dyconnmap Documentation

Marimpis Avraam

Mar 14, 2022

CONTENTS

Contents:

CHAPTER

ONE

DYCONNMAP PACKAGE

1.1 Subpackages

1.1.1 dyconnmap.chronnectomics package

1.1.1.1 Submodules

1.1.1.2 dyconnmap.chronnectomics.dwell_time module

Dwell Time

Dwell time measures the time (when used in the context of functional connectivity microstates) a which a state is active consecutive temporal segments (**Dimitriadis2019_**).

dyconnmap.chronnectomics.dwell_time.**dwell_time**(*x*)

Dwell Time

Compute the dwell time for the given symbolic, 1d time series.

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns

- **dwell** (*dictionary*) – KVP, where K=symbol id and V=array of dwell time.
- **mean** (*dictionary*) – KVP, where K=symbol id and V=mean dwell time.
- **std** (*dictionary*) – KVP, where K=symbol id and V=std dwell time.

1.1.1.3 dyconnmap.chronnectomics.flexibility_index module

Flexibility Index

In the context of graph clustering it was defined in (*Basset2011*), flexbility is the frequency of a nodea change module allegiance; the transition of brain states between consecutive temporal segments. The higher the number of changes, the larger the FI will be.

$$FI = \frac{\text{number of transitions}}{\text{total symbols - 1}}$$

dyconnmap.chronnectomics.flexibility_index.**flexibility_index**(*x*)
Flexibility Index

Compute the flexibility index for the given symbolic, 1d time series.

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns **fi** – The flexibility index.

Return type float

1.1.1.4 dyconnmap.chronnectomics.occupancy_time module

Occupancy Time

The fraction of number of distinct symbols occuring in the symbolic time series (**Dimitriadis2019_**).

dyconnmap.chronnectomics.occupancy_time.**occupancy_time**(*x*)
Occupancy Time

Compute the occupancy time for the given symbolic, 1d time series.

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns **ot** – KVP, where K=symbol id and V=occupancy time.

Return type dictionary

1.1.1.5 Module contents

dyconnmap.chronnectomics.**dwell_time**(*x*)
Dwell Time

Compute the dwell time for the given symbolic, 1d time series.

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns

- **dwell** (*dictionary*) – KVP, where K=symbol id and V=array of dwell time.
- **mean** (*dictionary*) – KVP, where K=symbol id and V=mean dwell time.
- **std** (*dictionary*) – KVP, where K=symbol id and V=std dwell time.

dyconnmap.chronnectomics.**flexibility_index**(*x*)
Flexibility Index

Compute the flexibility index for the given symbolic, 1d time series.

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns **fi** – The flexibility index.

Return type float

dyconnmap.chronnectomics.**occupancy_time**(*x*)
Occupancy Time

Compute the occupancy time for the given symbolic, 1d time series.

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns **ot** – KVP, where K=symbol id and V=occupancy time.

Return type dictionary

1.1.2 dyconnmap.cluster package

1.1.2.1 Submodules

1.1.2.2 dyconnmap.cluster.cluster module

Base class for clustring algorithms

class dyconnmap.cluster.cluster.**BaseCluster**
Bases: object

Base class for clustering alorithms.

encode(*data*, *metric='euclidean'*, *sort=True*)
Employ a nearest-neighbor rule to encode the given data using the codebook.

Parameters

- **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.
- **metric** (*string or None*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine

  If *None* is passed, the matric used for learning the data will be used.
- **sort** (*boolean*) – Whether or not to sort the symbols using MDS first. Default *True*

Returns

- **encoded_data** (*real array-like, shape(n_samples, n_features)*) – data, as represented by the prototypes in codebook.
- **ts_symbols** (*list, shape(n_samples, 1)*) – A discrete symbolic time series

1.1.2.3 dyconnmap.cluster.gng module

Growing NeuralGas

Growing Neural Gas (*GNG*) [Fritzke1995] is a dynamic neural network (as in adaptive) that learns topologies. Compared to Neural Gas, GNG provides the functionality of adding or purging the constructed graph of nodes and edges when certain criterion are met.

To do so, each node on the network stores a number of secondary information and statistics, such as, its learning vector, a local error, etc. Edge edge is assigned with a counter related to its age; so as older edges are pruned.

The convergence of the algorithm depends either on the maximum number of nodes of the graph, or an upper limit of elapsed iterations.

Briefly, the algorithm works as following:

1. Create two nodes, with weights drawn randomly from the original distibution; connect these two nodes. Set the edge's age to zero.
2. Draw randomly a sample ($\overrightarrow{x}$) from the distibution.
3. For each node (n) in the graph with associated weights $\overrightarrow{w}$, we compute the euclidean distance from $\overrightarrow{x}$: $||\overrightarrow{n}_w - \overrightarrow{x}||^2$. Next, we find the two nodes closest $\overrightarrow{x}$ with distances d_s and d_t.
4. The best matching unit (s) adjusts:

a. its weights: $\overrightarrow{s}_w \leftarrow \overrightarrow{s}_w + [e_w * (\overrightarrow{x} - \overrightarrow{s}_w)]$.

b. its local error: $s_{error} \leftarrow s_{error} + d_s$.

5. Next, the nodes (N) adjacent to s:

a. update their weights: $\overrightarrow{N}_w \leftarrow \overrightarrow{N}_w + [e_n * (\overrightarrow{x} - \overrightarrow{N}_w)]$.

b. increase the age of the connecting edges by one.

6. If the best and second mathing units (s and t) are connected, we reset the age of the connecting edge. Otherwise, we connect them.
7. Regarding the pruning of the network. First we remove the edges with older than a_{max}. In the seond pass, we remove any disconnected nodes.
8. We check the iteration ($iter$), whether is a multiple of λ and if the maximum number of iteration has been reached; then we add a new node (q) in the graph:

a. Let u denote the node with the highest error on the graph, and v its neighbor with the highest error.

b. we disconnect u and v

c. q is added between u and v: $\overrightarrow{q}_w \leftarrow \frac{\overrightarrow{u}_w + \overrightarrow{v}_w}{2}$.

d. connect q to u, and q to v

e. reduce the local errors of both u and v: $u_{error} \leftarrow \alpha * u_{error}$ and $v_{error} \leftarrow \alpha * v_{error}$

f. define the local error q: $q_{error} \leftarrow u_{error}$

8. Adjust the error of each node (n) on the graph: $n_{error} \leftarrow n_{error} - \beta * n_{error}$
9. Finally, increate the iteration ($iter$) and if any of the criterion is not satisfied, repeat from step #2.

class dyconnmap.cluster.gng.**GrowingNeuralGas**(*n_max_protos=30*, *l=300*, *a_max=88*, *a=0.5*, *b=0.0005*, *iterations=10000*, *lrate=None*, *n_jobs=1*, *rng=None*)

Bases: *dyconnmap.cluster.cluster.BaseCluster*

Growing Neural Gas

Parameters

- **n_max_protos** (*int*) – Maximum number of nodes.
- **l** (*int*) – Every iteration is checked if it is a multiple of this value.
- **a_max** (*int*) – Maximum age of edges.
- **a** (*float*) – Weights the local error of the nodes when adding a new node.
- **b** (*float*) – Weights the local error of all the nodes on the graph.
- **iterations** (*int*) – Total number of iterations.
- **lrate** (*list of length 2*) – The learning rates of the best matching unit and its neighbors.
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn)).
- **rng** (*object or None*) – An object of type numpy.random.RandomState.

protos

The prototypical vectors

Type array-like, shape(n_protos, n_features)

Notes

fit(*data*)

Learn data, and construct a vector codebook.

Parameters **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.

Returns **self** – The instance itself

Return type object

1.1.2.4 dyconnmap.cluster.mng module

Merge Neural Gas

Merge Neural Gas (MNG) [Strickert2003] is similar to the original Neural Gas algorithm, but each node, has an additional context vector (c) associated; and the best matching unit is determined by a linear combination of both the weight and context vector (thus the merge), from the previous iteration.

Similar to Neural Gas, N nodes have their weights (w) and context vectors (c) randomly initialized. When the network is presented with a new input sequence v, the distance and the corresponding rank of each node n is estimated.

1. The distance is computed by: $d_i(n) = (1 - \alpha) * ||v - w_i||^2 + ||c(n) - c_i||^2$ and the context by: $c(n) = (1 - \beta) * w_{I_n - 1} + \beta * c_{I_n - 1}$.

- α is the balancing factor between w and c
- β is the merging degree between two subsequent iterations
- $I_n - 1$ denotes the previous iteration

2. Each node is adapted with:
   a. $\Delta w_i = e_w(k) * h_{\lambda(k)}(r(d_i, d)) * (v - w_i)$ and its context by
   b. $\Delta c_i = e_w(k) * h_{\lambda(k)}(r(d_i, d)) * (c(n) - c_i)$
   - $r(d_i, d)$ denotes the rank of the i-th node

As with Neural Gas, $h_{\lambda(k)}(r(d_i, d)) = exp(\frac{-r(d_i,d)}{\lambda(k)})$ represents the neighborhood ranking function. $\lambda(k)$ denotes the influence of the neighbood: $\lambda_0(\frac{\lambda_T}{\lambda_0})^{(\frac{t}{T_{max}})}$.

Notes

---

class dyconnmap.cluster.mng.**MergeNeuralGas**(*n_protos=10*, *iterations=1024*, *merge_coeffs=None*, *epsilon=None*, *lrate=None*, *n_jobs=1*, *rng=None*)

Bases: *dyconnmap.cluster.cluster.BaseCluster*

Merge Neural Gas

Parameters

- **n_protos** (*int*) – The number of prototypes
- **iterations** (*int*) – The maximum iterations
- **merge_coeffs** (*list of length 2*) – The merging coefficients
- **epsilon** (*list of length 2*) – The initial and final training rates.
- **lrate** (*list of length 2*) – The initial and final rearning rates.
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn)).
- **rng** (*object*) – An object of type numpy.random.RandomState.

protos
The prototypical vectors

Type array-like, shape(n_protos, n_features)

distortion
The normalized distortion error

Type float

encode(*data*, *metric='euclidean'*)
Employ a nearest-neighbor rule to encode the given data using the codebook.

Parameters

- **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.
- **metric** (*string or None*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

- euclidean
- cityblock
- l1
- cosine

If *None* is passed, the matric used for learning the data will be used.

