API Documentation

API Documentation

April 28, 2014

Contents

1 Package meet

```
This is the Modular EEg Toolkit (MEET) for Python 2.

**************************************************************************
***Disclosure:                                                         ***
***-----------                                                         ***
***This software comes as it is - there might be errors at runtime     ***
***and results might be wrong although the code was tested and did     ***
***work as expected. Since resuluts might be wrong you must            ***
***absolutely not use this software for a medical purpuse - decisions***
***converning diagnosis, treatment or prophylaxis of any medical      ***
***condition mustn't rely on this software.                            ***
**************************************************************************

Only Python 2 is supported at the moment, however modifications should
be easy to do.

Dependencies:
-------------
-Python 2
-Numpy
-Scipy
-Matplotlib

Installation:
-------------
Using the usual procedure:

python setup.py build
python setup.py install (be sure that the python executable is Python 2)

Uninstallation:
---------------
if you use pip you can uninstall doing:

pip uninstall meet

Version Compatibility:
----------------------
I try to avoid incompatibilities when updating functions,
this however cannot be totaly avoided from time to time. However
functions are thoroughly tested.

Citation:
---------
If you use this software for scientific publications please give proper
citation.
In the moment please cite as (or similar)
G. Waterstraat, 2014. Modular EEg toolkit (MEET) for Python.
```

```
https://github.com/neurophysics/meet. Retrieved on <date>

There is a properly citable publication on the way as well which may be
cited additionally.

License:
--------
Copyright (c) 2014 Gunnar Waterstraat

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Author & Contact
----------------
Written by Gunnar Waterstraat
email: gunnar[dot]waterstraat[at]charite.de
```

1.1 Modules

- **spatfilt**: Spatial Filters
  (Section 10, p. 34)
- **sphere**: Spherical spline interpolation and CSD
  (Section 11, p. 39)
- **tf**: S transform (time-frequency transformation)
  (Section 12, p. 49)

2 Module meet._cSPoC

```
Implements canonical Source Power Correlation analysis (cSPoC)

    Reference:
    ----------
    Dahne, S., et al., Finding brain oscillations with power dependencies
    in neuroimaging data, NeuroImage (2014),
    http://dx.doi.org/10.1016/j.neuroimage.2014.03.075

Hidden submodule of the Modular EEg Toolkit - MEET for Python.
(Imported by spatfilt module)

This module implements some spatial filters such as CSP, CCA, CCAvReg,
bCSTP and QCA.
```

2.1 Functions

pattern_from_filter(*filter*, *X*)

```
Get a an activation pattern from the filter matrix
and the input data X.

Input:
------
filter - is a p (x k) numpy array, where p is the dimensionality of the
         data and k is the number of filters (if k == 1, the array may
         be 1-dimensional)
X - is an p x N (x tr) numpy array, where p is the dimensionality of
    the data, N is the number of datapoints (per trial, and tr is the
    number of trials)
    If X is 3d, the patterns are computed by collapsing the last 2 axes

Output:
-------
pattern - is a (k x) p numpy array of activation patterns, where p is
          the dimensionality of the data and k is the number of filters
          If the input filter array is 1d so will be this array
```

cSPoC(*X*, *Y*, *opt*='max', *num*=1, *log*=True, *bestof*=15)

```
canonical Soure Power Correlation analysis (cSPoC)

For the datasets X and Y, find a pair of linear filters wx and wy, such
that the correlation of the amplitude envelopes wx.T.dot(X) and
wy.T.dot(Y) is maximized.

Reference:
----------
Dahne, S., et al., Finding brain oscillations with power dependencies
in neuroimaging data, NeuroImage (2014),
http://dx.doi.org/10.1016/j.neuroimage.2014.03.075

Notes:
------
Datasets X and Y can be either 2d numpy arrays of shape
(channels x datapoints) or 3d array of shape
(channels x datapoints x trials). For 3d arrays the average envelope
in each trial is calculated. If log == True, then the log transform is
taken before the average inside the trial

The filters are in the columns of the filter matrices Wx and Wy,
for 2d input the data can be filtered as:

np.dot(Wx.T, X)

for 3d input:

np.tensordot(Wx, X, axes=(0,0))

Input:
------
-- X numpy array - the first dataset of shape px x N (x tr), where px
                   is the number of sensors, N the number of data-
                   points, tr the number of trials
-- Y is the second dataset of shape py x N (x tr)
-- opt {'max', 'min'} - determines whether the correlation coefficient
                        should be maximized - seeks for positive
                        correlations ('max', default);
                        or minimized - seeks for anti-correlations
                        ('min')
-- num int > 0 - determine the number of filter-pairs that will be
                 derived. This depends also on the ranks of X and Y,
                 if X and Y are 2d the number of filter pairs will be:
                 min([num, rank(X), rank(Y)]). If X and/or Y are 3d the
                 array is flattened into a 2d array before calculating
                 the rank
-- log {True, False} - compute the correlation between the log-
                       transformed envelopes, if datasets come in
                       epochs, then the log is taken before averaging
                       inside the epochs, defaults to True
-- bestof int > 0 - the number of restarts for the optimization of the
                    individual filter pairs. The best filter over all
                    these restarts with random initializations is
                    chosen, defaults to 15.

Output:
```

