PTSA Documentation

Release 1.0.0

Maciek Swat

May 04, 2017

Contents

PTSA is an open source project and Python package that facilitates time-series analysis of EEG signals using Python. PTSA builds on xarray functionality and provides several convenience tools that significantly simplify analysis of the EEG data..

To use all features provided by PTSA you will need to install several dependencies: xarray, scipy, numpy, PyWavelets and make sure you have working C/C++ compiler on your machine when you install PTSA

The main object that you will be using in the new PTSA API is called `TimeSeriesX`. `TimeSeriesX` is built on top of `xarray.DataArray`. `xarray.DataArray`, defined in the `xarray` Python package, represents N-D arrays. Because `TimeSeriesX` is a subclass of `xarray.DataArray` it has all the functionality of `xarray.DataArray` in addition to new functions it defines, used specifically in EEG data analysis.

Note: In legacy versions of PTSA the object representing time series is called `TimeSeries` and is built on top of custom-written `dimarray` module. To keep the old analysis code written for older PTSA versions running, we prepended letter **X** to the object representing time-series in the new PTSA , hence the name `TimeSeriesX`.

Besides `TimeSeriesX`, PTSA has 3 main categories of objects: readers, writers, filters. Readers , read various data formats (e.g eeg files, bipolar electrodes files etc..) to make input operations as smooth as possible. Writers (still under development) will output data in several formats (currently only NetCDF output is supported). Filters take as an input `TimeSeriesX` object and output diffrent `TimeSeriesX` object. Most of the tasks related to EEG analysis will rely on using those 3 categories of PTSA objects.

If you'd like to learn Python via series of statistics tutorials look no further than introduction to computational statistics in Python

---

Tip: If you'd like to see PTSA in action check the following self-contained tutorial: Computing classifier of good memory using RAM Data

---

Contents:

CHAPTER 1

Installation

Installing with conda from source

Tip: When experimenting with PTSA, you may want to create a fresh conda environment. You can create one with:

```
conda create -n ptsa python=$VERSION
```

where `$VERSION` is whatever Python version you're using.

After cloning the git repository, install dependencies:

```
conda install -y numpy scipy xarray pywavelets swig
```

Install PTSA:

```
python setup.py install
```

Optional dependencies

For netCDF and IO

- netCDF4: recommended if you want to use xarray for reading or writing netCDF files
- h5netcdf: an alternative library for reading and writing netCDF4 files that does not use the netCDF-C libraries

CHAPTER 2

PTSA Tutorial

Prerequisites

To master PTSA you need to learn few standard packages that scientific Python relies on:

1. numpy (Check out this numpy tutorial)
2. xarray (Full xarray documentation)
3. a little bit of scipy (Scipy tutorial)
4. Stats refresher introduction to computational statistics in Python

Because most of the PTSA functionality builds on `xarray.DataArray` class it is highly recommended you familiarize yourself with xarray. Here is a list of minimum recommended reading you should do before proceeding with this tutorial:

1. Introduction to xarray.DataArray
2. Indexing and selecting data

TimeSeriesX

`TimeSeriesX` object is a basic PTSA object that we typically use to store eeg data and to annotate dimensions.

To create `TimeSeriesX` object we first need to import required ptsa modules:

```
from ptsa.data.TimeSeriesX import TimeSeriesX
from ptsa.data.common import xr
import numpy as np
```

After that we can create our first TimeSeriesX object. Interestingly the syntax we will use here is identical to the syntax you use to create `xarray.DataArray` (really, check the tutorials here) and this is because `TimeSeriesX` is a subclass of `xarray.DataArray`:

```
data = np.arange(0,12.5,0.5)
ts = TimeSeriesX(data, dims=['time'], coords={'time':np.arange(data.shape[0])*2})
```

We create time series using `np.arange` function.I will leave it to you to check the documentation of this `numpy` function (learn it now , you will really need it later). Our TimeSeriesX object is simply a container that stores `data` and provides some very handy axis annotation capabilities. If you print the content of the `ts` on the screen this is what you will get:

```
<xray.TimeSeriesX (time: 25)>
array([  0. ,   0.5,   1. ,   1.5,   2. ,   2.5,   3. ,   3.5,   4. ,
         4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
         9. ,   9.5,  10. ,  10.5,  11. ,  11.5,  12. ])
Coordinates:
  * time     (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
```

As you can tell not only is the `data` stored in the `timeSeriesX` array but we also created `time` axis. notice that the `time` axis is not a series of consecutive integers as in plain array but actually contains "actual" times

If, for example we arenterested interested in getting time series data for times between 2 and 8 seconds (yes, I implicitely assumed the time units are seconds) we can issue the following command

```
ts[(ts.time>=2) & (ts.time<=8)]
```

and the output will be

```
<xray.TimeSeriesX (time: 4)>
array([ 0.5,  1. ,  1.5,  2. ])
Coordinates:
  * time     (time) int64 2 4 6 8
```

We can assign part of the time series in a single call :

```
ts[(ts.time>=2) & (ts.time<=8)] = 1.0
```

```
<xray.TimeSeriesX (time: 25)>
array([  0. ,   1. ,   1. ,   1. ,   1. ,   2.5,   3. ,   3.5,   4. ,
         4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
         9. ,   9.5,  10. ,  10.5,  11. ,  11.5,  12. ])
Coordinates:
  * time     (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
```

It is a good idea to learn this technique of selecting array elements.

Now that we know how to make basic `TimeSeriesX` object, let us explore few operations that come handy when analysing EEG signals

Multi-dimensional TimeSeriesX

Let us add another dimension to our `TimeSeriesX` object:

```
data = np.arange(0,12.5,0.5).reshape(1,25)
ts = TimeSeriesX(data, dims=['bp_pairs','time'], coords={'time':np.arange(data.
↪shape[1])*2})
```

Note that we added new dimension `bp_pairs`. Here is the printout of the `ts` object:

```
<xray.TimeSeriesX (bp_pairs: 1, time: 25)>
array([[  0. ,   0.5,   1. ,   1.5,   2. ,   2.5,   3. ,   3.5,   4. ,
          4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
          9. ,   9.5,  10. ,  10.5,  11. ,  11.5,  12. ]])
Coordinates:
  * time      (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
  * bp_pairs  (bp_pairs) int64 0
```

Note that the `bp_pairs` axis has dimension of one at the axis elements are consecutive integers because we did not included entry for the `bp_pairs` in the coords argument of thr `TimeSeriesX` constructor

Let us fix this by assigning the labels to `bp_pairs` axes:

```
ts['bp_pairs'] = np.array(['LPOG2-LPOG10'],dtype='|S32')
```

Now the `ts` array will look as follows:

```
<xray.TimeSeriesX (bp_pairs: 1, time: 25)>
array([[  0. ,   0.5,   1. ,   1.5,   2. ,   2.5,   3. ,   3.5,   4. ,
          4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
          9. ,   9.5,  10. ,  10.5,  11. ,  11.5,  12. ]])
Coordinates:
  * time      (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
  * bp_pairs  (bp_pairs) |S32 'LPOG2-LPOG10'
```

Concatenating Two TimeSeriesX objects

Let us create a second `TimeSeriesX` object

```
data1 = data = np.arange(12.5,25.0,0.5).reshape(1,25)
ts1 = TimeSeriesX(
    data1,
    dims=['bp_pairs','time'],
    coords={
            'time':np.arange(data.shape[1])*2,
            'bp_pairs':np.array(['LPOG2-LPOG9'],dtype='|S32')
    }
 )
```

Notice that this time we assigned `bp_pairs` in the constructor of the `ts1`

Let us "glue" together `ts` and `ts1` using `concat` function from `xarray`.

```
tsm = xr.concat([ts,ts1],dim='bp_pairs')
```

the output is as expected:

```
<xray.DataArray (bp_pairs: 2, time: 25)>
array([[  0. ,   0.5,   1. ,   1.5,   2. ,   2.5,   3. ,   3.5,   4. ,
          4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
          9. ,   9.5,  10. ,  10.5,  11. ,  11.5,  12. ],
       [ 12.5,  13. ,  13.5,  14. ,  14.5,  15. ,  15.5,  16. ,  16.5,
         17. ,  17.5,  18. ,  18.5,  19. ,  19.5,  20. ,  20.5,  21. ,
         21.5,  22. ,  22.5,  23. ,  23.5,  24. ,  24.5]])
Coordinates:
  * time      (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
  * bp_pairs  (bp_pairs) |S32 'LPOG2-LPOG10' 'LPOG2-LPOG9'
```

Note: To refer to `xarray` functionality in PTSA we use `xr` alias. At the begining of your script include

```
from ptsa.data.common import xr
```

and then xr will refer to `xarrat` or `xray` toolkits. This weay you do not have to worry too much wheather you are working with `xarray` or its predecessor `xray`

Mean

To compute mean array of teh time series along the specified axis type:

```
mean_tsm = tsm.mean(dim='time')
```

The output will be

```
array([  6. ,  18.5])
Coordinates:
  * bp_pairs  (bp_pairs) |S32 'LPOG2-LPOG10' 'LPOG2-LPOG9'
```

As you can see TimeSeriesX syntax is quite expressive and clean making it easier to remember which axis we use for aggregation operations. For example if we were to compute mean along `bp_pairs` axis the output woudl look as follows:

```
mean_tsm = tsm.mean(dim='bp_pairs')
```

```
<xray.DataArray (time: 25)>
array([  6.25,   6.75,   7.25,   7.75,   8.25,   8.75,   9.25,   9.75,
        10.25,  10.75,  11.25,  11.75,  12.25,  12.75,  13.25,  13.75,
        14.25,  14.75,  15.25,  15.75,  16.25,  16.75,  17.25,  17.75,
        18.25])
Coordinates:
  * time     (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
```

Min/Max

To find min/max along given axis we do the following:

```
min_tsm = tsm.min(dim='time')
```

and the result is :

```
<xray.DataArray (bp_pairs: 2)>
array([  0. ,  12.5])
Coordinates:
  * bp_pairs  (bp_pairs) |S32 'LPOG2-LPOG10' 'LPOG2-LPOG9'
```

Obviously for max operation you woudl replace `min` with `max` in the above code. I will leave this challenging exercise for you to complete by the end of the quarter.