- **sort** (*boolean*) – Whether or not to sort the symbols using MDS first. Default *True*

Returns

- **encoded_data** (*real array-like, shape(n_samples, n_features)*) – `data`, as represented by the prototypes in codebook.
- **ts_symbols** (*list, shape(n_samples, 1)*) – A discrete symbolic time series

fit(*data*)

Parameters **data** –

Returns

1.1.2.5 dyconnmap.cluster.ng module

NeuralGas

Inspired by Self Organizing Maps (SOMs), Neural Gas (*NG*), an unsupervised adaptive algorithm coined by [Martinetz1991]. Neural Gas does not assume a preconstructed lattice thus the adaptation cannot be based on the distances between the neighbor neurons (like in SOMs) because be definition there are no neighbors.

The adaptation-convergence is driven by a stochastic gradient function with a soft-max adaptation rule that minimizes the average distortion error.

First, we construct a number of neurons ($N_{\vec{w}}$) with random weights ($\vec{w}$). Then we train the model by feeding it feature vectors sequentially drawn from the distribution $P(t)$. When a new feature vector is presented to the model, we sort all neurons' weights ($N_{\vec{w})}$ based on their Euclidean distance from $\vec{x}$. Then, the adaptation if done by:

$$\vec{w} \leftarrow \vec{w} + [N_{\vec{w}} \cdot e(t)) \cdot h(k)) \cdot (\vec{x} - \vec{w})], \forall \vec{w} \in N_{\vec{w}}$$

where,

$$h(t) = exp\frac{-k}{\sigma^2}(t)$$
$$\sigma^2 = \lambda_i(\frac{\lambda_T}{\lambda_0})^{(\frac{t}{T_{max}})}$$
$$e(t) = e_i(\frac{e_T}{e_0})^{(\frac{t}{T_{max}})}$$

The parameter λ, governs the initial and final learning rate, while the parameter e the training respectively.

After the presentation of a feature vector, increase the itaration counter t and repeat until all desired criteria are met, or $t = T_{max}$.

With these prototypes, we can represent all the input feature vectors $\vec{x}$ using a Nearest Neighbor rule. The quality of this encoding can measured by the normalized distortion error:

$$\frac{\sum_{t=1}^{T} ||X(t) - X^*(t))||^2}{\sum_{t=1}^{T} \left|\left|X(t) - \overline{X})\right|\right|^2}$$

where

$$\overline{X}` = \frac{1}{T}\sum_{t=1}^{T} X(t)$$

T is the number of reference prototypes; in X the input patterns are stored; X^* contains the approximated patterns as produced by the Nearest Neighbor rule.

Notes

For faster convergence, we can also draw random weights from the given probability distribution $P(t)$

---

class dyconnmap.cluster.ng.**NeuralGas**(*n_protos=10*, *iterations=1024*, *epsilon=None*, *lrate=None*, *n_jobs=1*, *metric='euclidean'*, *rng=None*)

Bases: *dyconnmap.cluster.cluster.BaseCluster*

Neural Gas

Parameters

- **n_protos** (*int*) – The number of prototypes
- **iterations** (*int*) – The maximum iterations
- **epsilon** (*list of length 2*) – The initial and final training rates
- **lrate** (*list of length 2*) – The initial and final rearning rates
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn))
- **metric** (*string*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine
- **rng** (*object or None*) – An object of type numpy.random.RandomState

protos

The prototypical vectors

Type array-like, shape(n_protos, n_features)

distortion

The normalized distortion error

Type float

Notes

Slightly based on *http://webloria.loria.fr/~rougier/downloads/ng.py*

fit(*data*)
Learn data, and construct a vector codebook.

Parameters **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.

Returns **self** – The instance itself

Return type object

1.1.2.6 dyconnmap.cluster.rng module

Relational Neural Gas

Relational Neural Gas (RNG) [Hammer2007] is a variant of Neural Gas, that allows clustering and mining data from a pairwise similarity or dissimilarity matrix.

---

class dyconnmap.cluster.rng.**RelationalNeuralGas**(*n_protos=10*, *iterations=100*, *lrate=None*, *metric='euclidean'*, *rng=None*)

Bases: *dyconnmap.cluster.cluster.BaseCluster*

Relational Neural Gas

Parameters

- **n_protos** (*int*) – The number of prototypes
- **iterations** (*int*) – The maximum iterations
- **lrate** (*list of length 2*) – The initial and final rearning rates
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn))
- **metric** (*string*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine
- **rng** (*object or None*) – An object of type numpy.random.RandomState

protos
The prototypical vectors

Type array-like, shape(n_protos, n_features)

fit(*data*)
Fit

Parameters

- **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.
- **metric** (*string or None*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine

  If *None* is passed, the matric used for learning the data will be used

1.1.2.7 dyconnmap.cluster.som module

Self Organizing Map

T is the number of reference prototypes; in X the input patterns are stored; X^* contains the approximated patterns as produced by the Nearest Neighbor rule.

Notes

For faster convergence, we can also draw random weights from the given probability distribution $P(t)$

---

class dyconnmap.cluster.som.**SOM**(*grid=(10, 10)*, *iterations=1024*, *lrate=0.1*, *n_jobs=1*, *rng=None*)
Bases: *dyconnmap.cluster.cluster.BaseCluster*

Self Organizing Map

Parameters

- **grid** (*list of length 2*) – The X and Y sizes of the grid
- **iterations** (*int*) – The maximum iterations
- **lrate** (*float*) – The initial rearning rate
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn))
- **rng** (*object or None*) – An object of type numpy.random.RandomState

protos
The prototypical vectors

Type array-like, shape(n_protos, n_features)

classmethod findBMU(*x*, *y*)

fit(*data*)

1.1.2.8 dyconnmap.cluster.umatrix module

UMatrix Visualization

dyconnmap.cluster.umatrix.**umatrix**(*M*)

1.1.2.9 dyconnmap.cluster.validity module

---

dyconnmap.cluster.validity.**davies_bouldin**(*data*, *labels*)
Davies-Bouldin Index

Parameters

- **data** (*array-like, shape(n_ts, n_samples)*) – Input time series
- **labels** (*array-like, shape(n_ts)*) – Cluster assignements (labels) per time serie.

Returns **index**

Return type float

dyconnmap.cluster.validity.**ray_turi**(*data*, *labels*)
Ray-Turi Index

Parameters

- **data** (*array-like, shape(n_ts, n_samples)*) – Input time series
- **labels** (*array-like, shape(n_ts)*) – Cluster assignements (labels) per time serie.

Returns **index**

Return type float

1.1.2.10 Module contents

class dyconnmap.cluster.**GrowingNeuralGas**(*n_max_protos=30*, *l=300*, *a_max=88*, *a=0.5*, *b=0.0005*, *iterations=10000*, *lrate=None*, *n_jobs=1*, *rng=None*)
Bases: *dyconnmap.cluster.cluster.BaseCluster*

Growing Neural Gas

Parameters

- **n_max_protos** (*int*) – Maximum number of nodes.
- **l** (*int*) – Every iteration is checked if it is a multiple of this value.
- **a_max** (*int*) – Maximum age of edges.
- **a** (*float*) – Weights the local error of the nodes when adding a new node.
- **b** (*float*) – Weights the local error of all the nodes on the graph.
- **iterations** (*int*) – Total number of iterations.

- **lrate** (*list of length 2*) – The learning rates of the best matching unit and its neighbors.
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn)).
- **rng** (*object or None*) – An object of type numpy.random.RandomState.

protos
The prototypical vectors

Type array-like, shape(n_protos, n_features)

Notes

fit(*data*)
Learn data, and construct a vector codebook.

Parameters **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.

Returns **self** – The instance itself

Return type object

class dyconnmap.cluster.**MergeNeuralGas**(*n_protos=10*, *iterations=1024*, *merge_coeffs=None*, *epsilon=None*, *lrate=None*, *n_jobs=1*, *rng=None*)
Bases: *dyconnmap.cluster.cluster.BaseCluster*

Merge Neural Gas

Parameters

- **n_protos** (*int*) – The number of prototypes
- **iterations** (*int*) – The maximum iterations
- **merge_coeffs** (*list of length 2*) – The merging coefficients
- **epsilon** (*list of length 2*) – The initial and final training rates.
- **lrate** (*list of length 2*) – The initial and final rearning rates.
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn)).
- **rng** (*object*) – An object of type numpy.random.RandomState.

protos
The prototypical vectors

Type array-like, shape(n_protos, n_features)

distortion
The normalized distortion error

Type float

encode(*data*, *metric='euclidean'*)
Employ a nearest-neighbor rule to encode the given `data` using the codebook.

Parameters

- **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.

- **metric** (*string or None*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine

  If *None* is passed, the matric used for learning the data will be used.

- **sort** (*boolean*) – Whether or not to sort the symbols using MDS first. Default *True*

Returns

- **encoded_data** (*real array-like, shape(n_samples, n_features)*) – `data`, as represented by the prototypes in codebook.
- **ts_symbols** (*list, shape(n_samples, 1)*) – A discrete symbolic time series

fit(*data*)

Parameters **data** –

Returns

class dyconnmap.cluster.**NeuralGas**(*n_protos=10*, *iterations=1024*, *epsilon=None*, *lrate=None*, *n_jobs=1*, *metric='euclidean'*, *rng=None*)

Bases: *dyconnmap.cluster.cluster.BaseCluster*

Neural Gas

Parameters

- **n_protos** (*int*) – The number of prototypes
- **iterations** (*int*) – The maximum iterations
- **epsilon** (*list of length 2*) – The initial and final training rates
- **lrate** (*list of length 2*) – The initial and final rearning rates
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn))
- **metric** (*string*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine

- **rng** (*object or None*) – An object of type numpy.random.RandomState

protos

The prototypical vectors

Type array-like, shape(n_protos, n_features)

distortion
The normalized distortion error

Type float

Notes

Slightly based on *http://webloria.loria.fr/~rougier/downloads/ng.py*

fit(*data*)
Learn data, and construct a vector codebook.

Parameters **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.

Returns **self** – The instance itself

Return type object

class dyconnmap.cluster.**RelationalNeuralGas**(*n_protos=10*, *iterations=100*, *lrate=None*, *metric='euclidean'*, *rng=None*)
Bases: *dyconnmap.cluster.cluster.BaseCluster*

Relational Neural Gas

Parameters

- **n_protos** (*int*) – The number of prototypes
- **iterations** (*int*) – The maximum iterations
- **lrate** (*list of length 2*) – The initial and final rearning rates
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn))
- **metric** (*string*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine
- **rng** (*object or None*) – An object of type numpy.random.RandomState

protos
The prototypical vectors

Type array-like, shape(n_protos, n_features)

fit(*data*)
Fit

Parameters

- **data** (*real array-like, shape(n_samples, n_features)*) – Data matrix, each row represents a sample.