cSPoAC(*X*, *tau*=1, *opt*='max', *num*=1, *log*=True, *bestof*=15)

```
canonical Soure Power Auto-Correlation analysis (cSPoAC)

For the dataset X, find a linear filters wx, such
that the correlation of the amplitude envelopes wx.T.dot(X[:,:-tau])
and wx.T.dot(X[:,tau:]) is maximized, i.e. it seeks a spatial filter
to maximize the auto-correlation of amplitude envelopes for a shift
of tau.

The solution is inspired by and derived by the original cSPoC-Analysis

Reference:
----------
Dahne, S., et al., Finding brain oscillations with power dependencies
in neuroimaging data, NeuroImage (2014),
http://dx.doi.org/10.1016/j.neuroimage.2014.03.075

Notes:
------
Dataset X can be either a 2d numpy array of shape
(channels x datapoints) or 3d a array of shape
(channels x datapoints x trials). For 2d array tau denotes a lag in
the time domain. For a 3d array the average envelope in each trial is
calculated and tau denotes a trial-wise lag.
If log == True, then the log transform is taken before the average
inside the trial

The filters are in the columns of the filter matrices Wx and Wy,
for 2d input the data can be filtered as:

np.dot(Wx.T, X)

for 3d input:

np.tensordot(Wx, X, axes=(0,0))

Input:
------
-- X numpy array - the dataset of shape px x N (x tr), where px is the
                   number of sensors, N the number of data-points, tr
                   the number of trials
-- tau int - the lag to calculate the autocorrelation , if X.ndim==2,
             this is a time-wise lag, if X.ndim==3, this is a trial-
             wise lag.
-- opt {'max', 'min'} - determines whether the correlation coefficient
                        should be maximized - seeks for positive
                        correlations ('max', default);
                        or minimized - seeks for anti-correlations
                        ('min')
-- num int > 0 - determine the number of filters that will be derived.
                 This depends also on the rank of X, if X is 2d, the
                 number of filter pairs will be: min([num, rank(X)]).
                 If X is 3d, the array is flattened into a 2d array
                 before calculating the rank
-- log {True, False} - compute the correlation between the log-
                       transformed envelopes, if datasets come in
                       epochs, then the log is taken before averaging
```

2.2 Variables

Name	Description
__package__	**Value:** `'meet'`

3 Module meet._dot_new

A drop in replacement for numpy.dot

Avoid temporary copies of non C-contiguous arrays

The code is available here: http://pastebin.com/raw.php?i=M8TfbURi In future this will be available in numpy directly: https://github.com/numpy/numpy/pull/2730

3.1 Functions

dot(*A*, *B*, *out*=`None`)

A drop in replaement for numpy.dot Computes A.B optimized using fblas call

test_dot()

test_to_fix()

1d array, complex and 3d

3.2 Variables

Name	Description
__package__	**Value:** `'meet'`

4 Module meet._interp

```
hidden functions for interpolating EEG

Hidden submodule of the Modular EGg Toolkit - MEET for Python.

Author:
-------
Gunnar Waterstraat
gunnar[dot]waterstraat[at]charite.de
```

4.1 Functions

akima(x, y)

mchi(x, y)

Monotone cubic hermite interpolation

4.2 Variables

Name	Description
__package__	**Value:** `'meet'`

5 Module meet.basic

```
Basic functions for reading binaries and EEG manipulation

Submodule of the Modular EEg Toolkit - MEET for Python.

Author:
-------
Gunnar Waterstraat
gunnar[dot]waterstraat[at]charite.de
```

5.1 Functions

readBinary(*fname*, *num_channels*, *channels*='all', *readnum_dp*='all', *data_type*='float4', *buffermem*=512)

```
Read EEG from a binary file and output as numpy array.
The binary of a signal with k channels and n datapoints must be
of the type:

        t0   t1 ... ... tn-1
        ----------------------------
ch_0  | 0    k      ... (n-1)*k
ch_1  | 1    k+1    ... (n-1)*k+1
...   | ...  ...    ... ...
ch_k-1| k-1  2*k-1  ... n*k-1

The endianess of the runtime system is used.

Input:
------
-- fname - (str) - input file name
-- num_channels - int - total number of channels in the file
-- channels - numpy array OR 'all' - iterable of channels to read
              (starting with 0) if 'all', all channels are read
-- readnum_dp - int OR 'all' - number of datapints to read
-- data_type - str - any of 'int2', 'int4', 'int8', 'float4',
                     'float8', 'float16' where the digit determins
                     the number of bytes (= 8 bits) for each element
-- buffermem - float  - number of buffer to us in MegaBytes

Output:
-------
-- data - numpy array - data shaped k x n where k is number of
                        channels and n is number of datapoints

Example:
--------
>>> readBinary(_path.join(_path.join(_packdir, 'test_data'),            'sample.dat'), 2, data_ty
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

interpolateEEG(*data*, *markers*, *win*, *interpolate_type*='mchs')

```
Interpolates segemnets in the data

Input:
------
-- data - one or two dimensional array
          1st dimension: channels (can be ommited if single channel)
          2nd dimension: datapoints
-- markers - marker positions arranged in 1d array
-- win - iterable of len 2 - determining the window in datapoints to
         be interpolated (win[0] is in, win[1] is out of the window)
-- interpolate_type: ['linear', 'mchs', 'akima'] - linear or
                     Monotone Cubic Hermite Spline
   or Akima interpolation

Output:
-------
interpolated dataset

Examples:
--------
>>> data = _np.arange(20, dtype=float).reshape(2,-1)
>>> interpolateEEG(data, [5], [-1,2], 'linear')
array([[ 0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.]])
>>> interpolateEEG(data, [5], [-1,2], 'mchs')
array([[ 0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.]])
>>> interpolateEEG(data, [5], [-1,2], 'akima')
array([[ 0.   ,   1.   ,   2.   ,   3.   ,   3.625,   5.   ,   6.375,
         7.   ,   8.   ,   9.   ],
       [ 10.   ,  11.   ,  12.   ,  13.   ,  13.625,  15.   ,  16.375,
        17.   ,  18.   ,  19.   ]])
```

epochEEG(*data*, *marker*, *win*)

```
Arange the dataset into trials (=epochs) according to the marker and
window.

markers and the window borders are sorted in ascending order.

Input:
------
-- data - numpy array - 1st dim channels (can be ommited if single
                                          channel)
                       2nd dim datapoints
-- marker - iterable - the marker
-- win - iterable of len 2 - determing the start and end of epchos
         in dp (win[0] is in, win[1] is out of the window)

Output:
-------
-- epochs - numpy array - dimension one more then data input
                        - 1st dim: channel (might be ommited - see
                                            above)
                        - 2nd dim: epoch length = win[1] - win[0]
                        - 3rd dim: number of epochs

Example:
--------
>>> data = _np.arange(20, dtype=float).reshape(2,-1)
>>> epochEEG(data, [3,5,7], [-2,2])
array([[[  1.,   3.,   5.],
        [  2.,   4.,   6.],
        [  3.,   5.,   7.],
        [  4.,   6.,   8.]],
<BLANKLINE>
       [[ 11.,  13.,  15.],
        [ 12.,  14.,  16.],
        [ 13.,  15.,  17.],
        [ 14.,  16.,  18.]]])
```

calculateRMS(*data*, *axis*=-1)