Standard Deviation

Finding standard deviation is easy as well

```
std_tsm = tsm.min(dim='time')
```

with output being

```
<xray.DataArray (bp_pairs: 2)>
array([ 3.60555128,  3.60555128])
Coordinates:
  * bp_pairs  (bp_pairs) |S32 'LPOG2-LPOG10' 'LPOG2-LPOG9'
```

Transposing axes in TimeSeriesX

Quite often you will find it very convenient to rearrange the order of axes in your multi-dimmensional array. TimeSeriesX makes it very easy:

```
swapaxes_tsm = tsm.transpose('time','bp_pairs')
```

All you have to do is to call `transpose` function and list names of all dimensions in the desired order. Take a look:

```
<xray.DataArray (time: 25, bp_pairs: 2)>
array([[  0. ,  12.5],
       [  0.5,  13. ],
       [  1. ,  13.5],
       [  1.5,  14. ],
       [  2. ,  14.5],
       [  2.5,  15. ],
       [  3. ,  15.5],
       [  3.5,  16. ],
       [  4. ,  16.5],
       [  4.5,  17. ],
       [  5. ,  17.5],
       [  5.5,  18. ],
       [  6. ,  18.5],
       [  6.5,  19. ],
       [  7. ,  19.5],
       [  7.5,  20. ],
       [  8. ,  20.5],
       [  8.5,  21. ],
       [  9. ,  21.5],
       [  9.5,  22. ],
       [ 10. ,  22.5],
       [ 10.5,  23. ],
       [ 11. ,  23.5],
       [ 11.5,  24. ],
       [ 12. ,  24.5]])
Coordinates:
  * time      (time) int64 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 ...
  * bp_pairs  (bp_pairs) |S32 'LPOG2-LPOG10' 'LPOG2-LPOG9'
```

Other useful aggregation operations

Since `TimeseriesX` is a subclass of `xarray.DataArray`, you can use all sorts of aggregation operations that `xarray.DataArray` provides. For a full list of those please consult Computation with DataArray or xarray API

CHAPTER 3

Interacting with RAM Data

Even though PTSA is a general Python framework for time series analysis, it has some built-in modules that facilitate working with various formats of EEG data and associated experimental data. In this section we will see how to efficiently ready and process data store in formats used by the

DARPA RAM project.

Let's start by looking at how to read experimental events stored in Matlab Format. The class we will use is called `BaseEventReader`.

Reading events using BaseEventReader

To read events that RAM project uses we need to mount RAM data directory on our computer. In my case I mounted it as `/Users/rhino_root/data`. Now, to read the events we first need to import `BaseEventReader`

```
from ptsa.data.readers import BaseEventReader
```

then we will specify path to the event file, create instance of the `BaseEventReader` called `base_e_reader` and pass two arguments to the constructor. The first argument specifies event file and the second one instructs the reader to remove all the event entries that do not have valid EEG file associated with it. Subsequently we call `read()` function and select only those events that are ot `type` "WORD"

```
e_path = '/Volumes/rhino_root/data/events/RAM_FR1/R1111M_events.mat'
# ------------------- READING EVENTS
base_e_reader = BaseEventReader(filename=e_path, eliminate_events_with_no_eeg=True)
base_events = base_e_reader.read()
base_events = base_events[base_events.type == 'WORD']
```

when we print events to the screen we will get the following output:

```
rec.array([ ('R1111M', 0, 1, 1, 'WORD', 'BEAR', 17, 1, 1453499295325.0, 1, 5211, -999,
 ↪ 0, 'v_1.05', 'X', -999.0, -999.0, '[]', -999.0, '[]', 0, '/Volumes/rhino_root/data/
 ↪eeg/R1111M/eeg.noreref/R1111M_FR1_0_22Jan16_1638', 100521),
('R1111M', 0, 1, 2, 'WORD', 'WING', 294, 1, 1453499297942.0, 1, 5749, -999, 0, 'v_1.05
 ↪', 'X', -999.0, -999.0, '[]', -999.0, '[]', 0, '/Volumes/rhino_root/data/eeg/R1111M/
 ↪eeg.noreref/R1111M_FR1_0_22Jan16_1638', 101829),
```

```
('R1111M', 0, 1, 3, 'WORD', 'DOOR', 79, 1, 1453499300510.0, 1, 7882, -999, 0, 'v_1.05
↪', 'X', -999.0, -999.0, '[]', -999.0, '[]', 0, '/Volumes/rhino_root/data/eeg/R1111M/
↪eeg.noreref/R1111M_FR1_0_22Jan16_1638', 103113),
...,
('R1111M', 3, 20, 10, 'WORD', 'TRUCK', 282, 1, 1454447574230.0, 1, 4369, -999, 0, 'v_
↪1.05', 'X', -999.0, -999.0, '[]', -999.0, '[]', 0, '/Volumes/rhino_root/data/eeg/
↪R1111M/eeg.noreref/R1111M_FR1_3_02Feb16_1528', 1128811),
('R1111M', 3, 20, 11, 'WORD', 'CORD', 62, 0, 1454447576613.0, 1, -999, -999, 0, 'v_1.
↪05', 'X', -999.0, -999.0, '[]', -999.0, '[]', 0, '/Volumes/rhino_root/data/eeg/
↪R1111M/eeg.noreref/R1111M_FR1_3_02Feb16_1528', 1130002),
('R1111M', 3, 20, 12, 'WORD', 'OAR', 169, 0, 1454447579014.0, 1, -999, -999, 0, 'v_1.
↪05', 'X', -999.0, -999.0, '[]', -999.0, '[]', 0, '/Volumes/rhino_root/data/eeg/
↪R1111M/eeg.noreref/R1111M_FR1_3_02Feb16_1528', 1131203)],
dtype=[('subject', 'S256'), ('session', '<i8'), ('list', '<i8'), ('serialpos', '<i8'),
↪ ('type', 'S256'), ('item', 'S256'),
('itemno', '<i8'), ('recalled', '<i8'), ('mstime', '<f8'), ('msoffset', '<i8'), (
↪'rectime', '<i8'), ('intrusion', '<i8'),
('isStim', '<i8'), ('expVersion', 'S256'), ('stimLoc', 'S256'), ('stimAmp', '<f8'), (
↪'stimAnode', '<f8'), ('stimAnodeTag', 'S256'), ('stimCathode', '<f8'),
('stimCathodeTag', 'S256'), ('stimList', '<i8'), ('eegfile', 'S256'), ('eegoffset', '
↪<i8')])
```

Indicating that the event object is in fact `numpy.recarray`

Finding Paths using JsonIndexReader

While one can always specify the path to the events structure by hand, PTSA has a class `JsonIndexReader` that tracks this information. The location of the various event files is kept in JSON format, in */protocols/r1.json*, and `JsonIndexReader` allows one to query the index by property.

We build the reader with:

```
jr = JsonIndexReader('/protocols/r1.json')
```

To get the location of the event files for subject R1111M from the FR1 experiment, we _____:

```
event_paths = jr.aggregate_values('all_events',subject='R1111M',experiment='FR1')
```

The *aggregate_values* method returns the set of all fields in the JSON index that match the keyword arguments. The most useful keyword arguments are 'subject', 'experiment', and 'session'.

Since With the paths in hand, we can load the events using the BaseEventReader discussed above:

```
events = [BaseEventReader(filename=path).read() for path in sorted(event_paths)]
```

which will return a list of event structures. The call to `sorted()` ensures that the events are read in order of session. To collapse the list into a single array, we call `numpy.concatenate()`:

```
events =  numpy.concatenate(events)
```

To access the fields of the array as though they were attributes, we need to convert it to a record array:

```
events = events.view(numpy.recarray)
```

and now the events structure is exactly as described in the previous section.

Reading Electrode Information using TalReader

To read electrode information that is stored in the so called tal_structs we will use `TalReader` object. We first import TalReader:

```
from ptsa.data.readers import TalReader
```

Next we specify path to the actual `.mat` file containing information about electrodes , construct `tal_reader` object and call `read` function to initiate reading of the `tal_structs` file.

```
tal_path = '/Volumes/rhino_root/data/eeg/R1111M/tal/R1111M_talLocs_database_bipol.mat'
tal_reader = TalReader(filename=tal_path)
tal_structs = tal_reader.read()
```

The `read` function returns `numpy.recarray` populated with electrode information:

```
Out[77]:
rec.array([ ('R1111M', array([1, 2]), 'LPOG1-LPOG2', 'LPOG', -67.6431, -19.84015, -17.
↪08995, 'Left Cerebrum',
'Temporal Lobe', 'Middle Temporal Gyrus', 'Gray Matter', 'Brodmann area 21', '[]',
↪'lsag', '1-2', 'G', 8.22266263809965
...
```

This is not the most infromative output so it is best to first check what columns are available in the `tal_structs`:

```
print tal_structs.dtype.names
```

for which you get an output

```
('subject',
 'channel',
 'tagName',
 'grpName',
 'x',
 'y',
 'z',
 'Loc1',
 'Loc2',
 'Loc3',
 'Loc4',
 'Loc5',
 'Loc6',
 'Montage',
 'eNames',
 'eType',
 'bpDistance',
 'avgSurf',
 'indivSurf',
 'locTag')
```

At this point we can print single columns e.g. `channel` and `tagName`

```
print tal_structs[['channel','tagName']]
```

that outputs

```
rec.array([(array([1, 2]), 'LPOG1-LPOG2'), (array([1, 9]), 'LPOG1-LPOG9'),
(array([2, 3]), 'LPOG2-LPOG3'), (array([ 2, 10]), 'LPOG2-LPOG10'),
(array([3, 4]), 'LPOG3-LPOG4'), (array([ 3, 11]), 'LPOG3-LPOG11'),
(array([4, 5]), 'LPOG4-LPOG5'), (array([ 4, 12]), 'LPOG4-LPOG12'),
(array([5, 6]), 'LPOG5-LPOG6'), (array([ 5, 13]), 'LPOG5-LPOG13'),
(array([6, 7]), 'LPOG6-LPOG7'), (array([ 6, 14]), 'LPOG6-LPOG14'),
...
```