- **metric** (*string or None*) – One of the following valid options as defined for function http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html.

  Valid options include:

  - euclidean
  - cityblock
  - l1
  - cosine

  If *None* is passed, the matric used for learning the data will be used

class dyconnmap.cluster.**SOM**(*grid=(10, 10)*, *iterations=1024*, *lrate=0.1*, *n_jobs=1*, *rng=None*)

Bases: *dyconnmap.cluster.cluster.BaseCluster*

Self Organizing Map

Parameters

- **grid** (*list of length 2*) – The X and Y sizes of the grid
- **iterations** (*int*) – The maximum iterations
- **lrate** (*float*) – The initial rearning rate
- **n_jobs** (*int*) – Number of parallel jobs (will be passed to scikit-learn))
- **rng** (*object or None*) – An object of type numpy.random.RandomState

protos

The prototypical vectors

Type array-like, shape(n_protos, n_features)

classmethod findBMU(*x*, *y*)

fit(*data*)

dyconnmap.cluster.**davies_bouldin**(*data*, *labels*)

Davies-Bouldin Index

Parameters

- **data** (*array-like, shape(n_ts, n_samples)*) – Input time series
- **labels** (*array-like, shape(n_ts)*) – Cluster assignements (labels) per time serie.

Returns **index**

Return type float

dyconnmap.cluster.**ray_turi**(*data*, *labels*)

Ray-Turi Index

Parameters

- **data** (*array-like, shape(n_ts, n_samples)*) – Input time series
- **labels** (*array-like, shape(n_ts)*) – Cluster assignements (labels) per time serie.

Returns **index**

Return type float

dyconnmap.cluster.**umatrix**(*M*)

1.1.3 dyconnmap.fc package

1.1.3.1 Submodules

1.1.3.2 dyconnmap.fc.aec module

Amplitude Envelope Correlation

Amplitude Envelope Correlation (*AEC*), estimates the coupling (without phase coherence and even among different frequencies [*Bruns2000*]) by computing the correlation coefficient of a signal's amplitude envelope.

$$r_{AEC} = \text{corr}(\alpha_{lo}, \alpha_{hi})$$

Where α denotes the Instantaneous Amplitude of a given signal, filtered in a specfic frequency band (lo or hi).

---

dyconnmap.fc.aec.**aec**(*data*, *fb_lo*, *fb_hi*, *fs*)

Amplitude Envelope Correlation

Estimate the Amplitude-Envelope Correlation for the given `data`.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

1.1.3.3 dyconnmap.fc.biplv module

Bi-Phase Locking Value

dyconnmap.fc.biplv.**biplv**(*data*, *fb_lo*, *fb_hi*, *fs*, *pairs=None*)

Bi-Phase Locking Value

Estimate the Bi-Phase Locking Value for the given `data`, between the :attr:`pairs (if given) of channels

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.

- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.

1.1.3.4 dyconnmap.fc.coherence module

Coherence

Coherence (*Coh*) is one of the most commonly utilized connectivity estimators; it is a measurement of the linear relationship of two signals at a specific frequency [*Nolte2004*].

Given two time series x and y, coherece is given by:

$$coh_{xy}^2(f) = \frac{|G_{xy}(f)^2|}{G_{xx}(f)G_{yy}(f)}$$

Where $G_{xy}(f)$ is the estimated cross-spectral density between x and y, while $G_{xx}(f)$ and $G_{yy}(f)$ are the autospectrum of x and y respectively.

The result is a symmetric matrix of size $[n\_channels \times n\_channels]$ bearing no information about the directionality of the interaction, with values within the range $[0, 1]$.

---

class dyconnmap.fc.coherence.**Coherence**(*fb*, *fs*, *pairs=None*, ***kwargs*)

Bases: *dyconnmap.fc.estimator.Estimator*

An *dyconnmap.fc.Estimator* class that implements *dyconnmap.fc.coherence*.

See also:

dyconnmap.fc.coherence Coherence

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*, *data_against=None*)

Returns

- **ts** (*complex array-like, shape(n_channels, n_channels, n_samples)*) – Estimated PLV time series (complex valued).
- **avg** (*array-like, shape(n_channels, n_channels)*) – Average PLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*ts1*, *ts2*)

Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

preprocess(*data*)
Preprocess the data.

dyconnmap.fc.coherence.**coherence**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)
Coherence

Estimate the Coherence for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **coh** – Estimated Coherence.

Return type array-like, shape(n_channels, n_channels)

See also:

dyconnmap.fc.Coherece Coherece (Class Estimator)

dyconnmap.fc.icoherence Imaginary Coherence

1.1.3.5 dyconnmap.fc.corr module

Correlation

@see https://docs.scipy.org/doc/numpy/reference/generated/numpy.corrcoef.html

class dyconnmap.fc.corr.**Corr**(*fb=None*, *fs=None*, *pairs=None*)
Bases: *dyconnmap.fc.estimator.Estimator*

Correlation

See also:

dyconnmap.fc.corr Correlation

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*, *data_against=None*)

Returns **r** – Estimated correlation values.

Return type array-like, shape(n_rois, n_rois, n_samples)

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*signal1*, *signal2*)

Returns

- **r** (*array-like, shape(1, n_samples)*) – Estimated correlation values (real valued).
- **_** (*None*) – None.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

mean(*value*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

dyconnmap.fc.corr.**corr**(*data*, *fb=None*, *fs=None*, *pairs=None*)
Correlation

Compute the correlation for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **r** – Estimated correlation values.

Return type array-like, shape(n_rois, n_rois)

See also:

dyconnmap.fc.Corr Correlation

1.1.3.6 dyconnmap.fc.cos module

Cosine

dyconnmap.fc.cos.**cos**(*data: numpy.ndarray*, *fb: Optional[float] = None*, *fs: Optional[float] = None*, *pairs: Optional[List[List[int]]] = None*)

Cosine

Compute the correlation for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **c** – Estimated connectivity matrix.

Return type array-like, shape(n_rois, n_rois)

1.1.3.7 dyconnmap.fc.crosscorr module

Cross Correlation

see @https://docs.scipy.org/doc/numpy/reference/generated/numpy.correlate.html

dyconnmap.fc.crosscorr.**crosscorr**(*data*, *fb*, *fs*, *pairs=None*)

1.1.3.8 dyconnmap.fc.dpli module

Directed Phase Lag Index

Directed Phase Lag Index (*dPLI*) was introduced in [*Stam2012*] to capture the phase and lag relationship as a measure of directed functional connectivity.

- if $0.5 \leq dPLI_{xy} \leq 1.0$, x is leading y
- if $0.0 \leq dPLI_{xy} = 0.5$, y is leading x
- if $dPLI_{xy} = 0.5$, neither x nor y is leading or lagging

---

dyconnmap.fc.dpli.**dpli**(*data*, *fb*, *fs*, *pairs=None*)

Directed Phase Lag Index

Estimate the Directed Phase Lag Index for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **dpliv** – Estimated Directed PLI.

Return type array-like, shape(n_channels, n_channels)

See also:

dyconnmap.fc.PLI Phase Lag Index

1.1.3.9 dyconnmap.fc.esc module

Envelope-to-Signal Correlation

Proposed by Bruns and Eckhorn [**Bruns2004_**], Envelope-to-Signal Correlation (*ESC*) is similar to Amplitude Envelope Correlation (*dyconnmap.fc.aec*), but the the amplitude of the lower frequency oscillation is signed; and thus the phase information is preserved.

$$r_{ESC} = \text{corr}(\chi_{lo}, \alpha_{hi})$$

Where χ is the input signal filtered to the frequency band lo and α denotes the Instantaneous Amplitude of the same input signal at the frequency band hi.

---

dyconnmap.fc.esc.**esc**(*data*, *fb_lo*, *fb_hi*, *fs*)

Envelope-Signal-Correlation

Estimate the Envelope-Signal-Correlation the given `data`.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

1.1.3.10 dyconnmap.fc.estimator module

Base class for estimators

class dyconnmap.fc.estimator.**Estimator**(*fb: Optional[float] = None*, *fs: Optional[float] = None*, *pairs: Optional[List[List[int]]] = None*)

Bases: object

Base class for estimators.

Through this abstract class, an estimator can provide the necessary methods to be used for a time-varying functional connectivity analysis.

See also:

dynfunconn.tvfcgs.tvfcgs Time-Varying Functional Connectivity Graphs

dynfunconn.tvfcgs.tvfcgs_cfc Time-Varying Functional Connectivity Graphs (for Cross frequency Coupling)

dynfunconn.tvfcgs.tvfcgs_ts Time-Varying Functional Connectivity Graphs (from time series)

estimate(*data: numpy.ndarray*, *data_against: Optional[numpy.ndarray] = None*)
Estimate the connectivity within the given dataset.

estimate_pair(*signal1: numpy.ndarray*, *signal2: numpy.ndarray*)
Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

mean(*ts: numpy.ndarray*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

prepare_pairs(*rois: int*, *symmetric: bool = False*)
Prepares a list of indices of ROIs sourced in an estimator.

Parameters **rois** (*int*) – Number of rois

preprocess(*data: numpy.ndarray*)
Preprocess the data.

typeCast(*data: numpy.ndarray*, *cast_type: numpy.dtype*)

1.1.3.11 dyconnmap.fc.glm module

General Linear Model

General linear modeling (*GLM*) is used widely in neuroimaging [*Penny2006*] to detect coupling between a low and higher frequency.

$$\chi_{hf} = X_{\beta} + e$$

Where β are the corresponding regression coefficients and e is the additive Gaussian noise. Finally, X is the design matrix of size $n \times 3$ (n the number of samples). Columns 1 and 2, contain the cosines and sines counterparts of the instantaneous phases (of the low frequency) of the predictors, while the third row only 1s.

---

dyconnmap.fc.glm.**glm**(*data*, *fb_lo*, *fb_hi*, *fs*, *pairs=None*, *window_size=- 1*)

General Linear Model

Estimate the r^2 for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- **window_size** (*int*) – The number of samples that will be used in each window.