```
Calculate rms value of the input data along the indicated axis

Input:
------
-- data - numpy array - input data
-- axis - int - axis along which the rms is calculated; if None, the
                flattened array is used
Output:
-------
-- rms value along the indicated axis

Example:
--------
>>> data = _np.arange(20, dtype=float).reshape(2,-1)
>>> calculateRMS(data, None)
11.113055385446435
>>> calculateRMS(data, 0)
array([  7.07106781,   7.81024968,   8.60232527,   9.43398113,
        10.29563014,  11.18033989,  12.08304597,  13.        ,
        13.92838828,  14.86606875])
>>> calculateRMS(data, 1)
array([  5.33853913,  14.7817455 ])
```

getMarker(*marker*, *width*=50, *mindist*=100)

```
Gets position of markers from the trigger channel
GetMarkerPosFromData(marker)

input:
-- marker - one-dimensional array with trigger channel - each
            impulse or zero crossing is treated a marker
--width - int - calculates the local mean in window of size width
                - defaults to 50
--mindist - int - minimal distance between triggers in dp
                  - defaults to 100
output:
-- marker - one-dimensional array containing trigger positions

Example:
--------
>>> x = _np.ones(1000)
>>> x[200:400] = -1
>>> x[600:800] = -1
>>> getMarker(x)
array([200, 400,  600,  800])
```

5.2 Variables

Name	Description
__package__	**Value:** `'meet'`

6 Module meet.cSPoC

```
Implements canonical Source Power Correlation analysis (cSPoC)

    Reference:
    ----------
    Dahne, S., et al., Finding brain oscillations with power dependencies
    in neuroimaging data, NeuroImage (2014),
    http://dx.doi.org/10.1016/j.neuroimage.2014.03.075
```

6.1 Functions

cSPoC(*X*, *Y*, *opt*='max', *num*=1, *log*=True, *bestof*=15)

```
canonical Soure Power Correlation analysis (cSPoC)

For the datasets X and Y, find a pair of linear filters wx and wy, such
that the correlation of the amplitude envelopes wx.T.dot(X) and
wy.T.dot(Y) is maximized.

Reference:
----------
Dahne, S., et al., Finding brain oscillations with power dependencies
in neuroimaging data, NeuroImage (2014),
http://dx.doi.org/10.1016/j.neuroimage.2014.03.075

Notes:
------
Datasets X and Y can be either 2d numpy arrays of shape
(channels x datapoints) or 3d array of shape
(channels x datapoints x trials). For 3d arrays the average envelope
in each trial is calculated. If log == True, then the log transform is
taken before the average inside the trial

The filters are in the columns of the filter matrices Wx and Wy,
for 2d input the data can be filtered as:

np.dot(Wx.T, X)

for 3d input:

np.tensordot(Wx, X, axes=(0,0))

Input:
------
-- X numpy array - the first dataset of shape px x N (x tr), where px
                   is the number of sensors, N the number of data-
                   points, tr the number of trials
-- Y is the second dataset of shape py x N (x tr)
-- opt {'max', 'min'} - determines whether the correlation coefficient
                        should be maximized - seeks for positive
                        correlations ('max', default);
                        or minimized - seeks for anti-correlations
                        ('min')
-- num int > 0 - determine the number of filter-pairs that will be
                 derived. This depends also on the ranks of X and Y,
                 if X and Y are 2d the number of filter pairs will be:
                 min([num, rank(X), rank(Y)]). If X and/or Y are 3d the
                 array is flattened into a 2d array before calculating
                 the rank
-- log {True, False} - compute the correlation between the log-
                       transformed envelopes, if datasets come in
                       epochs, then the log is taken before averaging
                       inside the epochs, defaults to True
-- bestof int > 0 - the number of restarts for the optimization of the
                    individual filter pairs. The best filter over all
                    these restarts with random initializations is
```

pattern_from_filter(*filter*, *X*)

```
Get a an activation pattern from the filter matrix
and the input data X.

Input:
------
filter - is a p (x k) numpy array, where p is the dimensionality of the
         data and k is the number of filters (if k == 1, the array may
         be 1-dimensional)
X - is an p x N (x tr) numpy array, where p is the dimensionality of
    the data, N is the number of datapoints (per trial, and tr is the
    number of trials)
    If X is 3d, the patterns are computed by collapsing the last 2 axes

Output:
-------
pattern - is a (k x) p numpy array of activation patterns, where p is
          the dimensionality of the data and k is the number of filters
          If the input filter array is 1d so will be this array
```

6.2 Variables

Name	Description
__package__	**Value:** 'meet'

7 Module meet.eeg_viewer

```
Simple interactive EEG viewer

Submodule of the Modular EEg Toolkit - MEET for Python.

Author:
-------
Gunnar Waterstraat
gunnar[dot]waterstraat[at]charite.de
```

7.1 Variables

Name	Description
__package__	**Value:** `'meet'`

7.2 Class plotEEG

7.2.1 Methods

__init__(*self*, *signals*, *ylabels*, *t*, *t_res*=30, *title*=False)

```
Function to Plot EEG-Signals in of several channels
Def: PlotEEG(signals, ylabels, t, t_res, title = False)

Interaction:
------------
PageUp -> Go backward a big step
PageDown -> Go forward a big step
Up -> Go backward a small step
Down -> go forward a small step
i -> Zoom in (show smaller temporal window)
o -> Zoom out (show larger temporal window)
+ -> increase gain
- -> decrease gain
Pos1 -> Go to start
End -> Go to end
LeftMouseClick -> saves the x coordinate in self.clicks

Input:
------
    -signals: 2d array of input data
    -ylabels: list of channel names in same order as in
              'signals' (unit is presumed to be mikro V)
    -t: array of time values in s
    -t_res: mm per second (standard: 30 mm / s)
    -title: title string
Output:
-------
    -EEG_viewer class: plot with EEG.show()
     clicks are saved in self.clicks
```

change_gain(*self*, *new_offset*)

change_t(*self*, *new_t0*, *new_t_show*)

show(*self*)

8 Module meet.elm

```
Extreme Learning Machine Classification

Submodule of the Modular EEg Toolkit - MEET for Python.

This module implements regularized Extreme Learning Machine
Classification and Weighted Extreme Learning Machine Classification.

