:

df: pandas dataframe The dataframe to be filter

rm: tuple/list Control the informations to remove. See the syntax definition

rm_idx: bool, optional [def [True]] Add a column to df to check what are the rows that has been removed.

Return: A pandas Dataframe without the removed informations.

search (*df*, **keep*)
Search in a pandas dataframe.

Args:

df: pandas dataframe The dataframe to filter

keep: tuple/list Control the informations to search. See the syntax definition

Return: List of row index for informations found.

4.22.3 Arrays

tools.**ndsplit** (*x*, *sp*, *axis=0*)
Split array (work for odd dimensions)

Args:

x: array Data to split

sp: int Number of chunk

Kargs:

axis: int, optional, [def: 0] Axis for splitting array

Return: List of splitted arrays

tools.**ndjoin**(*x*, *axis=0*)
Join arrays in a list

Args:

x: list List of data.

Kargs:

axis: optional, [def: 0] Axis to join arrays

Return: Array

4.22.4 File management

Save file

system.**savefile**(*name*, **arg*, ***kwargs*)
Save a file without carrying of extension.

arg: for .npy extension kwargs: for .pickle or .mat extensions

Load file

system.**loadfile**(*name*)
Load a file without carrying of extension. The function return a dictionnary data.

4.22.5 Others

tools.**p2str**(*p*)
Convert a pvalue to a string. Usefull for saving !

tools.**uorderlst**(*lst*)
Return a unique set of a list, and preserve order of appearance

4.23 References

This is a non exhaustive list of function and description of what is actually implemented in brainpipe. This list correspond to the most usefull functions.

4.23.1 Pre-processing

Bundle of functions to pre-process data.

```
from brainpipe.preprocessing import *
```

Function	Description
bipolarization	Bipolarise stereotactic eeg electrodes
xyz2phy	Get physiological informations about structures using MNI or Talairach coordinates

4.23.2 Features

Bundle of functions to extract features from neural signals.

```
from brainpipe.features import *
```

Function	Description
sigfilt	Filtered signal only
amplitude	Amplitude of the signal
power	Power of the signal
phase	Phase of the signal
PLF	Phase-Locking Factor
TF	Time-frequency maps
pac	Phase-Amplitude Coupling (large variety of methods)
PhaseLockedPower	Time-frequency maps phase locked to a specific phase
erpac	Event Related Phase-Amplitude Coupling (time-resolved pac)
pfdphase	Preferred-phase
PLV	Phase-Locking Value
PSD	Power Spectrum Density
powerPSD	Power exacted from PSD
SpectralEntropy	Spectral entropy (entropy extracted from PSD)

brainpipe also provide a bundle of tools for features

Function	Description
bandRef	Get usual oscillation bands informations
findBandName	Get physiological name of a frequency band
findBandFcy	Get frequency band from a physiological name
cfcVec	Generate cross-frequency vectors
cfcRndSignals	Generate signals artificialy coupled (great to test pac methods)

4.23.3 Classification

Bundle of functions to classify extracted features.

```
from brainpipe.classification import *
```

Function	Description
defClf	Define a classifier
defCv	Define a cross-validation
classify	Classify features (either each one of them or grouping)
generalization	Generalization of decoding performance (generally, across time)
mf	Multi-features procedure. Select the best combination of features
MFpipe	Multi-features pipeline

4.23.4 Statistics

Bundle of functions to apply statistics.

```
from brainpipe.statistics import *
```

Function	Description
bino_da2p	Get associated p-value of a decoding accuracy using a binomial law
bino_p2da	Get associated decoding accuracy of a p-value using a binomial law
bino_signifeat	Get significant features using a binomial law
perm_2pvalue	Get p-value from a permutation dataset
perm_metric	Get a metric (usefull for mastat)
perm_pvalue2level	Get p-associated level in a permutation distribution
perm_rndDatasets	Generate random dataset of permutations
perm_swap	Randomly swap ndarray (matricial implementation)
perm_rep	Repeat a ndarray of permutations (matricial implementation)
bonferroni	Multiple comparison: Bonferroni
fdr	Multiple comparison: False Discovery Rate
maxstat	Multiple comparison: Maximum statistic
circ_corrcc	Correlation coefficient between one circular and one linear random variable
circ_r	Computes mean resultant vector length for circular data
circ_rtest	Computes Rayleigh test for non-uniformity of circular data

4.23.5 Visualization

Bundle of functions to visualize results and make some <3 pretty plots <3.

```
from brainpipe.visual import *
```

Function	Description
BorderPlot	Plot data and deviation/sem in transparency
addPval	Add p-values to an existing plot
continuouscol	Plot lines with continuous color
addLines	Quickly add vertical and horizontal lines
tilerplot	Generate automatic 1D or 2D subplots with a lot of control
addPval	Add significants p-value to an existing plot
rmaxis	Remove ticks and axis of a existing plot
despine	Despine axis of a existing plot

4.23.6 Tools

This part provide a set complement

```
from brainpipe.tools import *
```

Function	Description
study	Manage your current study without carrying of path
savefile	Quickly save files using most common extensions
loadfile	Quickly load files using most common extensions
pdTools	Some complement functions for pandas Dataframe (search, keep, remove)
ndsplit	Split ndarray (works on odd and even dimensions)
ndjoin	Join ndarray (works on odd and even dimensions)
p2str	Transform a p-value to string (usefull to save files with corresponding p-value)
uorderlst	Get unique ordered elements from a list

Indices and tables

- genindex

- modindex
- search