Returns

- **ts** (*complex array-like, shape(n_windows, n_channels, n_channels)*) – Estimated r^2 time series (in each window).
- **ts_avg** (*complex array-like, shape(n_channels, n_channels)*) – Average r^2 (across all windows).

1.1.3.12 dyconnmap.fc.icoherence module

Imaginary Coherence

Imaginary Coherence (*ICoh*)

Nolte, G., Bai, O., Wheaton, L., Mari, Z., Vorbach, S., & Hallett, M. (2004). Identifying true brain interaction from EEG data using the imaginary part of coherency. Clinical neurophysiology, 115(10), 2292-2307. Chicago

class dyconnmap.fc.icoherence.**ICoherence**(*fb*, *fs*, *pairs=None*)
Bases: *dyconnmap.fc.estimator.Estimator*

Imaginary Coherence

estimate(*data*, *data_against=None*)
Estimate the connectivity within the given dataset.

estimate_pair(*signal1*, *signal2*)
Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

preprocess(*data*)
Preprocess the data.

dyconnmap.fc.icoherence.**icoherence**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)
Imaginary Coherence

Compute the Imaginary part of Coherence for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **icoh** – Estimated Imaginary part of Coherence.

Return type array-like, shape(n_channels, n_channels)

See also:

dyconnmap.fc.coherence Coherence

1.1.3.13 dyconnmap.fc.iplv module

Imaginary part of Phase Locking Value

Imaginary Phase Locking Value (*IPLV*) was proposed to resolve PLV's sensitivity to volume conduction and common reference effects.

IPLV is computed similarly as PLV, but taking the imaginary part of the summation:

$$ImPLV = \frac{1}{N}\left|Im\left(\sum_{t=1}^{N} e^{i(\phi_{j1}(t)-\phi_{j2}(t))}\right)\right|$$

class dyconnmap.fc.iplv.**IPLV**(*fb=None*, *fs=None*, *pairs=None*)

Bases: *dyconnmap.fc.estimator.Estimator*

Imaginary part of PLV (iPLV)

See also:

dyconnmap.fc.iplv Imaginary part of PLV

dyconnmap.fc.plv Phase Locking Value

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*)

Returns

- *ts*
- *avg*

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*ts1*, *ts2*)

Returns

- **ts** (*array-like, shape(1, n_samples)*) – Estimated iPLV time series.
- **avg** (*float*) – Average iPLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

mean(*ts*)

The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)

Preprocess the data.

dyconnmap.fc.iplv.**iplv**(*data*, *fb=None*, *fs=None*, *pairs=None*)

Imaginary part of Phase Locking Value

Compute the Imaginary part of Phase Locking Value for the given *data*, between the *pairs* (if given) of rois.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.

- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns

- **ts** (*array-like, shape(n_rois, n_rois, n_samples)*) – Estimated IPLV time series.
- **avg** (*array-like, shape(n_rois, n_rois)*) – Average IPLV.

dyconnmap.fc.iplv.**iplv_fast**(*data*, *pairs=None*)
Imaginary part of Phase Locking Value

1.1.3.14 dyconnmap.fc.mui module

Mutual Information

Mutual Information (*MI*),

---

dyconnmap.fc.mui.**mutual_information**(*x*, *y*, *n_bins=10*)

1.1.3.15 dyconnmap.fc.nesc module

Amplitude-Normalized Envelope-to-Signal Correlation

---

dyconnmap.fc.nesc.**nesc**(*data*, *f_lo*, *f_hi*, *fs*, *pairs=None*)
Amplitude-Normalized Envelope-to-Signal-Correlation

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

1.1.3.16 dyconnmap.fc.pac module

Phase-Amplitude Coupling

Phase-Amplitude Coupling (*PAC*) is the most famous and prominent approach for studuying the Cross Frequency Coupling between slow and faster oscillations. The phase of a low frequency drive the power of a higher frequency.

class dyconnmap.fc.pac.**PAC**(*f_lo*, *f_hi*, *fs*, *estimator*, *pairs=None*)

Bases: *dyconnmap.fc.estimator.Estimator*

Phase Amplitude Coupling (PAC)

estimate(*phases*, *phases_lohi*)

Estimate the connectivity within the given dataset.

estimate_pair(*phase1*, *phase1_lohi*)

Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

mean(*ts*)

The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)

Preprocess the data.

dyconnmap.fc.pac.**pac**(*data*, *f_lo*, *f_hi*, *fs*, *estimator*, *pairs=None*)

Phase-Amplitude Coupling

Compute the Phase Amplitude Couplgin using the given estimator for the given *data*, between the specified *pairs* of channels.

Parameters

- **data** (*array-like, shape = [n_electrodes, n_samples]*) – Multichannel recording data.
- **pairs** (*array-like*) – Each element is a tuple of length two.
- **f_lo** (*list of length 2*) – The low and high frequencies.
- **f_hi** (*list of length 2*) – The low and high frequencies.
- **fs** (*float*) – Sampling frequency.
- **estimator** (*iplv | plv | pli | corr*) – Estimator used Valid options:

  'iplv' : Imaginary Phase Locking Value 'plv' : Phase Locking Value 'pli' : Phase Lag Index

Returns

- **ts** (*complex array-like, shape = [n_electrodes, n_electrodes, n_samples]*) – The PAC computed each time series.

- **avg** (*complex array-like, shape = [n_electrodes, n_electrodes]*) – The average PAC across all samples.

1.1.3.17 dyconnmap.fc.partcorr module

Partial Correlation

dyconnmap.fc.partcorr.**partcorr**(*data*, *fb*, *fs*, *pairs=None*)
Partial correlation

1.1.3.18 dyconnmap.fc.pec module

Power-Envelope Correlation

Similarly to *dyconnmap.fc.aec*, we can use the followig formula to estimate the correlations in power between the different frequency bands [*Friston1996*].

$$r_{PAEC} = \text{corr}(\alpha_{lo}^2, \alpha_{hi}^2)$$

---

dyconnmap.fc.pec.**pec**(*data*, *fb_lo*, *fb_hi*, *fs*)
Power Envelope Correlation

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

1.1.3.19 dyconnmap.fc.pli module

Phase Lag Index

Phase Lag Index (*PLI*) [*Stam2007*], proposed as an alternative (to PLV) phase synchronization estimator that is less prone to the effects of common sources (namely, volume conduction and active reference electrodes). These effects can artificially generate functional connectivity as the same signal signal is measured at different electrodes [*Hardmeier2014*].

PLI estimates the asymmetry in the distribution of two time series' instantaneous phase differences.

Given two time series of equal length $x(t)$ and $y(t)$, we extract their respective instantaneous phases $\phi_x(t)$ and $\phi_y(t)$ using the Hilbert transform (consult *dyconnmap.analytic_signal* for more details). Then, for such a pair of phases, PLI is computed as follows:

$$PLI = |\left\langle sign[sin(\phi_x(t) - \phi_y(t))]\right\rangle|$$

Where, $sign$ refers to the signum function, left langle right rangle denotes the mean value and || the absolute value.

class dyconnmap.fc.pli.**PLI**(*fb=None*, *fs=None*, *pairs=None*)
Bases: *dyconnmap.fc.estimator.Estimator*

Phase Lag Index (PLI)

estimate(*data*, *data_against=None*)
Estimate the connectivity within the given dataset.

estimate_pair(*signal1*, *signal2*)
Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

mean(*ts*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

dyconnmap.fc.pli.**pli**(*data*, *fb=None*, *fs=None*, *pairs=None*)
Phase Lag Index

Compute the PLI for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns

- **ts** (*array-like, shape(n_rois, n_rois, n_samples)*) – Estimated PLI time series.
- **avg** (*array-like, shape(n_rois, n_rois)*) – Average PLI.

See also:

dyconnmap.fc.PLI Phase Lag Index

1.1.3.20 dyconnmap.fc.plv module

Phase Locking Value

One of the pioneer methods called Phase Locking Value (*PLV*) is discussed in [Lachaux1998]; it utilizes the Hilbert representation (consult *dyconnmap.analytic_signal* for more details) an EEG time series (of $N_{sensors}$) and quantifies their interaction based on their instantaneous phase in a specific band frequency.

So, for a pair of Instantaneous Phases of two time series of equal length, $\phi_{j1}(t)$ and $\phi_{j2}(t)$, the Phase Locking Value for each sample in time (t) is computed as:

$$e^{i(\phi_{j1}(t)-\phi_{j2}(t))}$$

A value of zero means that no coupling (or negligible) observed between two phases, while a value of one denotes a perfect synchronization.

---

class dyconnmap.fc.plv.**PLV**(*fb=None*, *fs=None*, *pairs=None*)

Bases: *dyconnmap.fc.estimator.Estimator*

Phase Locking Value (PLV)

See also:

dyconnmap.fc.plv Phase Locking Value

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*, *data_against=None*)

Returns

- **ts** (*complex array-like, shape(n_channels, n_channels, n_samples)*) – Estimated PLV time series (complex valued).
- **avg** (*array-like, shape(n_channels, n_channels)*) – Average PLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*signal1*, *signal2*)

Returns

- **ts** (*array-like, shape(1, n_samples)*) – Estimated PLV time series (real valued).
- **avg** (*float*) – Average PLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

mean(*ts*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

dyconnmap.fc.plv.**plv**(*data*, *fb=None*, *fs=None*, *pairs=None*)
Phase Locking Value

Compute the PLV for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns

- **ts** (*array-like, shape(n_rois, n_rois, n_samples)*) – Estimated PLV time series.
- **avg** (*array-like, shape(n_rois, n_rois)*) – Average PLV.

See also:

dyconnmap.fc.PLV Phase Locking Value (Class Estimator)

dyconnmap.fc.iplv Imaginary part of PLV

dyconnmap.fc.pli Phase Lag Index

dyconnmap.fc.plv.**plv_fast**(*data*, *pairs=None*)
Phase Locking Value

1.1.3.21 dyconnmap.fc.rho_index module

index

$$\rho_{p,q}(t) = \frac{H_{max} - H}{H_{max}}$$

Where H is the Shannon entropy estimated within M number of phase bins, and $H_{max} = ln(M)$ is the maximal entropy and and p_k is the relative frequency of finding frequency difference in the k th phase bin.

$$H = -\sum_{k=1}^{M} p_k ln(p_k)$$

The computed value varies within the range $[0, 1]$

---

dyconnmap.fc.rho_index.**rho_index**(*data*, *n_bins*, *fb*, *fs*, *pairs=None*)

Synchronization Index

Compute the synchronization index for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **n_bins** (*int*) – Number of bins.
- **fb** (*list of length 2*) – The low and high frequencies.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **rho** – Estimated rho index.