`TalReader` also provides two convenience functions `get_monopolar_channels` and ‘‘ get_bipolar_pairs‘‘ that extract a list of individual channel numbers and a list of bipolar pairs.

```
monopolar_channels = tal_reader.get_monopolar_channels()
bipolar_pairs = tal_reader.get_bipolar_pairs()
```

Note: You can also extract bipolar pairs by typing:

```
tal_structs['channel']
```

Reading EEG time series using EEGReader

To read EEG time series’ associated with events we typically use `EEGReader`. Here is the syntax:

```
from ptsa.data.readers import EEGReader
eeg_reader = EEGReader(events=base_events, channels=monopolar_channels,
                       start_time=0.0, end_time=1.6, buffer_time=1.0)

base_eegs = eeg_reader.read()
```

After importing `EEGReader` we pass the following objects to `EEGReader` constructor: - `events` - this is the array of events (read using `BaseEventReader`) for which we want to obtain eeg time series’ - `channels` - and array of monopolar channels (NOT bipolar pairs) for which we want eeg signals - `start_time` - offset in seconds relative the the onset of event at which we start reading EEG signal - `end_time` - offset in seconds relative the the onset of event at which we stop reading EEG signal - `buffer` - time interval in seconds which determines how much extra data will be added to each eeg signal segment

Here is the output:

```
<xray.TimeSeriesX (channels: 100, events: 1020, time: 1800)>
array([[[ 3467.059196,  3471.312604,  3473.970984, ...,  3580.306184,
          3581.901212,  3588.813   ],
        [ 3609.548364,  3609.548364,  3612.73842 , ...,  3368.16746 ,
          3351.153828,  3343.710364],
        [ 3444.728804,  3449.513888,  3454.298972, ...,  3513.315008,
          3519.163444,  3512.251656],
        ...,
        [ 3404.321428,  3404.853104,  3410.70154 , ...,  3164.535552,
          3163.4722  ,  3157.623764],
        [ 3175.700748,  3156.028736,  3167.725608, ...,  3151.775328,
          3142.20516 ,  3147.52192 ],
        [ 3128.91326 ,  3136.8884  ,  3134.761696, ...,  3286.289356,
          3263.958964,  3272.46578 ]],
```

CHAPTER 4

Filtering Time Series

Filtering is a type of operation that takes as an input one time series and outputs another. For example Butterworth band-pass filter may remove unwanted frequencies from your signal. Computing Wavelets is also a form of filtering operation that takes one time series and outputs another one with signal that is decomposed into wavelet components

Note: All filter objects define function `filter`. This is the function you call to make filter do its job

Let us start with something simple - `MonopolarToBipolarMapper`.

MonopolartoBipolarMapper

This filter takes as inputs an array of monopolar eeg data - `time_series` parameter below and the array of bipolar pairs (`bipolar_pairs`) and outputs another time series containing pairwise differences for electrode pairs specified in `bipolar_pairs`

Here is the syntax:

```
from ptsa.data.filters import MonopolarToBipolarMapper
m2b = MonopolarToBipolarMapper(time_series=base_eegs, bipolar_pairs=bipolar_pairs)
bp_eegs = m2b.filter()
```

We import `MonopolarToBipolarMapper` from `ptsa.data.filters` PTSA package , crteate an instance of MonopolarToBipolarMapper with appropriate parameters and then call `filter` function to compute pairwise signal differences. Here is the output:

```
<xray.TimeSeriesX (bipolar_pairs: 141, events: 1020, time: 1800)>
array([[[  7119.14164 ,   7119.673316,   7119.14164 , ...,   7156.35896 ,
           7159.549016,   7164.3341  ],
        [  7175.499296,   7178.157676,   7186.132816, ...,   7022.376608,
           7009.084708,   7009.084708],
        [  7061.188956,   7063.31566 ,   7067.037392, ...,   7227.071868,
           7228.13522 ,   7221.223432],
```

```
...
```

Notice that this `TimeSeriesX` object is indexed by bipolar_pairs. As a matter of fact if you type:

```
bp_eegs.bipolar_pairs
```

you will get

```
<xray.DataArray 'bipolar_pairs' (bipolar_pairs: 141)>
array([('001', '002'), ('001', '009'), ('002', '003'), ('002', '010'),
       ('003', '004'), ('003', '011'), ('004', '005'), ('004', '012'),
```

ButterworthFilter

To use Butterworth filtering inside PTSA you have two choices: use `ButterworthFilter` object and passing `TimeSeriesX` object to it or use a convenience function inside `TimeSeriesX` object.

Let's us start by showing first `ButterworthFilter`:

```
from ptsa.data.filters import ButterworthFilter
b_filter = ButterworthFilter(time_series=bp_eegs, freq_range=[58., 62.], filt_type=
↪'stop', order=4)
bp_eegs_filtered = b_filter.filter()
```

Here we create ButterworthFilter object (after importing it from PTSA's `filters` package) and specify filter parameters: we specify frequency range that we want to filter out (to remove frequencies we set `filt_type` to `'stop'`) and specify filter order (here it is 4)

As before, once the filter object is initialized we call `filter` function to get the result (filtered `TimeSeriesX`).

If you prefer you may use alternative way of running Butterworth filter on a `TimeSeriesX` by calling `filtered` function on a `TimeseriesX` object

```
bp_eegs_filtered_1 = bp_eegs.filtered(freq_range=[58., 62.], filt_type='stop',␣
↪order=4)
```

CHAPTER 5

PTSA API

Subpackages

ptsa.data

ptsa.data.readers

ptsa.data.readers.IndexReader

class ptsa.data.readers.IndexReader.JsonIndexReader(index_file)

Bases: "object"

Reads from one of the top level indexing files (r1.json, ltp.json) Allows for aggregation of values across any field with any constraint through the use of aggregateValues() or the specific methods subject(), experiment(), session() or montage().

aggregate_values(field, **kwargs)

Aggregates values across different experiments, subjects, sessions, etc. Allows you to specify constraints for the query (e.g. subject='R1001P', experiment='FR1') :param field: The field to aggregate – can be a leaf or internal node of the json tree. :param kwargs: Constraints – subject='R1001P', experiment='FR1', etc. :return: a set of all of the fields that were found

experiments(**kwargs)

Requests a list of experiments, filtered by kwargs :param kwargs: e.g. subject='R1001P', localization=1 :return: list of experiments

get_value(field, **kwargs)

Gets a single field from the dictionary tree. Raises a KeyError if the field is not found, or there are multiple entries for the field with the specified constraints :param field: the name of the field to retrieve :param kwargs: constraints (e.g. subject='R1001P', session=0, experiment='FR3') :return: the value requested

montages(**kwargs)

Returns a list of the montage codes (#.#), filtered by kwargs :param kwargs: e.g. subject='R1001P', experiment='FR1', session=0 :return: list of montages

sessions(**kwargs)

Requests a list of session numbers, filtered by kwargs :param kwargs: e.g. subject='R1001P', experiment='FR3' :return: list of sessions

subjects(**kwargs)

Requests a list of subjects, filtered by kwargs :param kwargs: e.g. experiment='FR1', session=0 :return: list of subjects

ptsa.data.readers.BaseEventReader

class ptsa.data.readers.BaseEventReader.BaseEventReader(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.readers.BaseReader.BaseReader"

Reader class that reads event file and returns them as np.recarray

correct_eegfile_field(events)

Replaces 'eeg.reref' with 'eeg.noreref' in eegfile path :param events: np.recarray representing events. One of hte field of this array should be eegfile :return:

find_data_dir_prefix()

determining dir_prefix

data on rhino database is mounted as /data copying rhino /data structure to another directory will cause all files in data have new prefix example: self._filename='/Users/m/data/events/R1060M_events.mat' prefix is '/Users/m' we use find_dir_prefix to determine prefix based on common_root in path with and without prefix

Returns: data directory prefix

read_matlab()

Reads Matlab event file and returns corresponging np.recarray. Path to the eegfile is changed w.r.t original Matlab code to account for the following: 1. /data dir of the database might have been mounted under different mount point e.g. /Users/m/data 2. use_reref_eeg is set to True in which case we replaces 'eeg.reref' with 'eeg.noreref' in eegfile path

Returns: np.recarray representing events

ptsa.data.readers.EEGReader

class ptsa.data.readers.EEGReader.EEGReader(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.readers.BaseReader.BaseReader"

Reader that knows how to read binary eeg files. It can read chunks of the eeg signal based on events input or can read entire session if session_dataroot is non empty

compute_read_offsets(dataroot)

Reads Parameter file and exracts sampling rate that is used to convert from start_time, end_time, buffer_time (expressed in seconds) to start_offset, end_offset, buffer_offset expressed as integers indicating number of time series data points (not bytes!)