Classification is implemented in the ClassELM class

For faster execution of dot product the module dot_new is imported since
it avoids the need of temporary copy and calls fblas directly.
The code is available here: http://pastebin.com/raw.php?i=M8TfbURi
In future this will be available in numpy directly:
https://github.com/numpy/numpy/pull/2730

1. Extreme Learning Machine for Regression and Multiclass Classification
Guang-Bin Huang, Hongming Zhou, Xiaojian Diang, Rui Zhang
IEEE Transactions of Systems, Man and Cybernetics - Pat B: Cybernetics,
Vol. 42, No. 2. April 2012

2. Weighted extreme learning machine for imbalance learning.
Weiwei Zong, Guang-Bin Huang, Yiqiang Chen
Neurocomputing 101 (2013) 229-242

Author & Contact
----------------
Written by Gunnar Waterstraat
email: gunnar[dot]waterstraat[at]charite.de
```

8.1 Functions

accuracy(*conf_matrix*)

```
Measure of the performance of the classifier.
The Accuracy is the proportion of correctly classified items in
relation to the total number of items.

You should be aware that this is very sensitive to imbalanced data
(data with very unequal sizes of each class):
Imagine a sample with 99% of the items belonging to class 0 and 1%
of items belonging to class 1. A classifier might have an accuracy
of 99% by just assigning all items to class 0. However, the
sensitivity for class 1 is 0% in that case. It depends on your needs
if this is acceptable or not.

Input:
------
conf_matrix - shape ny x ny, where ny is the number of classes
              the rows belong to the actual, the columns to the
              predicted class: item ij is hence predicted as class
              j, while it would have belonged to class i

Output:
-------
float - the accuracy
```

G_mean(*conf_matrix*)

```
The G-mean is the geometric mean of the per-class-sensitivities. It
is much more stable to imbalance of the dataset than the global
accuray. However it depends on your needs, which measure of
performance of the classifier to use.

Input:
------
conf_matrix - shape ny x ny, where ny is the number of classes
             the rows belong to the actual, the columns to the
             predicted class: item ij is hence predicted as class j,
             while it would have belonged to class i

Output:
-------
the geometric mean of per-class sensitivities
```

Matthews(*conf_matrix*)

```
The Matthews correlation coefficient is used in machine learning as
a measure of the quality of binary (two-class) classifications. It
takes into account true and false positives and negatives and is
generally regarded as a balanced measure which can be used even if
the classes are of very different sizes. The MCC is in essence a
correlation coefficient between the observed and predicted binary
classifications; it returns a value between -1 and +1. A coefficient
of +1 represents a perfect prediction, 0 no better than random
prediction and -1 indicates total disagreement between prediction
and observation.

Source: Wikipedia (2013-09-25)

Input:
------
conf_matrix - shape 2 x 2, where 2 is the number of classes
              the rows belong to the actual, the columns to the
              predicted class: item ij is hence predicted as class j,
              while it would have belonged to class i
              AN ERROR IS THROWN IF THE SHAPE OF THE MATRIX IS NOT
              CORRECT

Output:
-------
float - the the Matthews Correlation Coefficient
```

PPV(*conf_matrix*)

```
Calculate the Positive Predictive Value

Input:
------
conf_matrix - shape 2 x 2, where 2 is the number of classes
              the rows belong to the actual, the columns to the
              predicted class: item ij is hence predicted as class j,
              while it would have belonged to class i
              AN ERROR IS THROWN IF THE SHAPE OF THE MATRIX IS NOT
              CORRECT

Output:
-------
float - the PPV
```

ssk_cv(*data*, *labels*, *folds*=3)

```
Cut data into folds with method:
shuffled, stratified, k-folds cross-validation

Input:
------
data - numpy array - shape n x p, with n items anf p features

Output:
-------
returns a list, with each list-element including the indices of one
fold
```

get_conf_matrix(*true*, *pred*)

```
Get a confusion matrix

Input:
------
true - the true labels
pred - the predicted labels

Output:
-------
conf_matrix - shape ny x ny, where ny is the number of classes
              the rows belong to the actual, the columns to the
              predicted class: item ij is hence predicted as class j,
              while it would have belonged to class i
```

8.2 Variables

Name	Description
__package__	**Value:** 'meet'

8.3 Class ClassELM

```
Class for Extreme Learning Machine Classification
-------------------------------------------

Input:
------
L - (int) - dimensionality of the feature space (defaults to 1000)
change_alg - (int) - number of samples to change from implementation
```

```
                    I to II
kernel - (str) - any of: 'sigmoid' - Sigmoid function
                         - more functions not implemented yet
-------------------------------------------

use self.cv() for cross-validation
use self.train() for training

after cross-validation or training use

self.classifiy()
```

8.3.1 Methods

__**init**__(*self*, *L*=1000, *kernel*='sigmoid')

cv(*self*, *data*, *labels*, *method*='ssk_cv', *C_array*=None, *folds*=3, *precision_func*='accuracy', *scale*=True, *weights*=True, *mem_size*=512, *verbose*=True)

```
Perform Cross-Validation of Extreme Learning Machine parameter C

Input:
------
data - numpy array - shape (n x p) with n being sample number
       and p being number of features
labels - numpy array - shape (n) with the class labels
         0,1,...,ny-2,ny-1, where ny is the number of classes
method - string - cross-validation method
                - 'ssk_cv' - shuffled stratified k-folds
                  cross-validation
C_array - numpy array - default is None - the C's which are
                        cross-validated
                      - if None from 2**(-25), 2**(-24), ...,
                                     2**(24), 2**(25)
folds - integer - default 3 - number of folds
precision_func - string or function - standard is 'accuray' -
                 Measure of performance
               - as string implemented: 'accuracy' -
                                          proportion of
                                          correctly classified
                                          to total number of
                                          samples
                                        'G_mean' - geometric
                                          mean of per-class
                                          accuracies
                                        'Matthews' - Matthews
                                          Correlation
                                          Coefficient - Only
                                          for binary
                                          classification
                                        'PPV' - Positive
                                          Predictive Value -
                                          Only for binary
                                          classification
               - if function: with confusion matrix as single
                 input and float (0,1) as single output
scale - bool (True | False) - whether data should be scaled to
                              range (-1,1)
weights - can be: - bool (True | False): - standard is True
                                           if True, data is
                                           re-weighted to a
                                           class ratio of 1.0
                                           if False, data is not
                                           re-weighted
                  - float in half-open interval [0,1)
                        - data is re-weighted such that the
                          minority / majority ratio is this
                          float
                        - minority classes are the ones having
                          less members than on average, majority
                          classes have more than average
                        - Zong et al. proposed to use the golden
                          ratio (approx. 0.618 -
                          scipy.constants.golden) as a good
```

train(*self*, *data*, *labels*, *C*, *scale*=True, *weights*=True, *mem_size*=512)