Return type array-likem, shape(n_channels, n_channels)

1.1.3.22 dyconnmap.fc.wpli module

Weighted Phase Lag Index and Debiased Weighted Phase Lag Index

PLI is prone to noise and volume conduction effects; thus, Weighted Lag Index (*wPLI*) [Vinck2011] was proposed in [Vinck, 2011] alongside with an alternative, debiased design (*dwPLI*). Similar to PLI, wPLI operates on the cross-spectrum of two real-valued signals; but, it furthermore weights the cross-spectrum with the magnitude of the imaginary component.

$$wPLI = \frac{|E\{\Im(Z)\}|}{E\{\Im(Z)\}} = \frac{|E\{|\Im(Z)|sign(\Im(Z))\}|}{E\{|\Im(Z)|\}}$$

Furthermore, to overcome the possible sample-bias, the authors defined a debiased variant of wPLI:

$$dwPLI = \frac{\sum_{j=1}^{N}\sum_{k\neq j}\Im\{X_j\}\Im\{X_k\}}{\sum_{j=1}^{N}\sum_{k\neq j}|\Im\{X_j\}\Im\{X_k\}|}$$

---

dyconnmap.fc.wpli.**dwpli**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)

Debiased Weighted Phase Lag Index

Compute the Debiased Weight Phase Lad Index for the given *data*, between the specified *pairs* of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.

- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **dwpli** – Estimated Debiased Weighted PLI.

Return type array-like, shape(n_channels, n_channels)

dyconnmap.wpli.wpli: Weighted Phase Lag Index

`dyconnmap.fc.wpli.wpli`(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)

Weighted Phase Lag Index

Compute the Weight Phase Lad Index for the given *data*, between the specified *pairs* of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **wpli** – Estimated Weighted PLI.

Return type array-like, shape(n_channels, n_channels)

Notes

1. The resulting wpli value has a phase shift.
2. The results do not match those from MATLAB because of the *mlab.cpsd*.

dyconnmap.wpli.dwpli: Debiased Weighted Phase Lag Index

1.1.3.23 Module contents

class dyconnmap.fc.**Coherence**(*fb*, *fs*, *pairs=None*, ***kwargs*)

Bases: *dyconnmap.fc.estimator.Estimator*

An *dyconnmap.fc.Estimator* class that implements *dyconnmap.fc.coherence*.

See also:

dyconnmap.fc.coherence Coherence

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*, *data_against=None*)

Returns

- **ts** (*complex array-like, shape(n_channels, n_channels, n_samples)*) – Estimated PLV time series (complex valued).
- **avg** (*array-like, shape(n_channels, n_channels)*) – Average PLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*ts1*, *ts2*)
Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

preprocess(*data*)
Preprocess the data.

class dyconnmap.fc.**Corr**(*fb=None*, *fs=None*, *pairs=None*)
Bases: *dyconnmap.fc.estimator.Estimator*

Correlation

See also:

dyconnmap.fc.corr Correlation

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*, *data_against=None*)

Returns **r** – Estimated correlation values.

Return type array-like, shape(n_rois, n_rois, n_samples)

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*signal1*, *signal2*)

Returns

- **r** (*array-like, shape(1, n_samples)*) – Estimated correlation values (real valued).
- **_** (*None*) – None.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

mean(*value*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

class dyconnmap.fc.**Estimator**(*fb: Optional[float] = None*, *fs: Optional[float] = None*, *pairs: Optional[List[List[int]]] = None*)

Bases: `object`

Base class for estimators.

Through this abstract class, an estimator can provide the necessary methods to be used for a time-varying functional connectivity analysis.

See also:

dynfunconn.tvfcgs.tvfcgs Time-Varying Functional Connectivity Graphs

dynfunconn.tvfcgs.tvfcgs_cfc Time-Varying Functional Connectivity Graphs (for Cross frequency Coupling)

dynfunconn.tvfcgs.tvfcgs_ts Time-Varying Functional Connectivity Graphs (from time series)

estimate(*data: numpy.ndarray*, *data_against: Optional[numpy.ndarray] = None*)
Estimate the connectivity within the given dataset.

estimate_pair(*signal1: numpy.ndarray*, *signal2: numpy.ndarray*)
Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

mean(*ts: numpy.ndarray*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

prepare_pairs(*rois: int*, *symmetric: bool = False*)
Prepares a list of indices of ROIs sourced in an estimator.

Parameters **rois** (*int*) – Number of rois

preprocess(*data: numpy.ndarray*)
Preprocess the data.

typeCast(*data: numpy.ndarray*, *cast_type: numpy.dtype*)

class dyconnmap.fc.**IPLV**(*fb=None*, *fs=None*, *pairs=None*)

Bases: *dyconnmap.fc.estimator.Estimator*

Imaginary part of PLV (iPLV)

See also:

dyconnmap.fc.iplv Imaginary part of PLV

dyconnmap.fc.plv Phase Locking Value

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*)

Returns

- *ts*
- *avg*

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*ts1*, *ts2*)

Returns

- **ts** (*array-like, shape(1, n_samples)*) – Estimated iPLV time series.
- **avg** (*float*) – Average iPLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

mean(*ts*)

The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)

Preprocess the data.

class dyconnmap.fc.**PAC**(*f_lo*, *f_hi*, *fs*, *estimator*, *pairs=None*)

Bases: *dyconnmap.fc.estimator.Estimator*

Phase Amplitude Coupling (PAC)

estimate(*phases*, *phases_lohi*)

Estimate the connectivity within the given dataset.

estimate_pair(*phase1*, *phase1_lohi*)

Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

mean(*ts*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

class dyconnmap.fc.**PLI**(*fb=None*, *fs=None*, *pairs=None*)
Bases: *dyconnmap.fc.estimator.Estimator*

Phase Lag Index (PLI)

estimate(*data*, *data_against=None*)
Estimate the connectivity within the given dataset.

estimate_pair(*signal1*, *signal2*)
Estimate the connectivity between two signals (time series).

Notes

This is invoked from cross-frequency coupling methods.

mean(*ts*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

class dyconnmap.fc.**PLV**(*fb=None*, *fs=None*, *pairs=None*)
Bases: *dyconnmap.fc.estimator.Estimator*

Phase Locking Value (PLV)

See also:

dyconnmap.fc.plv Phase Locking Value

dyconnmap.tvfcg Time-Varying Functional Connectivity Graphs

estimate(*data*, *data_against=None*)

Returns

- **ts** (*complex array-like, shape(n_channels, n_channels, n_samples)*) – Estimated PLV time series (complex valued).

- **avg** (*array-like, shape(n_channels, n_channels)*) – Average PLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

estimate_pair(*signal1*, *signal2*)

Returns

- **ts** (*array-like, shape(1, n_samples)*) – Estimated PLV time series (real valued).
- **avg** (*float*) – Average PLV.

Notes

Called from *dyconnmap.tvfcgs.tvfcg*.

mean(*ts*)
The function used to compute the mean synchronization in a timeseries.

This is needed because some estimators produce complex (imaginary), and special treatment is needed (i.e. taking only the real part).

Returns **mtx** – The average synchronization.

Return type array-like

preprocess(*data*)
Preprocess the data.

dyconnmap.fc.**aec**(*data*, *fb_lo*, *fb_hi*, *fs*)
Amplitude Envelope Correlation

Estimate the Amplitude-Envelope Correlation for the given `data`.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

dyconnmap.fc.**coherence**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)
Coherence

Estimate the Coherence for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.

- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **coh** – Estimated Coherence.

Return type array-like, shape(n_channels, n_channels)

See also:

dyconnmap.fc.Coherece Coherece (Class Estimator)

dyconnmap.fc.icoherence Imaginary Coherence

dyconnmap.fc.**corr**(*data*, *fb=None*, *fs=None*, *pairs=None*)

Correlation

Compute the correlation for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **r** – Estimated correlation values.

Return type array-like, shape(n_rois, n_rois)

See also:

dyconnmap.fc.Corr Correlation

dyconnmap.fc.**cos**(*data: numpy.ndarray*, *fb: Optional[float] = None*, *fs: Optional[float] = None*, *pairs: Optional[List[List[int]]] = None*)

Cosine

Compute the correlation for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **c** – Estimated connectivity matrix.

Return type array-like, shape(n_rois, n_rois)

dyconnmap.fc.**crosscorr**(*data*, *fb*, *fs*, *pairs=None*)

dyconnmap.fc.**dpli**(*data*, *fb*, *fs*, *pairs=None*)

Directed Phase Lag Index

Estimate the Directed Phase Lag Index for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **dpliv** – Estimated Directed PLI.

Return type array-like, shape(n_channels, n_channels)

See also:

dyconnmap.fc.PLI Phase Lag Index

dyconnmap.fc.**dwpli**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)

Debiased Weighted Phase Lag Index

Compute the Debiased Weight Phase Lad Index for the given *data*, between the specified *pairs* of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **dwpli** – Estimated Debiased Weighted PLI.

Return type array-like, shape(n_channels, n_channels)

dyconnmap.wpli.wpli: Weighted Phase Lag Index

dyconnmap.fc.**esc**(*data*, *fb_lo*, *fb_hi*, *fs*)

Envelope-Signal-Correlation

Estimate the Envelope-Signal-Correlation the given `data`.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.

- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

dyconnmap.fc.**glm**(*data*, *fb_lo*, *fb_hi*, *fs*, *pairs=None*, *window_size=- 1*)

General Linear Model

Estimate the r^2 for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- **window_size** (*int*) – The number of samples that will be used in each window.

Returns

- **ts** (*complex array-like, shape(n_windows, n_channels, n_channels)*) – Estimated r^2 time series (in each window).
- **ts_avg** (*complex array-like, shape(n_channels, n_channels)*) – Average r^2 (across all windows).

dyconnmap.fc.**icoherence**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)

Imaginary Coherence

Compute the Imaginary part of Coherence for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **icoh** – Estimated Imaginary part of Coherence.