Parameters: **dataroot** – core name of the eeg datafile

Returns: tuple of 3 {int} - start_offset, end_offset, buffer_offset

read()

Calls read_events_data or read_session_data depending on user selection :return: TimeSeriesX object

read_events_data()

Reads eeg data for individual event :return: TimeSeriesX object (channels x events x time) with data for individual events

read_session_data()

Reads entire session worth of data :return: TimeSeriesX object (channels x events x time) with data for entire session the events dimension has length 1

ptsa.data.readers.PTSAEventReader

class ptsa.data.readers.PTSAEventReader.PTSAEventReader(**kwds)

Bases: "ptsa.data.readers.BaseEventReader.BaseEventReader", "ptsa.data.readers.BaseReader.BaseReader"

Event reader that returns original PTSA Events object with attached rawbinwrappers rawbinwrappers are objects that know how to read eeg binary data

attach_rawbinwrapper_groupped(evs)

attaches raw bin wrappers to individual records. Single rawbinwrapper is shared between events that have same eegfile :param evs: Events object :return: Events object with attached rawbinarappers

attach_rawbinwrapper_individual(evs)

attaches raw bin wrappers to individual records. Uses separate rawbinwrapper for each record :param evs: Events object :return: Events object with attached rawbinarappers

read()

Reads Matlab event file , converts it to np.recarray and attaches rawbinwrappers (if appropriate flags indicate so) :return: Events object. depending on flagg settings the rawbinwrappers may be attached as well

ptsa.data.readers.TalReader

class ptsa.data.readers.TalReader.TalReader(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.readers.BaseReader.BaseReader"

Reader that reads tal structs Matlab file and converts it to numpy recarray

get_bipolar_pairs()

Returns: numpy recarray where each record has two fields 'ch0' and 'ch1' storing channel labels.

get_monopolar_channels()

Returns: numpy array of monopolar channel labels

read()

:return:np.recarray representing tal struct array (originally defined in Matlab file)

ptsa.data.filters

ptsa.data.filters.ButterworthFilter

class ptsa.data.filters.ButterworthFilter.ButterworthFilter(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.filters.BaseFilter.BaseFilter"

Applies Butterworth filter to a time series

Parameters: **kwds** – allowed values are:

:param time_series TimeSeriesX object

:param order Butterworth filter order

:param freq_range{list-like} Array [min_freq, max_freq] describing the filter range

:return :None

filter()

Applies Butterwoth filter to input time series and returns filtered TimeSeriesX object :return: TimeSeriesX object

ptsa.data.filters.DataChopper

class ptsa.data.filters.DataChopper.DataChopper(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.filters.BaseFilter.BaseFilter"

EventDataChopper converts continuous time series of entire session into chunks based on the events specification In other words you may read entire eeg session first and then using EventDataChopper divide it into chunks corresponding to events of your choice

filter()

Chops session into chunks orresponding to events :return: timeSeriesX object with chopped session

get_event_chunk_size_and_start_point_shift(eegoffset, samplerate, offset_time_array)

Computes number of time points for each event and read offset w.r.t. event's eegoffset :param ev: record representing single event :param samplerate: samplerate fo the time series :param offset_time_array: "offsets" axis of the DataArray returned by EEGReader. This is the axis that represents time axis but instead of beind dimensioned to seconds it simply represents position of a given data point in a series The time axis is constructed by dividint offsets axis by the samplerate :return: event's read chunk size {int}, read offset w.r.t. to event's eegoffset {}

ptsa.data.filters.MonopolarToBipolarMapper

class ptsa.data.filters.MonopolarToBipolarMapper.MonopolarToBipolarMapper(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.filters.BaseFilter.BaseFilter"

Object that takes as an input time series for monopolar electrodes and an array of bipolar pairs and outputs Time series where 'channels' axis is replaced by 'bipolar_pairs' axis and the time series data is a difference between time series corresponding to different electrodes as specified by bipolar pairs

bipolar_pairs = rec.array([], dtype=[('ch0', 'S3'), ('ch1', 'S3')])

filter()

Turns time series for monopolar electrodes into time series where where 'channels' axis is replaced by 'bipolar_pairs' axis and the time series data is a difference between time series corresponding to different electrodes as specified by bipolar pairs

Returns: TimeSeriesX object

time_series = <xarray.TimeSeriesX (time: 1)> array([0.]) Coordinates: * time (time) int64 0

ptsa.data.filters.MonopolarToBipolarMapper.main_fcn()

ptsa.data.filters.MonopolarToBipolarMapper.new_fcn()

ptsa.data.filters.MorletWaveletFilter

ptsa.data.filters.ResampleFilter

class ptsa.data.filters.ResampleFilter.ResampleFilter(**kwds)

Bases: "ptsa.data.common.TypedUtils.PropertiedObject", "ptsa.data.filters.BaseFilter.BaseFilter"

Resample Filter

filter()

resamples time series :return:resampled time series with sampling frequency set to resamplerate

ptsa.data.edf package

Submodules

ptsa.data.edf.edf module

ptsa.data.edf.edf.read_annotations(filepath)

Read in all the annotations from an EDF/BDF file into a record array. Note that the onset times are converted to seconds.

filepath [{str}] The path and name of the EDF/BDF file.

annotations [{np.recarray}] A record array with onsets, duration, and annotations.

ptsa.data.edf.edf.read_number_of_samples(filepath, edfsignal)

Read the number of samples of a signal in an EDF/BDF file. Note that different signals can have different numbers of samples.

filepath [{str}] The path and name of the EDF/BDF file.

edfsignal [{int}] The signal whose samplerate to retrieve.

num_samples [{long}] The number of samples for that signal.

ptsa.data.edf.edf.read_number_of_signals(filepath)

Read in number of signals in the EDF/BDF file.

filepath [{str}] The path and name of the EDF/BDF file.

num_signals [{int}] Number of signals in the EDF/BDF file.

ptsa.data.edf.edf.read_samplerate(filepath, edfsignal)

Read the samplerate for a signal in an EDF/BDF file. Note that different signals can have different samplerates.

filepath [{str}] The path and name of the EDF/BDF file.

edfsignal [{int}] The signal whose samplerate to retrieve.

samplerate [{float}] The samplerate for that signal.

ptsa.data.edf.edf.read_samples(filepath, edfsignal, offset, n)

Read in samples from a signal in an EDF/BDF file.

filepath [{str}] The path and name of the EDF/BDF file.

edfsignal [{int}] The signal whose samplerate to retrieve.

offset [{long}] Offset in samples into the file where to start reading.

n [{int}] Number of samples to read, starting at offset.

samples [{np.ndarray}] An ndarray of samples read from the file.

ptsa.data.edf.setup module

Module contents

ptsa.data.common package

Submodules

ptsa.data.common.TypedUtils module

class ptsa.data.common.TypedUtils.TypeValTuple(name, type, default)

Bases: "tuple"

default

Alias for field number 2

name

Alias for field number 0

type

Alias for field number 1

ptsa.data.common.axis_utils module

ptsa.data.common.path_utils module

ptsa.data.common.pathlib module

class ptsa.data.common.pathlib.PurePath

Bases: "object"

PurePath represents a filesystem path and offers operations which don't imply any actual filesystem I/O. Depending on your system, instantiating a PurePath will return either a PurePosixPath or a PureNTPath object. You can also instantiate either of these classes directly, regardless of your system.

as_bytes()

Return the bytes representation of the path. This is only recommended to use under Unix.

as_posix()

Return the string representation of the path with forward (/) slashes.

drive

The drive prefix (letter or UNC path), if any

ext

The final component's extension, if any.

is_absolute()

True if the path is absolute (has both a root and, if applicable, a drive).

is_reserved()

Return True if the path contains one of the special names reserved by the system, if any.

join(*args)

Combine this path with one or several arguments, and return a new path representing either a subpath (if all arguments are relative paths) or a totally different path (if one of the arguments is anchored).

match(path_pattern)

Return True if this path matches the given pattern.

normcase()

Return this path, possibly lowercased if the path flavour has case-insensitive path semantics. Calling this method is not needed before comparing Path instances.

parent(level=1)

A parent or ancestor (if *level* is specified) of this path.

parents()

Iterate over this path's parents, in ascending order.

parts

An object providing sequence-like access to the components in the filesystem path.

relative()

Return a new path without any drive and root.

relative_to(*other)

Return the relative path to another path identified by the passed arguments. If the operation is not possible (because this is not a subpath of the other path), raise ValueError.

root

The root of the path, if any

ptsa.data.common.xr module

Workaround for different working with multiple version of xray . In the new versions xray has been renamed to xarray

Module contents

ptsa.data.TimeSeriesX

ptsa.data.events

class ptsa.data.events.Events

Bases: "numpy.recarray"

A recarray with the events to be analyzed. Includes convenience functions to add and remove fields and a function to get a TimeSeries instance with the data linked to each event.

add_fields(**fields)

Add fields from the keyword args provided and return a new instance.

>><<fields_to_add** [{dictionary}] Names in the dictionary correspond to new field names and the values specify their content. To add an empty field, pass a dtype as the value.

New Events instance with the specified new fields.

events.add_fields(name1=array1, name2=dtype('i4'))

get_data(channels, start_time, end_time, buffer_time=0.0, resampled_rate=None, filt_freq=None, filt_type='stop', filt_order=4, keep_buffer=False, esrc='esrc', eoffset='eoffset', loop_axis=None, num_mp_procs=0, eoffset_in_time=True, **kwds)

Return the requested range of data for each event by using the proper data retrieval mechanism for each event.

channels: {list,int,None} Channels from which to load data.

start_time: {float} Start of epoch to retrieve (in time-unit of the data).

end_time: {float} End of epoch to retrieve (in time-unit of the data).

buffer_time: {float},optional Extra buffer to add on either side of the event in order to avoid edge effects when filtering (in time unit of the data).

resampled_rate: {float},optional New samplerate to resample the data to after loading.

filt_freq: {array_like},optional The range of frequencies to filter (depends on the filter type.)

filt_type = {scipy.signal.band_dict.keys()},optional Filter type.

filt_order = {int},optional The order of the filter.

keep_buffer: {boolean},optional Whether to keep the buffer when returning the data.

esrc [{string},optional] Name for the field containing the source for the time series data corresponding to the event.

eoffset: {string},optional Name for the field containing the offset (in seconds) for the event within the specified source.

eoffset_in_time: {boolean},optional If True, the unit of the event offsets is taken to be time (unit of the data), otherwise samples.

verbose: {bool} turns on verbose printout of the function - e.g. timing information will be output to the screen

A TimeSeries instance with dimensions (channels,events,time). or A TimeSeries instance with dimensions (channels,events,time) and xray.DataArray with dimensions (channels,events,time)

remove_fields(*fields_to_remove)

Return a new instance of the recarray with specified fields removed.