```
Train the ELM Classifier
------------------------

Input:
------
data - (numpy array) - shape n x p
                       n - number of observations
                       p - number of dimensions
       if data.ndim > 2, the array is reshaped as (n,-1)
labels - array with integer labels
C - regularization parameter
scale - bool (True | False) - standard is True
                            - switch, if the features of the
                              dataset should be scaled
                              to the interval (-1,1)
weights - can be: - bool (True | False): - standard is True
                                           if True, data is re
                                           weighted to a class
                                           ratio of 1.0 if
                                           False, data is not
                                           re-weighted
                  - float in half-open interval [0,1)
                        - data is re-weighted such that the
                          minority / majority ratio is this
                          float
                        - minority classes are the ones having
                          less members than on average, majority
                          classes have more than average
                        - Zong et al. proposed to use the golden
                        ratio (approx. 0.618 -
                        scipy.Constants.golden) as a good value
                  - numpy array with weights for each sorted
                  unique class in labels, each class weight is
                  expected to be in half-open interval [0,1)
mem_size - number - memory size of temporary array in Mb -
                    defaulte to 512

Output:
-------
No user ouput (Weights are generated and stored in the Class as
self._beta) self.istrained is set to True
```

classify(*self*, *data*, *mem_size*=512, *scale*=True)

```
Classify a dataset.

Input:
------
data - numpy array, shape N x p, where N is number of items, p
       is number of features
mem_size - number - memory size of temporary array in Mb -
           defaulte to 512
scale - bool (True | False) - if the input should be scaled
        by the network with parameters obtained during training
Output:
-------
labels - the predicted class labels: 0, 1, ..., ny-2, ny-1,
         where ny is the total number of classes
-----------------------------------------------------------------
Internally the method _run() is used
```

9 Module meet.iir

```
IIR Filtering

Submodule of the Modular EEg Toolkit - MEET for Python.

So far only butterworth filtering is implemented

Author:
-------
Gunnar Waterstraat
gunnar[dot]waterstraat[at]charite.de
```

9.1 Functions

butterworth(*data*, *fp*, *fs*, *s_rate*, *gpass*=3, *gstop*=8, *axis*=-1, *zero_phase*=True, *return_param*=False)

```
Apply a butterworth filter to the data.

fp is the passband frequency in Hz
fs is the stopband frequency in Hz

Example of Notations:
---------------------
For fs = 10, fp = 15 a high-pass filter is applied with the
   transition band between 10-15 Hz
For fs = 15, fp = 10 a low-pass filter is applied with the
   transition band between 10-15 Hz
For fs = [10, 40], fp = [20,30] a bandpass-filter with the passband
   20-30 Hz and transitions between 10-20 Hz and 30-40 Hz is applied
For fs = [20, 30], fp = [10,40] a bandstop-filter with the stopband
   20-30 Hz and transitions between 10-20 Hz and 30-40 Hz is applied

Input:
------
-- data - numpy array
-- fp - float - pass-band frequencies
-- fs - float - stop-band frequencies
-- s_rate - float - the sampling rate in Hz
-- gpass - float - maximal attenuation in the passband (in dB)
-- gstop - float - minimal attenuation in the passband (in dB)
-- axis - int -along which axis the filter is applied
-- zero_phase - bool - if zero phase filter is applied by filtering
                 in both directions
-- return_param - bool - if the order, b, and a shoud be returned

Output:
-------
-- data - numpy array - the filtered data array
if return_param:
   -- data
   -- ord - int - filter order (this must be doubled for the
                  zero_phase implementation)
   -- (b, a) - Numerator (‘b‘) and denominator (‘a‘) polynomials of
                the IIR filter.
```

9.2 Variables

Name	Description
__package__	**Value:** 'meet'

10 Module meet.spatfilt

```
Spatial Filters

Submodule of the Modular EEg Toolkit - MEET for Python.

This module implements some spatial filters such as CSP, CCA, CCAvReg,
bCSTP and QCA.

Author & Contact
----------------
Written by Gunnar Waterstraat
email: gunnar[dot]waterstraat[at]charite.de
```

10.1 Functions

CSP(*data1*, *data2*, *center*=`True`)

```
Common Spatial Pattern (CSP)

This takes the multivariate data in two conditions and finds the
spatial filters which minimize the variance in condition 2, while
simultaneously maximizing the variance in condition one.

The algorithm uses Singular Value Decomposition since this is more
stable than the Eigenvalue Decomposition of the Covariance Matrices

The filters are scaled such, that Var(cond1) + Var(cond2) = 1. The
eigenvalues give the variance in condition 1.
The filters are in the columns of the filter matrix. Patterns can be
obtained by inverting the filter matrix and are located in the rows
of the inverted filter matrix.

filter, eigval = CSP(data1, data2)
filtered_data = filter.T.dot(data1)

Input:
------
-- data1 - 2d array - data in condition 1 (variables in rows,
      observations in columns)
-- data2 - 2d array - data in condition 2 (variables in rows,
      observations in columns)
   The number of variables in data1 and data2 must be equal
-- center - bool - if data should be centered, defaults to True

Output:
-------
-- filter - where the individual filters are in the columns of the
      matrix (in order of decreasing eigenvalues)
-- eigvals - eigenvalues in decreasing order
```

CCA_data(X, Y)

```
Cannonical Correlation Analysis - by a combination of QR
decomposition and Singular Value Decomposition

The filters are scaled to give unit variance in the components and
are given in the order of decreasing canonical correlations.

Patterns can be obtained by inverting the filter matrices.