Return type array-like, shape(n_channels, n_channels)

See also:

dyconnmap.fc.coherence Coherence

dyconnmap.fc.**iplv**(*data*, *fb=None*, *fs=None*, *pairs=None*)
Imaginary part of Phase Locking Value

Compute the Imaginary part of Phase Locking Value for the given *data*, between the *pairs* (if given) of rois.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns

- **ts** (*array-like, shape(n_rois, n_rois, n_samples)*) – Estimated IPLV time series.
- **avg** (*array-like, shape(n_rois, n_rois)*) – Average IPLV.

dyconnmap.fc.**iplv_fast**(*data*, *pairs=None*)
Imaginary part of Phase Locking Value

dyconnmap.fc.**mutual_information**(*x*, *y*, *n_bins=10*)

dyconnmap.fc.**nesc**(*data*, *f_lo*, *f_hi*, *fs*, *pairs=None*)
Amplitude-Normalized Envelope-to-Signal-Correlation

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

dyconnmap.fc.**pac**(*data*, *f_lo*, *f_hi*, *fs*, *estimator*, *pairs=None*)
Phase-Amplitude Coupling

Compute the Phase Amplitude Couplgin using the given estimator for the given *data*, between the specified *pairs* of channels.

Parameters

- **data** (*array-like, shape = [n_electrodes, n_samples]*) – Multichannel recording data.
- **pairs** (*array-like*) – Each element is a tuple of length two.
- **f_lo** (*list of length 2*) – The low and high frequencies.
- **f_hi** (*list of length 2*) – The low and high frequencies.
- **fs** (*float*) – Sampling frequency.
- **estimator** (*iplv | plv | pli | corr*) – Estimator used Valid options:

  'iplv' : Imaginary Phase Locking Value 'plv' : Phase Locking Value 'pli' : Phase Lag Index

Returns

- **ts** (*complex array-like, shape = [n_electrodes, n_electrodes, n_samples]*) – The PAC computed each time series.
- **avg** (*complex array-like, shape = [n_electrodes, n_electrodes]*) – The average PAC across all samples.

dyconnmap.fc.**partcorr**(*data*, *fb*, *fs*, *pairs=None*)
Partial correlation

dyconnmap.fc.**pec**(*data*, *fb_lo*, *fb_hi*, *fs*)
Power Envelope Correlation

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** (*list of length 2*) – The low and high frequencies of the lower band.
- **fb_hi** (*list of length 2*) – The low and high frequencies of the upper band.
- **fs** (*float*) – Sampling frequency.

Returns **r** – Estimated Pearson correlation coefficient.

Return type array-like, shape(n_channels, n_channels)

dyconnmap.fc.**pli**(*data*, *fb=None*, *fs=None*, *pairs=None*)
Phase Lag Index

Compute the PLI for the given data, between the pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns

- **ts** (*array-like, shape(n_rois, n_rois, n_samples)*) – Estimated PLI time series.

- **avg** (*array-like, shape(n_rois, n_rois)*) – Average PLI.

See also:

dyconnmap.fc.PLI Phase Lag Index

dyconnmap.fc.**plv**(*data*, *fb=None*, *fs=None*, *pairs=None*)
Phase Locking Value

Compute the PLV for the given `data`, between the `pairs` (if given) of channels.

Parameters

- **data** (*array-like, shape(n_rois, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*float, optional*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns

- **ts** (*array-like, shape(n_rois, n_rois, n_samples)*) – Estimated PLV time series.
- **avg** (*array-like, shape(n_rois, n_rois)*) – Average PLV.

See also:

dyconnmap.fc.PLV Phase Locking Value (Class Estimator)

dyconnmap.fc.iplv Imaginary part of PLV

dyconnmap.fc.pli Phase Lag Index

dyconnmap.fc.**plv_fast**(*data*, *pairs=None*)
Phase Locking Value

dyconnmap.fc.**rho_index**(*data*, *n_bins*, *fb*, *fs*, *pairs=None*)
Synchronization Index

Compute the synchronization index for the given `data`, between the :attr:`pairs (if given) of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **n_bins** (*int*) – Number of bins.
- **fb** (*list of length 2*) – The low and high frequencies.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **rho** – Estimated rho index.

Return type array-likem, shape(n_channels, n_channels)

dyconnmap.fc.**wpli**(*data*, *fb*, *fs*, *pairs=None*, ***kwargs*)
Weighted Phase Lag Index

Compute the Weight Phase Lad Index for the given *data*, between the specified *pairs* of channels.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.
- ****kwargs** – Keyword arguments to be passed to `matplotlib.mlab.csd()`.

Returns **wpli** – Estimated Weighted PLI.

Return type array-like, shape(n_channels, n_channels)

Notes

1. The resulting wpli value has a phase shift.
2. The results do not match those from MATLAB because of the *mlab.cpsd*.

dyconnmap.wpli.dwpli: Debiased Weighted Phase Lag Index

1.1.4 dyconnmap.graphs package

1.1.4.1 Submodules

1.1.4.2 dyconnmap.graphs.e2e module

Edge-to-Edge Network

dyconnmap.graphs.e2e.**edge_to_edge**(*dfcgs: numpy.ndarray*) → numpy.ndarray
Edge-To-Edge

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer (undirected) graph. Each layer consists of a graph.

Returns **net**

Return type array-like

1.1.4.3 dyconnmap.graphs.gdd module

Graph Diffusion Distance

The Graph Diffusion Distance (GDD) metric (*Hammond2013*) is a measure of distance between two (positive) weighted graphs based on the Laplacian exponential diffusion kernel. The notion backing this metric is that two graphs are similar if they emit comparable patterns of information transmission.

This distance is computed by searching for a diffusion time t that maximizes the value of the Frobenius norm between the two diffusion kernels. The Laplacian operator is defined as $L = D - A$, where A is the positive symmetric data matrix and D is a diagonal degree matrix for the adjacency matrix A. The diffusion process (per vertex) on the adjacency matrix A is governed by a time-varying vector $u(t)R^N$. Thus, between each given pair of (vertices') weights i and j, their flux is quantified by $a_{ij}(u_i(t)u_j(t))$. The grand sum of these interactions is given by $\hat{u}_j(t) = \sum_i a_{ij}(u_i(t)u_j(t)) = -Lu(t)$. Given the initial condition $u^0, t = 0$ this sum has the following analytic solution $u(t) = exp(-tL)u^0$. The resulting matrix is known as the Laplacian exponential diffusion kernel. Letting the diffusion process run for t time we compute and store the diffusion patterns in each column. Finally, the actual distance measure between two adjacency matrices A_1 and A_2, at diffusion time t is given by:

$$(A_1, A_2; t) = exp(-tL_1) - exp(-tL_2)_F^2$$

where $_F$ is the Frobenious norm.

Notes

Based on the code accompanied the original paper. Available at https://www.researchgate.net/publication/259621918_A_Matlab_code_for_computing_the_GDD_presented_in_the_paper

---

dyconnmap.graphs.gdd.**graph_diffusion_distance**(*a: numpy.ndarray*, *b: numpy.ndarray*, *threshold: Optional[float] = 1e-14*) → Tuple[numpy.float32, numpy.float32]

Graph Diffusion Distance

Parameters

- **a** (*array-like, shape(N, N)*) – Weighted matrix.
- **b** (*array-like, shape(N, N)*) – Weighted matrix.
- **threshold** (*float*) – A threshold to filter out the small eigenvalues. If the you get NaN or INFs, try lowering this threshold.

Returns

- **gdd** (*float*) – The estimated graph diffusion distance.
- **xopt** (*float*) – Parameters (over given interval) which minimize the objective function. (see `scipy.optimize.fmindbound`)

1.1.4.4 dyconnmap.graphs.imd module

Ipsen-Mikhailov Distance

Given two graphs, this method quantifies their difference by comparing their spectral densities. This spectral density is computed as the sum of Lorentz distributions $\rho(\omega)$:

$$\rho(\omega) = K \sum_{i=1}^{N-1} \frac{\gamma}{(\omega - \omega_i)^2 + \gamma^2}$$

Where γ is the bandwidth, and K a normalization constant such that $\int_0^\infty \rho(\omega)d\omega = 1$. The spectral distance between two graphs G and H with densities $\rho_G(\omega)$ and $\rho_H(\omega)$ respectively, is defined as:

$$\epsilon = \sqrt{\int_0^\infty [\rho_G(\omega) - \rho_H(\omega)]^2 d(\omega)}$$

---

dyconnmap.graphs.imd.**im_distance**(*X: numpy.ndarray*, *Y: numpy.ndarray*, *bandwidth: Optional[float] = 1.0*) → float

Parameters

- **X** (*array-like, shape(N, N)*) – A weighted matrix.
- **Y** (*array-like, shape(N, N)*) – A weighted matrix.
- **bandwidth** (*float*) – Bandwidth of the kernel. Default *1.0*.

Returns **distance** – The estimated Ipsen-Mikhailov distance.

Return type float

1.1.4.5 dyconnmap.graphs.laplacian_energy module

Laplcian Energy

The Laplcian energy (LE) for a graph G is computed as

$$LE(G) = \sum_{i=1}^{n} |\mu_i - \frac{2m}{n}|(A_1, A_2; t) = exp(-tL_1) - exp(-tL_2)_F^2$$

Where μ_i denote the eigenvalue associated with the node of the Laplcian matrix of G (Laplcian spectrum) and $\frac{2m}{n}$ the average vertex degree.

For a details please go through the original work (**Gutman2006_**).

dyconnmap.graphs.laplacian_energy.**laplacian_energy**(*mtx: numpy.ndarray*) → float

Laplacian Energy

Parameters **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.

Returns **le** – The Laplacian Energy.

Return type float

1.1.4.6 dyconnmap.graphs.mi module

Mutual Information

Normalized Mutual Information (*NMI*) proposed by [Strehl2002] as an extension to Mutual Information [cover] to enable interpretations and comparisons between two partitions. Given the entropies $H(P^a) = -\sum_{i=1}^{k_a} \frac{n_i^a}{n} \log(\frac{n_i^a}{n})$ where n_i^a represents the number of patterns in group $C_i^a \in P^a$ (and computed for $H(P^b)$ accordingly); the initial matching of these two groups P^a and P^b in terms of mutual information is [*Fred2005*, *Strehl2002*]:

$$I(P^a, P^b) = \sum_{i=1}^{k_a} \sum_{j=1}^{k_b} \frac{n_{ij}^{ab}}{n} \log \left(\frac{\frac{n_{ij}ab}{n}}{\frac{n_i^a}{n} \frac{n_j^b}{n}} \right)$$

Where n_{ij}^{ab} denotes the number of shared patterns between the clusters C_i^a and C_j^b. By exploiting the definition of mutual information, the following property holds true: $I(P^a, P^b) \leq \frac{H(P^a)+H(P^b)}{2}$. This leads to the definition of NMI as:

$$NMI(A, B) = \frac{2I(P^a, P^b)}{H(P^a) + H(P^b)} = \frac{-2 \sum_{i=1}^{k_a} \sum_{j=1}^{k_b} n_{ij}^{ab} \log \left(\frac{n_{ij}^{ab} n}{n_i^a n_j^n} \right)}{\sum_{i=1}^{k_a} n_i^a \log \left(\frac{n_i^a}{n} \right) + \sum_{j=1}^{k_b} n_j^b \log \left(\frac{n_j^b}{n} \right)}$$

---

dyconnmap.graphs.mi.**mutual_information**(*indices_a: numpy.ndarray*, *indices_b: numpy.ndarray*) → Tuple[float, float]

Mutual Information

Parameters

- **indices_a** (*array-like, shape(n_samples)*) – Symbolic time series.
- **indices_b** (*array-like, shape(n_samples)*) – Symbolic time series.