>>*<<fields_to_remove : {list of strings}

New Events instance without the specified fields.

ptsa.data.timeseries

class ptsa.data.timeseries.TimeSeries(data, tdim, samplerate, *args, **kwargs)

Bases: "dimarray.dimarray.DimArray"

A subclass of DimArray to hold timeseries data (i.e. data with a time dimension and associated sample rate). It also provides methods for manipulating the time dimension, such as resampling and filtering the data.

data [{array_like}] The time series data.

tdim [{str}] The name of the time dimension.

samplerate [{float}] The sample rate of the time dimension. Constrained to be of type float (any passed in value is converted to a float).

>>*<<args [{>>*<<args},optional] Additinal custom attributes

>><<kwargs** [{>>**<<kwargs},optional] Additional custom keyword attributes.

Useful additional (keyword) attributes include dims, dtype, and copy (see DimArray docstring for details).

DimArray

```
>>> import numpy as np
>>> import dimarray as da
>>> import ptsa.data.timeseries as ts
>>> observations = da.Dim(['a','b','c'],'obs')
>>> time = da.Dim(np.arange(4),'time')
>>> data = ts.TimeSeries(np.random.rand(3,4),'time',samplerate=1,
                         dims=[observations,time])
>>> data
TimeSeries([[ 0.51244513,  0.39930142,  0.63501339,  0.67071605],
   [ 0.46962664,  0.51071395,  0.46748319,  0.78265951],
   [ 0.85515317,  0.10996395,  0.41642481,  0.50561768]])
```

```
>>> data.samplerate
1.0
>>> data.tdim
'time'
```

baseline_corrected(base_range)

Return a baseline corrected timeseries by subtracting the average value in the baseline range from all other time points for each dimension.

base_range: {tuple} Tuple specifying the start and end time range (inclusive) for the baseline.

ts [{TimeSeries}] A TimeSeries instance with the baseline corrected data.

filtered(freq_range, filt_type='stop', order=4)

Filter the data using a Butterworth filter and return a new TimeSeries instance.

freq_range [{array_like}] The range of frequencies to filter.

filt_type = {scipy.signal.band_dict.keys()},optional Filter type.

order = {int} The order of the filter.

ts [{TimeSeries}] A TimeSeries instance with the filtered data.

remove_buffer(duration)

Remove the desired buffer duration (in seconds) and reset the time range.

duration [{int,float,({int,float},{int,float})}] The duration to be removed. The units depend on the samplerate: E.g., if samplerate is specified in Hz (i.e., samples per second), the duration needs to be specified in seconds and if samplerate is specified in kHz (i.e., samples per millisecond), the duration needs to be specified in milliseconds. A single number causes the specified duration to be removed from the beginning and end. A 2-tuple can be passed in to specify different durations to be removed from the beginning and the end respectively.

ts [{TimeSeries}] A TimeSeries instance with the requested durations removed from the beginning and/or end.

resampled(resampled_rate, window=None, loop_axis=None, num_mp_procs=0, pad_to_pow2=False)

Resample the data and reset all the time ranges.

Uses the resample function from scipy. This method seems to be more accurate than the decimate method.

resampled_rate [{float}] New sample rate to resample to.

window [{None,str,float,tuple}, optional] See scipy.signal.resample for details

loop_axis: {None,str,int}, optional Sometimes it might be faster to loop over an axis.

num_mp_procs: int, optional Whether to try and use multiprocessing to loop over axis. 0 means no multiprocessing >0 specifies num procs to use None means yes, and use all possible procs

pad_to_pow2: bool, optional Pad along the time dimension to the next power of 2 so that the resampling is much faster (experimental).

ts [{TimeSeries}] A TimeSeries instance with the resampled data.

scipy.signal.resample

taxis

Numeric time axis (read only).

ptsa.extensions package

C extension modules, for accelerated calculations.

Subpackages

ptsa.extensions.circular_stat package

Submodules

ptsa.extensions.circular_stat.circular_stat module

Module contents

ptsa.extensions.morlet package

Submodules

ptsa.extensions.morlet.morlet module

class ptsa.extensions.morlet.morlet.MorletWaveFFT

Bases: "object"

fft

init(width, freq, win_size, sample_freq)

len

len0

nt

class ptsa.extensions.morlet.morlet.MorletWaveletTransform(*args)

Bases: "object"

fft_buf

init(width, low_freq, high_freq, nf, sample_freq, signal_len)

init_flex(width, freqs, sample_freq, signal_len)

morlet_wave_ffts

multiphasevec(signal, powers, phases=None)

multiphasevec_c(signal, wavelets)

multiphasevec_complex(signal, wavelets)

multiphasevec_powers(signal, powers)

multiphasevec_powers_and_phases(signal, powers, phases)

n_freqs

n_plans

phase_and_pow_fcn

plan_for_inverse_transform

plan_for_signal

prod_buf

result_buf

set_output_type(output_type)

signal_buf

signal_len_

wavelet_pow_phase(signal, powers, phases, wavelets)

wavelet_pow_phase_py(signal, powers, phases, wavelets)

wv_both(r, i, powers, phase, wavelets)

wv_complex(r, i, powers, phase, wavelet_complex)

wv_phase(r, i, powers, phase, wavelets)

wv_pow(r, i, powers, phase, wavelets)

class ptsa.extensions.morlet.morlet.MorletWaveletTransformMP(cpus=1)

Bases: "object"

compute_wavelets_threads()

compute_wavelets_worker_fcn(thread_no)

index(i, j, stride)

initialize_signal_props(sample_freq)

initialize_wavelet_props(width, freqs)

prepare_run()

set_num_freq(num_freq)

set_output_type(output_type)

set_signal_array(signal_array)

set_wavelet_complex_array(wavelet_complex_array)

set_wavelet_phase_array(wavelet_phase_array)

set_wavelet_pow_array(wavelet_pow_array)

Module contents

ptsa.plotting package

Submodules

ptsa.plotting.topo.topoplot(values=None, labels=None, sensors=None, axes=None, center=(0, 0), nose_dir=0.0, radius=0.5, head_props=None, sensor_props=None, label_props=None, contours=15, contour_props=None, resolution=400, cmap=None, axis_props='off', plot_mask='circular', plot_radius_buffer=0.2)

Plot a topographic map of the scalp in a 2-D circular view (looking down at the top of the head).

values [{None, array-like}, optional] Values to plot. There must be one value for each electrode.

labels [{None, array-like}, optional] Electrode labels/names to plot. There must be one for each electrode.

sensors [{None, tuple of floats}, optional] Polar coordinates of the sensor locations. If not None, sensors[0] specifies the angle (in degrees) and sensors[1] specifies the radius.

axes [{matplotlib.axes}, optional] Axes to which the topoplot should be added.

center [{tuple of floats}, optional] x and y coordinates of the center of the head.

nose_dir [{float}, optional] Angle (in degrees) where the nose is pointing. 0 is up, 90 is left, 180 is down, 270 is right, etc.

radius [{float}, optional] Radius of the head.

head_props [dict] Dictionary of head properties. See default_head_props for choices.

sensor_props [dict] Dictionary of sensor properties. See options for scatter in mpl and default_sensor_props.

label_props [dict] Dictionary of sensor label properties. See options for text in mpl and default_label_props.

contours [{int}, optional] Number of contours.

contour_props [dict] Dictionary of contour properties. See options for contour in mpl and default_contour_props.

resolution [{int}, optional] Resolution of the interpolated grid. Higher numbers give smoother edges of the plot, but increase memory and computational demands.

cmap [{None,matplotlib.colors.LinearSegmentedColormap}, optional] Color map for the contour plot. If colMap==None, the default color map is used.

axis_props [{str}, optional] Axis properties.

plot_mask [{str}, optional] The mask around the plotted values. 'linear' conects the outer electrodes with straight lines, 'circular' draws a circle around the outer electrodes (see plot_radius_buffer).

plot_radius_buffer [float, optional] Buffer outside the electrode circumference for generating interpolated values with a circular mask. This should be greater than zero to aviod interpolation errors.

Module contents

ptsa.stats package

Submodules

ptsa.stats.cluster module

ptsa.stats.cluster.find_clusters(x, threshold, tail=0, connectivity=None)

For a given 1d-array (test statistic), find all clusters which are above/below a certain threshold. Returns a list of 2-tuples.

x: 1D array Data

threshold: float Where to threshold the statistic

tail [-1 | 0 | 1] Type of comparison

connectivity [sparse matrix in COO format] Defines connectivity between features. The matrix is assumed to be symmetric and only the upper triangular half is used. Defaut is None, i.e, no connectivity.

clusters: list of slices or list of arrays (boolean masks) We use slices for 1D signals and mask to multidimensional arrays.

sums: array Sum of x values in clusters

ptsa.stats.cluster.pval_from_histogram(T, H0, tail)

Get p-values from stats values given an H0 distribution

For each stat compute a p-value as percentile of its statistics within all statistics in surrogate data

ptsa.stats.cluster.sensor_neighbors(sensor_locs)

Calculate the neighbor connectivity based on Delaunay triangulation of the sensor locations.

sensor_locs should be the x and y values of the 2-d flattened sensor locs.

ptsa.stats.cluster.simple_neighbors_1d(n)

Return connectivity for simple 1D neighbors.

ptsa.stats.cluster.sparse_dim_connectivity(dim_con)

Create a sparse matrix capturing the connectivity of a conjunction of dimensions.

ptsa.stats.cluster.tfce(x, dt=0.1, E=0.6666666666666666, H=2.0, tail=0, connectivity=None)

Threshold-Free Cluster Enhancement.

ptsa.stats.nonparam module

ptsa.stats.nonparam.gen_perms(dat, group_var, nperms)

Generate permutations within a group variable, but across conditions.

There is no need to sort your data as this method will shuffle the indices properly.

ptsa.stats.nonparam.permutation_test(X, Y=None, parametric=True, iterations=1000)

Perform a permutation test on paired or non-paired data.

Observations must be on the first axis.

ptsa.stats.stat_helper module

ptsa.stats.stat_helper.fdr_correction(pvals, alpha=0.05, method='indep')

P-value correction with False Discovery Rate (FDR)

Correction for multiple comparison using FDR.