Inputs:
-------
X - shaped p1 x N - variables in rows, obervations in columns
Y - shaped p2 x N - variables in rows, obervations in columns

Outputs:
--------
a - filters for X (shape p1 x d), where d is min(rank(X), rank(Y)),
    each filter is in one column
b - filters for Y (shape p2 x d), where d is min(rank(X), rank(Y)),
    each filter is in one column
s - canonical correlations in non-increasing order, each
    corresponding to the respective colum in a and b
```

CCAvReg(*trials*)

```
Canonical Correlation Average Regression

Calculate the CCA between trials and their average across trials

The filters are scaled to give unit variance in the components and
are given in the order of decreasing canonical correlations.

Patterns can be obtained by inverting the filter matrices.

notably, all single trials can be filtered at once by:
_np.tensordot(a, trials, axes=(0,0))

the filtered average can be obtained as:
_np.dot(b.T, trials.mean(-1))

Input:
------
-- trials - 3d numpy array - 1st dim: channels
                           - 2nd dim: epoch length
                           - 3rd dim: number of epochs

Output:
-------
-- a - 2d numpy array (channel x channel) - spatial filters for
       single trials, each filter is in one column
-- b - 2d numpy array (channel x channel) - spatial filters for
       average, each filter is in one column
-- s - the canonical correlations between single trials and averages
```

bCSTP(*trials1*, *trials2*, *num_iter*=30, *s_keep*=2, *t_keep*=2, *verbose*=True)

```
bilinear Common Spatial-Temporal Patterns

In each iteration the number of kept patterns is reduced by one,
in the last 5 iterations the finally desired number of patterns is
kept.
The minimal number of iterations must therefor be > 10

Filter matrices won't be square if the data was rank deficient,
patterns can then be obtained by using the pseudo-inverse
(numpy.linalg.pinv)

Input:
------
-- trials1 - 3d numpy array - 1st dim: channels
                            - 2nd dim: epoch length
                            - 3rd dim: number of epochs
   variance in condition 1 is maximized
-- trials2 - 3d numpy array - 1st dim: channels
                            - 2nd dim: epoch length
                            - 3rd dim: number of epochs
   variance in condition 2 is minimized
-- num_iter - int - number of iterations to do, defaults to 30
                   (minimum 10)
-- s_keep - int - number of spatial patterns to keep, defaults to 2
-- t_keep - int - number of temporal patterns to keep, defaults to 2
-- verbose - bool - if number of iterations should be printed during
                   execution

Output:
-------
-- W - list of 2d arrays - spatial filter matrix for each iteration,
       the final spatial filters are in W[-1]
-- V - list of 2d arrays - temporal filter matrix for eachiteration,
                           the final temporal filters are in V[-1]
-- s_eigvals - list of 1d array - spatial eigenvalues,
               the final spatial eigenvalues are in s_eigvals[-1]
-- t_eigvals - list of 1d array - temporal eigenvalues, the final
               temporal eigenvalues are in s_eigvals[-1]
```

10.2 Variables

Name	Description
__package__	**Value:** 'meet'

11 Module meet.sphere

```
Spherical spline interpolation and CSD

Submodule of the Modular EEg Toolkit - MEET for Python.

Algorithm from  Perrin et al., Electroenceph Clin Neurophysiol 1989,
72:184-187, Corrigenda in 1990, 76: 565-566

While the implementation was done independently, the code was tested
using the sample data of the CSD Toolbox for Matlab
(http://psychophysiology.cpmc.columbia.edu/Software/CSDtoolbox/)
and results are the same. Thanks a lot th Juergen Kayer for sharing his
work there.

Example:
--------
For Plotting a scalp map:

>>> import numpy as _np
>>> coords = getStandardCoordinates(['Fp1', 'Fp2', 'C1', 'C2', 'C3',         'C4', 'P2', 'P4'])
>>> coords = projectCoordsOnSphere(coords) # be sure that these coords        are on the surface of a
>>> data = _np.random.random(coords.shape[0]) # just create a random         vector for testing
>>> X, Y, Z = potMap(coords, data) # interpolate using spherical splines

For Calculating current source densities:

>>> import numpy as _np
>>> coords = getStandardCoordinates(['Fp1', 'Fp2', 'C1', 'C2', 'C3',         'C4', 'P2', 'P4'])
>>> coords = projectCoordsOnSphere(coords) # be sure that these coords        are on the surface of a
>>> data = _np.random.random([coords.shape[0], 1000]) # just create a        random vector for testin
>>> CSD = calcCSD(coords, data) # get CSD using spherical splines -        output is in data-unit/m**

Author:
-------
Gunnar Waterstraat
gunnar[dot]waterstraat[at]charite.de
```

11.1 Functions

addHead(*ax*, *lw*=2.0, *ec*='k', ***kwargs*)

```
Add a path representing a a head consisting of
a circle with center (0,0) and r = 1 to the axis.

Input:
------
--ax - a matplotlib axes instance into which the head will be drawn
--lw - linewidth - defaults to 2.0
--ec - edgecolor - defaults to black ('k')

Output:
-------
Nothing

Notes:
------
Kwargs are matplotlib patches kwargs and are passed to
matplotlib.patches.PathPatch

Parts of the code (for drawing the circle) are from:
https://sourcegraph.com/github.com/matplotlib/matplotlib/symbols/python/lib/matplotlib/path/Path/c
```

getStandardCoordinates(*elecnames*, *fname*='standard')

```
Read (cartesian) Electrode Coordinates from tab-seperated text file

The standard is plotting_1005.txt obtained from:
http://robertoostenveld.nl/electrode/plotting_1005.txt (however 1st
and 2nd column were exchanged to get x,y,z order)

Thanks to Robert Oostenveld for sharing these files!!!