Returns

- **MI** (*float*) – Mutual information.
- **NMI** (*float*) – Normalized mutual information.

1.1.4.7 dyconnmap.graphs.mpc module

Multilayer Participation Coefficient

---

dyconnmap.graphs.mpc.**multilayer_pc_degree**(*mlgraph: numpy.ndarray*) → numpy.ndarray

Multilayer Participation Coefficient (Degree)

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer (undirected) graph. Each layer consists of a graph.

Returns **mpc** – Participation coefficient based on the degree of the layers' nodes.

Return type array-like

dyconnmap.graphs.mpc.**multilayer_pc_gamma**(*mlgraph: numpy.ndarray*) → numpy.ndarray
Multilayer Participation Coefficient method from Guillon et al.

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer graph.

Returns **gamma** – Returns the original multilayer graph flattened, with the off diagional containing the estimated interlayer multilayer participation coefficient.

Return type array-like, shape(n_layers*n_rois, n_layers*n_rois)

dyconnmap.graphs.mpc.**multilayer_pc_strength**(*mlgraph: numpy.ndarray*) → numpy.ndarray
Multilayer Participation Coefficient (Strength)

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer (undirected) graph. Each layer consists of a graph.

Returns **mpc** – Participation coefficient based on the strength of the layers' nodes.

Return type array-like

1.1.4.8 dyconnmap.graphs.nodal module

Nodal network features

dyconnmap.graphs.nodal.**nodal_global_efficiency**(*mtx: numpy.ndarray*) → numpy.ndarray
Nodal Global Efficiency

Parameters **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.

Returns **nodal_ge** – The computed nodal global efficiency.

Return type array-like, shape(N, 1)

1.1.4.9 dyconnmap.graphs.spectral_euclidean_distance module

Spectral Euclidean Distance

The spectral distance between graphs is simply the Euclidean distance between the spectra.

$$d(G, H) = \sqrt{\sum_i (g_i - h_j)^2}$$

Notes

- The input graphs can be a standard adjency matrix, or a variant of Laplacian.

---

dyconnmap.graphs.spectral_euclidean_distance.**spectral_euclidean_distance**(*X: numpy.ndarray*, *Y: numpy.ndarray*) → float

Parameters

- **X** (*array-like, shape(N, N)*) – A weighted matrix.
- **Y** (*array-like, shape(N, N)*) – A weighted matrix:

Returns **distance** – The euclidean distance between the two spectrums.

Return type float

1.1.4.10 dyconnmap.graphs.spectral_k_distance module

Spectral-K Distance

Given two graphs G and H, we can use their k largest positive eigenvalues of their Laplacian counterparts to compute their distance.

$$d(G,H) = \begin{cases} \sqrt{\frac{\sum_{i=1}^{k}(g_i - h_i)^2}{\sum_{i=1}^{l} g_i^2}} & , \sum_{i=1}^{l} g_i^2 \leq \sum_{j=1}^{l} h_j^2 \\ \sqrt{\frac{\sum_{i=1}^{k}(g_i - h_i)^2}{\sum_{j=1}^{l} g_i^2}} & , \sum_{i=1}^{l} g_i^2 > \sum_{j=1}^{l} h_j^2 \end{cases}$$

Where g and h denote the spectrums of the Laplacian matrices.

This measure is non-negative, separated, symmetric and it satisfies the triangle inequality.

---

dyconnmap.graphs.spectral_k_distance.**spectral_k_distance**(*X: numpy.ndarray*, *Y: numpy.ndarray*, *k: int*) → float

Spectral-K Distance

Use the largest k eigenvalues of the given graphs to compute the distance between them.

Parameters

- **X** (*array-like, shape(N, N)*) – A weighted matrix.
- **Y** (*array-like, shape(N, N)*) – A weighted matrixY
- **k** (*int*) – Largest k eigenvalues to use.

Returns **distance** – Estimated distance based on selected largest eigenvalues.

Return type float

1.1.4.11 dyconnmap.graphs.threshold module

Thresholding schemes

Notes

- This is a direct translation from `Data Driven Topological Filtering of Brain Networks via Orthogonal Minimal Spanning Trees <https://github.com/stdimitr/topological_filtering_networks>'
- Original author is Stravros Dimitriadis <stidimitriadis@gmail.com>

---

dyconnmap.graphs.threshold.**k_core_decomposition**(*mtx: numpy.ndarray*, *threshold: float*) → numpy.ndarray

Threshold a binary graph based on the detected k-cores.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Binary matrix.
- **threshold** (*int*) – Degree threshold.

Returns **k_cores** – A binary matrix of the decomposed cores.

Return type array-like, shape(N, 1)

dyconnmap.graphs.threshold.**threshold_eco**(*mtx*)

dyconnmap.graphs.threshold.**threshold_global_cost_efficiency**(*mtx: numpy.ndarray*, *iterations: int*) → Tuple[numpy.ndarray, float, float, float]

Threshold a graph based on the Global Efficiency - Cost formula.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **iterations** (*int*) – Number of steps, as a resolution when search for optima.

Returns

- **binary_mtx** (*array-like, shape(N, N)*) – A binary mask matrix.
- **threshold** (*float*) – The threshold that maximizes the global cost efficiency.
- **global_cost_eff_max** (*float*) – Global cost efficiency.
- **efficiency** (*float*) – Global efficiency.
- **cost_max** (*float*) – Cost of the network at the maximum global cost efficiency

dyconnmap.graphs.threshold.**threshold_mean_degree**(*mtx: numpy.ndarray*, *threshold_mean_degree: int*) → numpy.ndarray

Threshold a graph based on the mean degree.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **threshold_mean_degree** (*int*) – Mean degree threshold.

Returns **binary_mtx** – A binary mask matrix.

Return type array-like, shape(N, N)

dyconnmap.graphs.threshold.**threshold_mst_mean_degree**(*mtx: numpy.ndarray*, *avg_degree: float*) → numpy.ndarray

Threshold a graph based on mean using minimum spanning trees.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **avg_degree** (*float*) – Mean degree threshold.

Returns **binary_mtx** – A binary mask matrix.

Return type array-like, shape(N, N)

dyconnmap.graphs.threshold.**threshold_omst_global_cost_efficiency**(*mtx: numpy.ndarray*, *n_msts: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, float, float, float, float]

Threshold a graph by optimizing the formula GE-C via orthogonal MSTs.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **n_msts** (*int or None*) – Maximum number of OMSTs to compute. Default *None*; an exhaustive computation will be performed.

Returns

- **nCIJtree** (*array-like, shape(n_msts, N, N)*) – A matrix containing all the orthogonal MSTs.
- **CIJtree** (*array-like, shape(N, N)*) – Resulting graph.
- **degree** (*float*) – The mean degree of the resulting graph.
- **global_eff** (*float*) – Global efficiency of the resulting graph.
- **global_cost_eff_max** (*float*) – The value where global efficiency - cost is maximized.
- **cost_max** (*float*) – Cost of the network at the maximum global cost efficiency.

dyconnmap.graphs.threshold.**threshold_shortest_paths**(*mtx: numpy.ndarray*, *treatment: Optional[bool] = False*) → numpy.ndarray

Threshold a graph via via shortest path identification using Dijkstra's algorithm.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **treatment** (*boolean*) – Convert the weights to distances by inversing the matrix. Also, fill the diagonal with zeroes. Default *false*.

Returns **binary_mtx** – A binary mask matrix.

Return type array-like, shape(N, N)

1.1.4.12 dyconnmap.graphs.vi module

Variation of Information

Variation of Information (*VI*) [Meilla2007] is an information theoretic criterion for comparing two partitions. It is based on the classic notions of entropy and mutual information. In a nutshell, VI measures the amount of information that is lost or gained in changing from clustering A to clustering B. VI is a true metric, is always non-negative and symmetric. The following formula is used to compute the VI between two groups:

$$VI(A, B) = [H(A) - I(A, B)] + [H(B) - I(A, B)]$$

Where H denotes the entropy computed for each partition separately, and I the mutual information between clusterings A and B.

The resulting distance score can be adjusted to bound it between $[0, 1]$ as follows:

$$VI^*(A, B) = \frac{1}{\log n} VI(A, B)$$

---

dyconnmap.graphs.vi.**variation_information**(*indices_a: numpy.ndarray*, *indices_b: numpy.ndarray*) → float

Variation of Information

Parameters

- **indices_a** (*array-like, shape(n_samples)*) – Symbolic time series.
- **indices_b** (*array-like, shape(n_samples)*) – Symbolic time series.

Returns **vi** – Variation of information.

Return type float

1.1.4.13 Module contents

dyconnmap.graphs.**edge_to_edge**(*dfcgs: numpy.ndarray*) → numpy.ndarray

Edge-To-Edge

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer (undirected) graph. Each layer consists of a graph.

Returns **net**

Return type array-like

dyconnmap.graphs.**graph_diffusion_distance**(*a: numpy.ndarray*, *b: numpy.ndarray*, *threshold: Optional[float] = 1e-14*) → Tuple[numpy.float32, numpy.float32]

Graph Diffusion Distance

Parameters

- **a** (*array-like, shape(N, N)*) – Weighted matrix.
- **b** (*array-like, shape(N, N)*) – Weighted matrix.
- **threshold** (*float*) – A threshold to filter out the small eigenvalues. If the you get NaN or INFs, try lowering this threshold.