This covers Benjamini/Hochberg for independent or positively correlated and Benjamini/Yekutieli for general or negatively correlated tests.

pvals [array_like] set of p-values of the individual tests.

alpha [float] error rate

method ['indep' | 'negcorr'] If 'indep' it implements Benjamini/Hochberg for independent or if 'negcorr' it corresponds to Benjamini/Yekutieli.

reject [array, bool] True if a hypothesis is rejected, False if not

pval_corrected [array] pvalues adjusted for multiple hypothesis testing to limit FDR

Reference: Genovese CR, Lazar NA, Nichols T. Thresholding of statistical maps in functional neuroimaging using the false discovery rate. Neuroimage. 2002 Apr;15(4):870-8.

Module contents

Submodules

ptsa.emd module

Empirical Mode Decomposition

ptsa.emd.calc_inst_info(modes, samplerate)

Calculate the instantaneous frequency, amplitude, and phase of each mode.

ptsa.emd.eemd(data, noise_std=0.2, num_ensembles=100, num_sifts=10)

Ensemble Empirical Mode Decomposition (EEMD)

*** Must still add in post-processing with EMD ***

ptsa.emd.emd(data, max_modes=10)

Calculate the Emprical Mode Decomposition of a signal.

ptsa.filt module

ptsa.filt.buttfilt(dat, freq_range, sample_rate, filt_type, order, axis=-1)

Wrapper for a Butterworth filter.

ptsa.filt.decimate(x, q, n=None, ftype='iir', axis=-1)

Downsample the signal x by an integer factor q, using an order n filter

By default, an order 8 Chebyshev type I filter is used or a 30 point FIR filter with hamming window if ftype is 'fir'.

(port to python of the GNU Octave function decimate.)

Inputs: x – the signal to be downsampled (N-dimensional array) q – the downsampling factor n – order of the filter (1 less than the length of the filter for a

'fir' filter)

ftype – type of the filter; can be 'iir' or 'fir' axis – the axis along which the filter should be applied

Outputs: y – the downsampled signal

ptsa.filt.filtfilt(b, a, x)

ptsa.filt.filtfilt2(b, a, x, axis=-1)

ptsa.filt.firls(N, f, D=None)

Least-squares FIR filter. N – filter length, must be odd f – list of tuples of band edges

Units of band edges are Hz with 0.5 Hz == Nyquist and assumed 1 Hz sampling frequency

D – list of desired responses, one per band

ptsa.filt.lfilter_zi(b, a)

ptsa.filtfilt module

ptsa.filtfilt.filtfilt(b, a, x, axis=-1, padtype='odd', padlen=None)

A forward-backward filter.

This function applies a linear filter twice, once forward and once backwards. The combined filter has linear phase.

Before applying the filter, the function can pad the data along the given axis in one of three ways: odd, even or constant. The odd and even extensions have the corresponding symmetry about the end point of the data. The constant extension extends the data with the values at end points. On both the forward and backwards passes, the initial condition of the filter is found by using lfilter_zi and scaling it by the end point of the extended data.

Parameters

- **b** (*array_like*, **1-D*) –
- **numerator coefficient vector of the filter.** (*The*) –
- **a** (*array_like*, **1-D*) –
- **denominator coefficient vector of the filter. If a****[****0**]****** (*The*) –
- **not 1**, **then both a and b are normalized by a****[****0**]****** (*is*) –
- **x** (*array_like*) –
- **array of data to be filtered.** (*The*) –
- **axis** (*int*, **optional*) –
- **axis of x to which the filter is applied.** (*The*) –
- **is -1.** (*Default*) –
- **padtype** (*str* or **None**, **optional*) –
- **be 'odd'**, **'even'**, **'constant'**, or **None. This determines the** (*Must*) –
- **of extension to use for the padded signal to which the filter** (*type*) –
- **applied. If padtype is None**, **no padding is used. The default** (*is*) –
- **'odd'.** (*is*) –
- **padlen** (*int* or **None**, **optional*) –
- **number of elements by which to extend x at both ends of** (*The*) –
- **before applying the filter. This value must be less than** (*axis*) –
- **padlen=0 implies no padding.** (*x.shape**[**axis*]**-1.*) –
- **default value is 3*max****(****len****(****a**)********,****len****(**b**)********)****** (*The*) –

Returns

- **y** (*ndarray*)
- *The filtered output, an array of type numpy.float64 with the same*
- shape as *x*.

`lfilter_zi()`, `lfilter()`

-[Examples]-

First we create a one second signal that is the sum of two pure sine waves, with frequencies 5 Hz and 250 Hz, sampled at 2000 Hz.

```
>>> t = np.linspace(0, 1.0, 2001)
>>> xlow = np.sin(2 * np.pi * 5 * t)
>>> xhigh = np.sin(2 * np.pi * 250 * t)
>>> x = xlow + xhigh
```

Now create a lowpass Butterworth filter with a cutoff of 0.125 times the Nyquist rate, or 125 Hz, and apply it to x with filtfilt. The result should be approximately xlow, with no phase shift.

```
>>> from scipy.signal import butter
>>> b, a = butter(8, 0.125)
>>> y = filtfilt(b, a, x, padlen=150)
>>> np.abs(y - xlow).max()
9.1086182074789912e-06
```

We get a fairly clean result for this artificial example because the odd extension is exact, and with the moderately long padding, the filter's transients have dissipated by the time the actual data is reached. In general, transient effects at the edges are unavoidable.

ptsa.filtfilt.lfilter_zi(b, a)

Compute an initial state *zi* for the lfilter function that corresponds to the steady state of the step response.

A typical use of this function is to set the initial state so that the output of the filter starts at the same value as the first element of the signal to be filtered.

Parameters

- **a** (*b*,***) –
- **IIR filter coefficients. See scipy.signal.lfilter for more** (*The*) –
- **information.** –

Returns

- **zi** (*1-D ndarray*)
- *The initial state for the filter.*

-[Notes]-

A linear filter with order m has a state space representation (A, B, C, D), for which the output y of the filter can be expressed as:

z(n+1) = A*z(n) + B*x(n) y(n) = C*z(n) + D*x(n)

where z(n) is a vector of length m, A has shape (m, m), B has shape (m, 1), C has shape (1, m) and D has shape (1, 1) (assuming x(n) is a scalar). lfilter_zi solves:

zi = A*zi + B

In other words, it finds the initial condition for which the response to an input of all ones is a constant.

Given the filter coefficients *a* and *b*, the state space matrices for the transposed direct form II implementation of the linear filter, which is the implementation used by scipy.signal.lfilter, are:

A = scipy.linalg.companion(a).T B = b[1:] - a[1:]*b[0]

assuming *a[0]* is 1.0; if *a[0]* is not 1, *a* and *b* are first divided by a[0].

-[Examples]-

The following code creates a lowpass Butterworth filter. Then it applies that filter to an array whose values are all 1.0; the output is also all 1.0, as expected for a lowpass filter. If the *zi* argument of *lfilter* had not been given, the output would have shown the transient signal.

```
>>> from numpy import array, ones
>>> from scipy.signal import lfilter, lfilter_zi, butter
>>> b, a = butter(5, 0.25)
>>> zi = lfilter_zi(b, a)
>>> y, zo = lfilter(b, a, ones(10), zi=zi)
>>> y
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Another example:

```
>>> x = array([0.5, 0.5, 0.5, 0.0, 0.0, 0.0, 0.0])
>>> y, zf = lfilter(b, a, x, zi=zi*x[0])
>>> y
array([ 0.5 , 0.5 , 0.5 , 0.49836039, 0.48610528,
0.44399389, 0.35505241])
```

Note that the *zi* argument to *lfilter* was computed using *lfilter_zi* and scaled by *x[0]*. Then the output *y* has no transient until the input drops from 0.5 to 0.0.

ptsa.fixed_scipy module ——————————==

Functions that are not yet included or fixed in a stable scipy release are provided here until they are easily available in scipy.

ptsa.fixed_scipy.morlet(M, w=5.0, s=1.0, complete=True)

Complex Morlet wavelet.

Parameters

- **M** (*int*) – Length of the wavelet.
- **w** (*float*) – Omega0
- **s** (*float*) – Scaling factor, windowed from -s*2*pi to +s*2*pi.
- **complete** (*bool*) – Whether to use the complete or the standard version.
- **Notes** –
- **——** –
- **standard version** (*The*) –

  pi**-0.25 * exp(1j*w*x) * exp(-0.5*(x**2))

  This commonly used wavelet is often referred to simply as the Morlet wavelet. Note that, this simplified version can cause admissibility problems at low values of w.
- **complete version** (*The*) –

  pi**-0.25 * (exp(1j*w*x) - exp(-0.5*(w**2))) * exp(-0.5*(x**2))

The complete version of the Morlet wavelet, with a correction term to improve admissibility. For w greater than 5, the correction term is negligible.

- **that the energy of the return wavelet is not normalised** (*Note*) –
- **to s.** (*according*) –
- **fundamental frequency of this wavelet in Hz is given** (*The*) –
- **f = 2*s*w*r / M where r is the sampling rate.** (*by*) –

ptsa.helper ———==

ptsa.helper.cart2pol(x, y, z=None, radians=True)

Converts corresponding Cartesian coordinates x, y, and (optional) z to polar (or, when z is given, cylindrical) coordinates angle (theta), radius, and z. By default theta is returned in radians, but will be converted to degrees if radians==False.

ptsa.helper.centered(arr, newsize)

Return the center newsize portion of the input array.

Parameters

- **arr** (*{array}*) – Input array
- **newsize** (*{tuple of ints}*) – A tuple specifing the size of the new array.

Returns

Return type A center slice into the input array

Note: Adapted from scipy.signal.signaltools._centered

ptsa.helper.deg2rad(degrees)

Convert degrees to radians.

ptsa.helper.getargspec(obj)

Get the names and default values of a callable's arguments

A tuple of four things is returned: (args, varargs, varkw, defaults).

- args is a list of the argument names (it may contain nested lists).
- varargs and varkw are the names of the * and ** arguments or None.
- defaults is a tuple of default argument values or None if there are no default arguments; if this tuple has n elements, they correspond to the last n elements listed in args.

Unlike inspect.getargspec(), can return argument specification for functions, methods, callable objects, and classes. Does not support builtin functions or methods.

See http://kbyanc.blogspot.com/2007/07/python-more-generic-getargspec.html

ptsa.helper.lock_file(filename, lockdirpath=None, lockdirname=None)

ptsa.helper.next_pow2(n)

Returns p such that 2 ** p >= n

ptsa.helper.pad_to_next_pow2(x, axis=0)

Pad an array with zeros to the next power of two along the specified axis.

Note: This is much easier with numpy version 1.7.0, which has a new pad method.

ptsa.helper.pol2cart(theta, radius, z=None, radians=True)

Converts corresponding angles (theta), radii, and (optional) height (z) from polar (or, when height is given, cylindrical) coordinates to Cartesian coordinates x, y, and z. Theta is assumed to be in radians, but will be converted from degrees if radians==False.

ptsa.helper.rad2deg(radians)

Convert radians to degrees.

ptsa.helper.release_file(filename, lockdirpath=None, lockdirname=None)

ptsa.helper.repeat_to_match_dims(x, y, axis=-1)

ptsa.helper.reshape_from_2d(data, axis, dshape)

Reshape data from 2D back to specified dshape.

ptsa.helper.reshape_to_2d(data, axis)

Reshape data to 2D with specified axis as the 2nd dimension.

ptsa.hilbert

ptsa.hilbert.hilbert_pow(dat_ts, bands=None, pad_to_pow2=False, verbose=True)

ptsa.iwasobi

class ptsa.iwasobi.IWASOBI(ar_max=10, rmax=0.99, eps0=5e-07)

Implements algorithm WASOBI for blind source separation of AR sources in a fast way, allowing separation up to 100 sources in the running time of the order of tens of seconds.

Ported from MATLAB code by Jakub Petkov / Petr Tichavsky

CRLB4(ARC)

function ISR = CRLB4(ARC) % % CRLB4(ARC) generates the CRLB for gain matrix elements (in term % of ISR) for blind separation of K Gaussian autoregressive sources % whose AR coefficients (of the length M, where M-1 is the AR order) % are stored as columns in matrix ARC.

THinv5(phi, K, M, eps)

function G=THinv5(phi,K,M,eps) % %%%% Implements fast (complexity O(M*K^2)) %%%% computation of the following piece of code: % %C=[]; %for im=1:M % A=toeplitz(phi(1:K,im),phi(1:K,im)')+hankel(phi(1:K,im),phi(K:2*K-1,im)')+eps(im)*eye(K); % C=[C inv(A)]; %end % % DEFAULT PARAMETERS: M=2; phi=randn(2*K-1,M); eps=randn(1,2); % SIZE of phi SHOULD BE (2*K-1,M). % SIZE of eps SHOULD BE (1,M).

ar2r(a)

%%%%% Computes covariance function of AR processes from %%%%% the autoregressive coefficients using an inverse Schur algorithm %%%%% and an inverse Levinson algorithm (for one column it is equivalent to %%%%% "rlevinson.m" in matlab) %

armodel(R, rmax)

function [AR,sigmy]=armodel(R,rmax) % % to compute AR coefficients of the sources given covariance functions % but if the zeros have magnitude > rmax, the zeros are pushed back. %

corr_est(x, T, q)

function R_est=corr_est(x,T,q) # %

uwajd(M, maxnumiter=20, W_est0=None)

function [W_est Ms]=uwajd(M,maxnumiter,W_est0) % % my approximate joint diagonalization with uniform weights % % Input: M .... the matrices to be diagonalized, stored as [M1 M2 ... ML] % West0 ... initial estimate of the demixing matrix, if available % % Output: W_est .... estimated demixing matrix % such that W_est * M_k * W_est' are roughly diagonal % Ms .... diagonalized matrices composed of W_est*M_k*W_est' % crit ... stores values of the diagonalization criterion at each % iteration %

wajd(M, H, W_est0=None, maxnumit=100)

function [W_est Ms]=wajd(M,H,W_est0,maxnumit) % % my approximate joint diagonalization with non-uniform weights % % Input: M .... the matrices to be diagonalized, stored as [M1 M2 ... ML] % H .... diagonal blocks of the weight matrix stored similarly % as M, but there is dd2 blocks, each of the size L x L % West0 ... initial estimate of the demixing matrix, if available % maxnumit ... maximum number of iterations % % Output: W_est .... estimated demixing matrix % such that W_est * M_k * W_est' are roughly diagonal % Ms .... diagonalized matrices composed of W_est*M_k*W_est' % crit ... stores values of the diagonalization criterion at each % iteration % %

weights(Ms, rmax, eps0)

function [H ARC]=weights(Ms,rmax,eps0) %

ptsa.iwasobi.iwasobi(data, ar_max=10, rmax=0.99, eps0=5e-07)

ptsa.pca

ptsa.pca.pca(X, ncomps=None, eigratio=1000000.0)

Principal components analysis

% [W,Y] = pca(X,NBC,EIGRATIO) returns the PCA matrix W and the principal % components Y corresponding to the data matrix X (realizations % columnwise). The number of components is NBC components unless the % ratio between the maximum and minimum covariance eigenvalue is below % EIGRATIO. In such a case, the function will return as few components as % are necessary to guarantee that such ratio is greater than EIGRATIO.

ptsa.sandbox

ptsa.version

Version management module.

ptsa.version.versionAtLeast(someString)

Check that the current ptsa Version >= argument string's version.

ptsa.version.versionWithin(str1, str2)

Check that the current ptsa version is in the version-range described by the 2 argument strings.

ptsa.versionString

PTSA Version

ptsa.wavelet

ptsa.wavelet.calcPhasePow(freqs, dat, samplerate, axis=-1, width=5, verbose=False, to_return='both')

Calculate phase and power over time with a Morlet wavelet.

You can optionally pass in downsample, which is the samplerate to decimate to following the power/phase calculation.

As always, it is best to pass in extra signal (a buffer) on either side of the signal of interest because power calculations and decimation have edge effects.

ptsa.wavelet.convolve_wave(wav, eegdat)

ptsa.wavelet.fconv_multi(in1, in2, mode='full')

Convolve multiple 1-dimensional arrays using FFT.

Calls scipy.signal.fft on every row in in1 and in2, multiplies every possible pairwise combination of the transformed rows, and returns an inverse fft (by calling scipy.signal.ifft) of the result. Therefore the output array has as many rows as the product of the number of rows in in1 and in2 (the number of colums depend on the mode).

Parameters

- **in1** (*{array_like}*) – First input array. Must be arranged such that each row is a 1-D array with data to convolve.
- **in2** (*{array_like}*) – Second input array. Must be arranged such that each row is a 1-D array with data to convolve.
- **mode** (*{'full'*,**'valid','same'}**,**optional*) – Specifies the size of the output. See the docstring for scipy.signal.convolve() for details.

Returns

- *Array with in1.shape[0]*in2.shape[0] rows with the convolution of*
- *the 1-D signals in the rows of in1 and in2.*

ptsa.wavelet.iswt(coefficients, wavelet)

Inverse Stationary Wavelet Transform

Input parameters:

coefficients approx and detail coefficients, arranged in level value exactly as output from swt: e.g. [(cA1, cD1), (cA2, cD2), ..., (cAn, cDn)]

wavelet Either the name of a wavelet or a Wavelet object

ptsa.wavelet.morlet(freq, t, width)

Generate a Morlet wavelet for specified frequncy for times t. The wavelet will be normalized so the total energy is 1. width defines the >>''<<width" of the wavelet in cycles. A value >= 5 is suggested.

ptsa.wavelet.morlet_multi(freqs, widths, samplerates, sampling_windows=7, complete=True)

Calculate Morlet wavelets with the total energy normalized to 1.

Calls the scipy.signal.wavelet.morlet() function to generate Morlet wavelets with the specified frequencies, samplerates, and widths (in cycles); see the docstring for the scipy morlet function for details. These wavelets are normalized before they are returned.

Parameters

- **freqs** (*{float*, **array_like of floats}*) – The frequencies of the Morlet wavelets.
- **widths** (*{float*, **array_like floats}*) – The width(s) of the wavelets in cycles. If only one width is passed in, all wavelets have the same width. If len(widths)==len(freqs), each frequency is paired with a corresponding width. If 1<len(widths)<len(freqs), len(freqs) must be evenly divisible by len(widths) (i.e., len(freqs)%len(widths)==0). In this case widths are repeated such that (1/len(widths))*len(freq) neigboring wavelets have the same width – e.g., if len(widths)==2, the the first and second half of the wavelets have widths of widths[0] and width[1] respectively, and if len(widths)==3 the first, middle, and last third of wavelets have widths of widths[0], widths[1], and widths[2] respectively.
- **samplerates** (*{float*, **array_like floats}*) – The sample rate(s) of the signal (e.g., 200 Hz).
- **sampling_windows** (*{float*, **array_like of floates}**,**optional*) – How much of the wavelets is sampled. As sampling_window increases, the number of samples increases and thus the samples near the edge approach zero increasingly closely. If desired different values can be specified for different wavelets (the syntax for multiple sampling windows is the same as for widths). One value >= 7 is recommended.
- **complete** (*{bool}**,**optional*) – Whether to generate a complete or standard approximation to the complete version of a Morlet wavelet. Complete should be True, especially for low (<=5) values of width. See scipy.signal.wavelet.morlet() for details.

Returns

Return type A 2-D (frequencies * samples) array of Morlet wavelets.

-[Notes]-

The in scipy versions <= 0.6.0, the scipy.signal.wavelet.morlet() code contains a bug. Until it is fixed in a stable release, this code calls a local fixed version of the scipy function.

-[Examples]-