Input:
------
-- elecnames - iterable - list of Electrode names
-- fname - str - filename from which positions should be read
               - this is a tab delimeed file with the UPPERCASE
                 electrode names in first column
                 x,y,z in subsequent columns
                 T7, T8, Oz, Fpz and Cz must be included!
Output:
-------
-- coords - 2D array containing the cartesian coordinates in rows
                - 1st column: x
                - 2nd column: y
                - 3rd column: z

Example:
--------
>>> getStandardCoordinates(['fc3', 'FC1', 'xx']) # 'xx' is not a           valid electrode
array([[-0.6638    ,  0.3610691 ,  0.6545    ],
       [-0.3581    ,  0.37682936,  0.8532    ],
       [        nan,         nan,         nan]])
```

getChannelNames(*fname*)

```
Read Names of Electrodes from text file

Input:
------
-- fname - str - tab-delimeted textfile containing electrode number
                 and electrode-names
                 example:
                 1  C1
                 2  C3
output:
-------
-- elecnames -  List of electrode names in ascending order as
                determined by the electrode numbers in fname

Example:
--------
>>> getChannelNames(_path.join(_path.join(_packdir, 'test_data'),              'elecnames.txt'))
['C1', 'C3']
```

projectCoordsOnSphere(*coords*)

```
The input coordinates are projected onto a sphere with center
(0,0,0) and radius 1.
For the input coordinates it is believed that they lie on a
sphere with center (0,0,0) and radius r = x**2 + y**2 + Z**2.
Subsequently this radius is scaled th 1, preserving altitude
and azimuth of the original coordinates.

Input:
------
--coords - 3D array containing the points in rows
           - 1st column: x
           - 2nd column: y
           - 3rd column: z
Output:
-------
-- out_coords - 3D array containing the coordinates in rows

Examples:
--------
>>> coords = _np.array([[0,0,1], [0,1,0], [1,1,1]]) # the last one            is not on a spheri
>>> projectCoordsOnSphere(coords)
array([[ 0.        ,  0.        ,  1.        ],
       [ 0.        ,  1.        ,  0.        ],
       [ 0.57735027,  0.57735027,  0.57735027]])
```

projectCircleOnSphere(*coords*, *projection*='stereographic')

```
Project 2d coordinates inside a unit circle on a sphere with
center (0,0) and radius 1.

Stereographic projection:
Where the given circle is believed to be the on the plane
crossing the equator (z=0) and the perspective point is the
southpole (0,0,-1). Points inside the circle are projected
onto the northern hemisphere, points outside the circle on
the souther hemisphere.

Orthographic projection:
All points have to lie inside the circle and are projected
on the northern hemisphere by keeping the xy coordinates and
adding a z coordinate to result in a radius of 1.
Poinst outside the circle result in nans.

Input:
------
-- coords - 2D array containing the coordinates in rows
          - 1st column: x
          - 2nd column: y
-- projection - str - 'stereographic' (standard) or 'orthographic'
                      (explanations see above)
Output:
------
-- sphere_coords - masked 3D array containing the coordinates in
                   rows
                 - 1st column: x
                 - 2nd column: y
                 - 3rd column: z
 (all coordinates outside a unit sphere are masked)
```

projectSphereOnCircle(*sphere_coords*, *projection*='stereographic')

```
Project coordinates that lie on the surface of a unit sphere
with center (0,0,0) and radius 1 into a unit circle

If projection == 'stereographic':
The vertex of the sphere is (0,0,1). The projection is done from
the "southpole" (0,0,-1) onto the plane with z=0.
The northern hemisphere is hereby projected into the unit circle,
the souther hemisphere outside the unit circle.

If projection == 'orthographic':
Only the northern hemisphere is mapped onto a plane through the
equator by keeping the xy-coordinates. For coordinates on the
southern hemisphere nan is returned.

Input:
------
-- sphere_coords - 2D array containing the coordinates in rows
          - 1st column: x
          - 2nd column: y
          - 3rd column: z
-- projection - str - 'stereographic' (standard) or 'orthographic'
                      (explanations see above)
Output:
-------
-- coords - 2D array containing the coordinates in rows
                 - 1st column: x
                 - 2nd column: y
```

potMap(*RealCoords*, *data*, *diameter_samples*=200, *n*=(7, 7), *m*=4, *smooth*=0, *projection*='stereographic')

```
Get a scalp map of potentials interpolated on a sphere.

If projection == 'stereographic':
The vertex of the sphere is (0,0,1). The projection is done from
the "southpole" (0,0,-1) onto the plane with z=0.
The northern hemisphere is hereby projected into the unit circle,
the souther hemisphere outside the unit circle.

If projection == 'orthographic':
Only the northern hemisphere is mapped onto a plane through the
equator by keeping the xy-coordinates. For coordinates on the
southern hemisphere nan is returned.

Input:
------
-- RealCoords - array of shape channels x 3 - (x,y,z) cartesian
                coordinates of physical electrodes on a sphere with
                center (0,0,0) and radius 1
-- data - array of shape channels x datapoints - containing values
          at electrodes in same order as in 'RealCoords'
-- diameter_samples - int - number of points along the diameter of
                      the scalp to interpolate
-- n - tuple of ints - how many terms of the Legendre-Polynomial
       should be calculated, defaults to (7,7)
-- m - int - order of the spherical spline interpolation, defaults
       to 4
-- smooth - float - amount of smoothing, defaults to 0
-- projection - str - 'stereographic' (standard) or 'orthographic'
                      (explanations see above)

Output:
-------
-- X, Y, Z - grid containing the X and Y coordinates and the
             interpolated values where everything outside the unit
             circle is masked.
```

csdMap(*RealCoords*, *data*, *diameter_samples*=200, *n*=(7, 20), *m*=4, *smooth*=1e-05, *projection*='stereographic')

```
Get a scalp map of CSDs calculated from spherical splines.

If projection == 'stereographic':
The vertex of the sphere is (0,0,1). The projection is done from
the "southpole" (0,0,-1) onto the plane with z=0.
The northern hemisphere is hereby projected into the unit circle,
the souther hemisphere outside the unit circle.

If projection == 'orthographic':
Only the northern hemisphere is mapped onto a plane through the
equator by keeping the xy-coordinates. For coordinates on the
southern hemisphere nan is returned.