```
>>> wavelet = morlet_multi(10,5,200)
>>> wavelet.shape
(1, 112)
>>> wavelet = morlet_multi([10,20,30],5,200)
>>> wavelet.shape
(3, 112)
>>> wavelet = morlet_multi([10,20,30],[5,6,7],200)
>>> wavelet.shape
(3, 112)
```

ptsa.wavelet.phasePow1d(freq, dat, samplerate, width)

Calculate phase and power for a single freq and 1d signal.

ptsa.wavelet.phasePow2d(freq, dat, samplerate, width)

Calculate phase and power for a single freq and 2d signal of shape (events,time).

This will be slightly faster than phasePow1d for multiple events because it only calculates the Morlet wavelet once.

ptsa.wavelet.phase_pow_multi(freqs, dat, samplerates=None, widths=5, to_return='both', time_axis=-1, conv_dtype=<type 'numpy.complex64'>, freq_name='freqs', **kwargs)

Calculate phase and power with wavelets across multiple events.

Calls the morlet_multi() and fconv_multi() functions to convolve dat with Morlet wavelets. Phase and power over time across all events are calculated from the results. Time/samples should include a buffer before onsets and after offsets of the events of interest to avoid edge effects.

Parameters

- **freqs** (*{int*, **float**, **array_like of ints** or **floats}*) – The frequencies of the Morlet wavelets.
- **dat** (*{array_like}*) – The data to determine the phase and power of. Sample rate(s) and time dimension must be specified as attributes of dat or in the key word arguments. The time dimension should include a buffer to avoid edge effects.
- **samplerates** (*{float*, **array_like of floats}**, **optional*) – The sample rate(s) of the signal. Must be specified if dat is not a TimeSeries instance. If dat is a TimeSeries instance, any value specified here will be replaced by the value stored in the samplerate attribute.
- **widths** (*{float*, **array_like of floats}**,**optional*) – The width(s) of the wavelets in cycles. See docstring of morlet_multi() for details.
- **to_return** (*{'both'*,**'power','phase'}**, **optional*) – Specify whether to return power, phase, or both.
- **time_axis** (*{int}*,**optional*) – Index of the time/samples dimension in dat. Must be specified if dat is not a TimeSeries instance. If dat is a TimeSeries instance any value specified here will be replaced by the value specified in the tdim attribute.
- **conv_dtype** (*{numpy.complex}***,**optional*) – Data type for the convolution array. Using a larger dtype (e.g., numpy.complex128) can increase processing time. This value influences the dtype of the output array. In case of numpy.complex64 the dtype of the output array is numpy.float32. Higher complex dtypes produce higher float dtypes in the output.
- **freq_name** (*{string}*,**optional*) – Name of frequency dimension of the returned TimeSeries object (only used if dat is a TimeSeries instance).
- ****kwargs** ({>>**<<kwargs},optional) – Additional key word arguments to be passed on to morlet_multi().

Returns

- *Array(s) of phase and/or power values as specified in to_return. The*
- *returned array(s) has/have one more dimension than dat. The added*
- *dimension is for the frequencies and is inserted as the first*
- *dimension.*

ptsa.wavelet.phase_pow_multi_old(freqs, dat, samplerates, widths=5, to_return='both', time_axis=-1, freq_axis=0, conv_dtype=<type 'numpy.complex64'>, **kwargs)

Calculate phase and power with wavelets across multiple events.

Calls the morlet_multi() and fconv_multi() functions to convolve dat with Morlet wavelets. Phase and power over time across all events are calculated from the results. Time/samples should include a buffer before onsets and after offsets of the events of interest to avoid edge effects.

Parameters

- **freqs** (*{int*, **float**, **array_like of ints** or **floats}*) – The frequencies of the Morlet wavelets.
- **dat** (*{array_like}*) – The data to determine the phase and power of. Time/samples must be last dimension and should include a buffer to avoid edge effects.
- **samplerates** (*{float*, **array_like of floats}*) – The sample rate(s) of the signal (e.g., 200 Hz).
- **widths** (*{float*, **array_like of floats}*) – The width(s) of the wavelets in cycles. See docstring of morlet_multi() for details.
- **to_return** (*{'both'*,**'power','phase'}**, **optional*) – Specify whether to return power, phase, or both.
- **time_axis** (*{int}*,**optional*) – Index of the time/samples dimension in dat. Should be in {-1,0,len(dat.shape)}
- **freq_axis** (*{int}*,**optional*) – Index of the frequency dimension in the returned array(s). Should be in {0, time_axis, time_axis+1,len(dat.shape)}.
- **conv_dtype** (*{numpy.complex}*,**optional*) – Data type for the convolution array. Using a larger dtype (e.g., numpy.complex128) can increase processing time. This value influences the dtype of the output array. In case of numpy.complex64 the dtype of the output array is numpy.float32. Higher complex dtypes produce higher float dtypes in the output.
- ****kwargs** ({>>**<<kwargs},optional) – Additional key word arguments to be passed on to morlet_multi().

Returns

- *Array(s) of phase and/or power values as specified in to_return. The*
- *returned array(s) has/have one more dimension than dat. The added*
- *dimension is for the frequencies and is inserted at freq_axis.*

ptsa.wavelet.swt(data, wavelet, level=None)

Stationary Wavelet Transform

This version is 2 orders of magnitude faster than the one in pywt even though it uses pywt for all the calculations.

Input parameters:

data One-dimensional data to transform

wavelet Either the name of a wavelet or a Wavelet object

level Number of levels

ptsa.wavelet.tsPhasePow(freqs, tseries, width=5, resample=None, keepBuffer=False, verbose=False, to_return='both', freqDimName='freq')

Calculate phase and/or power on an TimeSeries, returning new TimeSeries instances.

ptsa.wavelet_obsolete

ptsa.wica

class ptsa.wica.WICA(data, samplerate, pure_range=None)

Bases: `object`

Clean data with the Wavelet-ICA method described here:

N.P. Castellanos, and V.A. Makarov (2006). 'Recovering EEG brain signals: Artifact suppression with wavelet enhanced independent component analysis' J. Neurosci. Methods, 158, 300–312.

Instead of using the Infomax ICA algorithm, we use the (much much faster) IWASOBI algorithm.

We also pick components to clean by only cleaning components that weigh heavily on the EOG electrodes.

This ICA algorithm works better if you pass in data that have been high-pass filtered to remove big non-neural fluctuations and drifts.

You do not have to run the ICA step on your entire dataset. Instead, it is possible to provide the start and end indicies for a continguous chunk of data that is 'clean' except for having lots of eyeblink examples. This range will also be used inside the wavelet-based artifact correction code to determine the best threshold for identifying artifacts. You do, however, want to try and make sure you provide enough samples for a good ICA decomposition. A good rule of thumb is 3*(N^2) where N is the number of channels/sources.

`ICA_weights`

clean(comp_inds=None, Kthr=2.5, num_mp_procs=0)

get_corrected()

get_loading(comp)

pick(EOG_elecs=[0, 1], std_fact=1.5)

ptsa.wica.find_blinks(dat, L, fast_rate=0.5, slow_rate=0.975, thresh=None)

Identify eyeblinks with fast and slow running averages.

ptsa.wica.remove_strong_artifacts(data, A, icaEEG, Comp, Kthr=1.25, F=256, Cthr=None, num_mp_procs=0)

% This function denoise high amplitude artifacts (e.g. ocular) and remove them from the % Independent Components (ICs). %

Ported and enhanced from Matlab code distributed by the authors of:

N.P. Castellanos, and V.A. Makarov (2006). 'Recovering EEG brain signals: Artifact suppression with wavelet enhanced independent component analysis' J. Neurosci. Methods, 158, 300–312.

% INPUT: % % icaEEG - matrix of ICA components (Nchanel x Nobservations) % % Comp - # of ICs to be denoised and cleaned (can be a vector) % % Kthr - threshold (multiplayer) for denoising of artifacts % (default Kthr = 1.15) % % F - acquisition frequency % (default F = 256 Hz) % % OUTPUT: % % opt - vector of threshold values used for filtering of corresponding % ICs % % NOTE: If a component has no artifacts of a relatively high amplitude % the function will skip this component (no action), dispaly a % warning and the corresponding output "opt" will be set to zero.

ptsa.wica.wica_clean(data, samplerate=None, pure_range=(None, None), EOG_elecs=[0, 1], std_fact=1.5, Kthr=2.5, num_mp_procs=0)

Clean data with the Wavelet-ICA method described here:

N.P. Castellanos, and V.A. Makarov (2006). 'Recovering EEG brain signals: Artifact suppression with wavelet enhanced independent component analysis' J. Neurosci. Methods, 158, 300–312.

Instead of using the Infomax ICA algorithm, we use the (much much faster) IWASOBI algorithm.

We also pick components to clean by only cleaning components that weigh heavily on the EOG electrodes.

This ICA algorithm works better if you pass in data that have been high-pass filtered to remove big non-neural fluctuations and drifts.

You do not have to run the ICA step on your entire dataset. Instead, it is possible to provide the start and end indicies for a continguous chunk of data that is 'clean' except for having lots of eyeblink examples. This range will also be used inside the wavelet-based artifact correction code to determine the best threshold for identifying artifacts. You do, however, want to try and make sure you provide enough samples for a good ICA decomposition. A good rule of thumb is 3*(N^2) where N is the number of channels/sources.

Module contents

PTSA - The Python Time-Series Analysis toolbox.

ptsa.test(level=1, verbosity=1, flags=[])

Using NumpyTest test method.

Run test suite with level and verbosity.

level: None — do nothing, return None < 0 — scan for tests of level=abs(level),

don't run them, return TestSuite-list

> 0 — scan for tests of level, run them, return TestRunner

verbosity: >= 0 — show information messages > 1 — show warnings on missing tests

ptsa.testall(level=1, verbosity=1, flags=[])

Using NumpyTest testall method.

Run test suite with level and verbosity.

level: None — do nothing, return None < 0 — scan for tests of level=abs(level),

don't run them, return TestSuite-list

> 0 — scan for tests of level, run them, return TestRunner

verbosity: >= 0 — show information messages > 1 — show warnings on missing tests