Input:
------
-- RealCoords - array of shape channels x 3 - (x,y,z) cartesian
                coordinates physical electrodes on a sphere with
                center (0,0,0) and radius 1
-- data - array of shape channels x datapoints - containing values
          at electrodes in same order as in 'RealCoords'
-- diameter_samples - int - number of points along the diameter of
                      the scalp to interpolate
-- n - tuple of ints - how many terms of the Legendre-Polynomial
       should be calculated, defaults to (7,20)
-- m - int - order of the spherical spline interpolation,
       defaults to 4
-- smooth - float - amount of smoothing, defaults to 1E-5
-- projection - str - 'stereographic' (standard) or 'orthographic'
                      (explanations see above)

Output:
-------
-- X, Y, Z - grid containing the X and Y coordinates and the
             interpolated values where everything outside the unit
             circle is masked. the unit of z is data_unit / m**2
```

calcCSD(*Coords*, *data*, *n*=(7, 20), *m*=4, *smooth*=1e-05, *buffersize*=64)

```
Calculate current source densities at the same electrode posisionts
as in the original data set

Input:
------
-- Coords - 2d array - channels x (x,y,z) cartesian coordinates of
            physical electrodes
-- data - array containing values at electrodes in same order as in
          'Coords'
    -- 1st dim: channels
    -- 2nd dim: datapoints
-- n - tuple of ints - How many terms of the Legendre-Polynomial
       should be calculated, defaults to (7, 20) - 7 for the first
       and 20 for the 2nd polynomial
-- m - order of the spherical spline interpolation, defaults to 4
-- smooth - float - amount of smoothing, defaults to 1E-5
-- buffersize - float - determine the size of a buffer in MB to do
                the calculations

Output:
-------
-- pot - array containing CSD-values in data-unit / m**2

Example:
--------
>>> Coords = getStandardCoordinates(['C1', 'C3'])
>>> Coords = projectCoordsOnSphere(Coords) # be sure that the             points are on a sphere
>>> data = _np.arange(6, dtype=float).reshape(2,-1)
>>> calcCSD(Coords, data)
array([[-3.5474569 , -3.5474569 , -3.5474569 ],
       [ 3.60884286,  3.60884286,  3.60884286]])
```

smoothSP(*Coords*, *data*, *n*=(7, 7), *m*=4, *smooth*=1e-05, *buffersize*=64)

```
Calculate spatially smoothed potential values at the same electrode
positions as in the original data set

Input:
------
-- Coords - 2d array - channels x (x,y,z) cartesian coordinates of
            physical electrodes
-- data - array containing values at electrodes in same order as in
          'Coords'
    -- 1st dim: channels
    -- 2nd dim: datapoints
-- n - tuple of ints - How many terms of the Legendre-Polynomial
       should be calculated, defaults to (7, 7) - 7 for the first
       and 7 for the 2nd polynomial
-- m - order of the spherical spline interpolation, defaults to 4
-- smooth - float - amount of smoothing, defaults to 1E-5
-- buffersize - float - determine the size of a buffer in MB to do
                the calculations

Output:
-------
-- pot - array containing the spatially smoothed potentials

Example:
--------
>>> Coords = getStandardCoordinates(['C1', 'C3'])
>>> Coords = projectCoordsOnSphere(Coords) # be sure that the            points are on a sphere
>>> data = _np.arange(6, dtype=float).reshape(2,-1)
>>> smoothSP(Coords, data)
array([[ 0.01220571,  1.01220571,  2.01220571],
       [ 2.98778429,  3.98778429,  4.98778429]])
```

11.2 Variables

Name	Description
__package__	**Value:** 'meet'

12 Module meet.tf

```
S transform (time-frequency transformation)

Submodule of the Modular EEg Toolkit - MEET for Python.

the 'standard' S transform:
---------------------------
Stockwell, Robert Glenn, Lalu Mansinha, and R. P. Lowe. "Localization of
the complex spectrum: the S transform." Signal Processing,
IEEE Transactions on 44.4 (1996): 998-1001.

as well as

the fast dyadic S transform:
----------------------------------------
Brown, Robert A., M. Louis Lauzon, and Richard Frayne. "A general
description of linear time-frequency transforms and formulation of a
fast, invertible transform that samples the continuous S-transform
spectrum nonredundantly." Signal Processing, IEEE Transactions on 58.1
(2010): 281-290.

Author:
-------
Gunnar Waterstraat
gunnar[dot]waterstraat[at]charite.de

Example for a standard S transform:
-----------------------------------
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> data = np.random.random(1000) # generate a random 1000 dp signal
>>> tf_coords, S = gft(data, sampling='full')
>>> S = S.reshape(-1, data.shape[0]) # for the full transform no         interpolation is needed -> ju
>>> plt.imshow(_np.abs(S), aspect='equal', origin='lower') #plot the         result


Example for a dyadic S transform:
---------------------------------
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> data = np.random.random(1000) # generate a random 1000 dp signal
>>> tf_coords, S = gft(data, sampling='dyadic')
>>> t, f, S_interp = interpolate_gft(tf_coords, S, (data.shape[0]//2,         data.shape[0]), data.shap
>>> plt.imshow(_np.abs(S_interp), aspect='equal', origin='lower',         extent=[t[0], t[-1], f[0], f
```

12.1 Functions

gft(*sig*, *window*='gaussian', *axis*=-1, *sampling*='full', *full*=False, *hanning*=True)

```
Calculate the discrete general fourier family transform

Inputs:
-------
-- sig - signal of interest - 1 or 2 dimensional
-- window - window function, standard is gaussian
-- axis - axis along which transform should be performed
-- sampling - 'full'|'dyadic' - should the S-domain be sampled
              completely, i.e. redundant or dyadic
-- full - True|False - return also negative frequencies, standard
          is False
-- hanning - True|False - Apply a hanning window to 5% first and
             last datapoints

Outputs:
--------
-- Coords - (frequency, time) Coordinates of each point in S
-- S - complex S transform array
   if sampling is dyadic interpolate_gft should be used to
   interpolate the result onto a regular grid
   if sampling is full, this can be easily done by reshaping each S
   transform to the shape (-1, sig.shape[axis])
```

interpolate_gft(*Coords*, *S*, *IM_shape*, *data_len*, *kindf*='nearest', *kindt*='nearest')

```
Interpolate the result of a gft-transform - standard is nearest
neighbor interpolation

Input:
------
-- Coords, Coords as output by the gft command
-- S - A gft result list
-- IM_shape - requested shape of the interpolated matrix
       1st axis frequency
       2nd axis time
-- data_len - length of inital data
-- kindf - interpolation method along frequency axis - any of
           ['nearest', 'linear']
-- kindt - interpolation method along time axis - any of
           ['nearest', 'linear']

Output:
------
-- wanted times - time point array
-- wanted freqs - frequency array
-- IM - interpolated matrix
```

12.2 Variables

Name	Description
__package__	**Value:** 'meet'

Index