Returns

- **gdd** (*float*) – The estimated graph diffusion distance.
- **xopt** (*float*) – Parameters (over given interval) which minimize the objective function. (see `scipy.optimize.fmindbound`)

dyconnmap.graphs.**im_distance**(*X: numpy.ndarray*, *Y: numpy.ndarray*, *bandwidth: Optional[float] = 1.0*) → float

Parameters

- **X** (*array-like, shape(N, N)*) – A weighted matrix.
- **Y** (*array-like, shape(N, N)*) – A weighted matrix.
- **bandwidth** (*float*) – Bandwidth of the kernel. Default *1.0*.

Returns **distance** – The estimated Ipsen-Mikhailov distance.

Return type float

dyconnmap.graphs.**k_core_decomposition**(*mtx: numpy.ndarray*, *threshold: float*) → numpy.ndarray
Threshold a binary graph based on the detected k-cores.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Binary matrix.
- **threshold** (*int*) – Degree threshold.

Returns **k_cores** – A binary matrix of the decomposed cores.

Return type array-like, shape(N, 1)

dyconnmap.graphs.**laplacian_energy**(*mtx: numpy.ndarray*) → float
Laplacian Energy

Parameters **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.

Returns **le** – The Laplacian Energy.

Return type float

dyconnmap.graphs.**multilayer_pc_degree**(*mlgraph: numpy.ndarray*) → numpy.ndarray
Multilayer Participation Coefficient (Degree)

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer (undirected) graph. Each layer consists of a graph.

Returns **mpc** – Participation coefficient based on the degree of the layers' nodes.

Return type array-like

dyconnmap.graphs.**multilayer_pc_gamma**(*mlgraph: numpy.ndarray*) → numpy.ndarray
Multilayer Participation Coefficient method from Guillon et al.

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer graph.

Returns **gamma** – Returns the original multilayer graph flattened, with the off diagional containing the estimated interlayer multilayer participation coefficient.

Return type array-like, shape(n_layers*n_rois, n_layers*n_rois)

dyconnmap.graphs.**multilayer_pc_strength**(*mlgraph: numpy.ndarray*) → numpy.ndarray
Multilayer Participation Coefficient (Strength)

Parameters **mlgraph** (*array-like, shape(n_layers, n_rois, n_rois)*) – A multilayer (undirected) graph. Each layer consists of a graph.

Returns **mpc** – Participation coefficient based on the strength of the layers' nodes.

Return type array-like

dyconnmap.graphs.**mutual_information**(*indices_a: numpy.ndarray*, *indices_b: numpy.ndarray*) → Tuple[float, float]
Mutual Information

Parameters

- **indices_a** (*array-like, shape(n_samples)*) – Symbolic time series.
- **indices_b** (*array-like, shape(n_samples)*) – Symbolic time series.

Returns

- **MI** (*float*) – Mutual information.
- **NMI** (*float*) – Normalized mutual information.

dyconnmap.graphs.**nodal_global_efficiency**(*mtx: numpy.ndarray*) → numpy.ndarray
Nodal Global Efficiency

Parameters **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.

Returns **nodal_ge** – The computed nodal global efficiency.

Return type array-like, shape(N, 1)

dyconnmap.graphs.**spectral_euclidean_distance**(*X: numpy.ndarray*, *Y: numpy.ndarray*) → float

Parameters

- **X** (*array-like, shape(N, N)*) – A weighted matrix.
- **Y** (*array-like, shape(N, N)*) – A weighted matrix:

Returns **distance** – The euclidean distance between the two spectrums.

Return type float

dyconnmap.graphs.**spectral_k_distance**(*X: numpy.ndarray*, *Y: numpy.ndarray*, *k: int*) → float
Spectral-K Distance

Use the largest k eigenvalues of the given graphs to compute the distance between them.

Parameters

- **X** (*array-like, shape(N, N)*) – A weighted matrix.
- **Y** (*array-like, shape(N, N)*) – A weighted matrixY
- **k** (*int*) – Largest k eigenvalues to use.

Returns **distance** – Estimated distance based on selected largest eigenvalues.

Return type float

dyconnmap.graphs.**threshold_eco**(*mtx*)

dyconnmap.graphs.**threshold_global_cost_efficiency**(*mtx: numpy.ndarray*, *iterations: int*) → Tuple[numpy.ndarray, float, float, float]

Threshold a graph based on the Global Efficiency - Cost formula.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **iterations** (*int*) – Number of steps, as a resolution when search for optima.

Returns

- **binary_mtx** (*array-like, shape(N, N)*) – A binary mask matrix.
- **threshold** (*float*) – The threshold that maximizes the global cost efficiency.
- **global_cost_eff_max** (*float*) – Global cost efficiency.
- **efficiency** (*float*) – Global efficiency.
- **cost_max** (*float*) – Cost of the network at the maximum global cost efficiency

dyconnmap.graphs.**threshold_mean_degree**(*mtx: numpy.ndarray*, *threshold_mean_degree: int*) → numpy.ndarray

Threshold a graph based on the mean degree.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **threshold_mean_degree** (*int*) – Mean degree threshold.

Returns **binary_mtx** – A binary mask matrix.

Return type array-like, shape(N, N)

dyconnmap.graphs.**threshold_mst_mean_degree**(*mtx: numpy.ndarray*, *avg_degree: float*) → numpy.ndarray

Threshold a graph based on mean using minimum spanning trees.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **avg_degree** (*float*) – Mean degree threshold.

Returns **binary_mtx** – A binary mask matrix.

Return type array-like, shape(N, N)

dyconnmap.graphs.**threshold_omst_global_cost_efficiency**(*mtx: numpy.ndarray*, *n_msts: Optional[int] = None*) → Tuple[numpy.ndarray, numpy.ndarray, float, float, float, float]

Threshold a graph by optimizing the formula GE-C via orthogonal MSTs.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.

- **n_msts** (*int or None*) – Maximum number of OMSTs to compute. Default *None*; an exhaustive computation will be performed.

Returns

- **nCIJtree** (*array-like, shape(n_msts, N, N)*) – A matrix containing all the orthogonal MSTs.
- **CIJtree** (*array-like, shape(N, N)*) – Resulting graph.
- **degree** (*float*) – The mean degree of the resulting graph.
- **global_eff** (*float*) – Global efficiency of the resulting graph.
- **global_cost_eff_max** (*float*) – The value where global efficiency - cost is maximized.
- **cost_max** (*float*) – Cost of the network at the maximum global cost efficiency.

dyconnmap.graphs.**threshold_shortest_paths**(*mtx: numpy.ndarray*, *treatment: Optional[bool] = False*) → numpy.ndarray

Threshold a graph via via shortest path identification using Dijkstra's algorithm.

Parameters

- **mtx** (*array-like, shape(N, N)*) – Symmetric, weighted and undirected connectivity matrix.
- **treatment** (*boolean*) – Convert the weights to distances by inversing the matrix. Also, fill the diagonal with zeroes. Default *false*.

Returns **binary_mtx** – A binary mask matrix.

Return type array-like, shape(N, N)

dyconnmap.graphs.**variation_information**(*indices_a: numpy.ndarray*, *indices_b: numpy.ndarray*) → float

Variation of Information

Parameters

- **indices_a** (*array-like, shape(n_samples)*) – Symbolic time series.
- **indices_b** (*array-like, shape(n_samples)*) – Symbolic time series.

Returns **vi** – Variation of information.

Return type float

1.1.5 dyconnmap.sim_models package

1.1.5.1 Submodules

1.1.5.2 dyconnmap.sim_models.makinen module

Notes

Based on the MATLAB code from http://www.cs.bris.ac.uk/home/rafal/phasereset/phase.zip

dyconnmap.sim_models.makinen.**makinen**(*frames*, *epochs*, *fs*, *min_fr*, *max_fr*, *rng=None*)
Makinen

Parameters

- **frames** (*int*) – Number of signal frames per each trial.
- **epochs** (*int*) – Number of simulated trials.
- **fs** (*float*) – Sampling frequency.
- **min_fr** – Minimum frequency of the sinusoid which is being reset.
- **max_fr** – Maximum frequency of the sinusoid which is being reset.
- **rng** (*object or None*) – An object of type numpy.random.RandomState

dyconnmap.sim_models.makinen.**phasereset**(*frames*, *epochs*, *fs*, *min_fr*, *max_fr*, *rng=None*)
Phasereset

Parameters

- **frames** –
- **epochs** –
- **fs** –
- **min_fr** –
- **max_fr** –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

1.1.5.3 Module contents

dyconnmap.sim_models.**makinen**(*frames*, *epochs*, *fs*, *min_fr*, *max_fr*, *rng=None*)
Makinen

Parameters

- **frames** (*int*) – Number of signal frames per each trial.
- **epochs** (*int*) – Number of simulated trials.
- **fs** (*float*) – Sampling frequency.
- **min_fr** – Minimum frequency of the sinusoid which is being reset.
- **max_fr** – Maximum frequency of the sinusoid which is being reset.
- **rng** (*object or None*) – An object of type numpy.random.RandomState

dyconnmap.sim_models.**phasereset**(*frames*, *epochs*, *fs*, *min_fr*, *max_fr*, *rng=None*)
Phasereset

Parameters

- **frames** –
- **epochs** –
- **fs** –
- **min_fr** –
- **max_fr** –

- **rng** (*object or None*) – An object of type numpy.random.RandomState

1.1.6 dyconnmap.ts package

1.1.6.1 Submodules

1.1.6.2 dyconnmap.ts.ci module

Complexity Index

Complexity index (*Janson2004*, *Rapp2007*) computes the *l*-subword complexity (*l*-subword spectrum) of a one-dimensional, symbolic (integer) time series, by finding the number of distinct subwords of length *l*. The total complexity is given by the sum of all subwords of different lengths. The unnormalized measure can be used to compare sequences of equal lengths.

Notes

- This is a direct translation from the *Complexity toolbox available at* http://users.auth.gr/~stdimitr/files/software/complexitiy.rar
- Original author is Stravros Dimitriadis <stidimitriadis@gmail.com>

dyconnmap.ts.ci.**complexity_index**(*x: np.ndarray[np.int32]*, *sub_len: Optional[int] = - 1*, *normalize: Optional[bool] = False*, *iterations: Optional[int] = 100*) → Union[Tuple[numpy.float32, np.ndarray[np.int32]], Tuple[numpy.float32, numpy.float32, np.ndarray[np.int32]]]

Complexity Index

Parameters

- **x** (*array-like, shape(n_samples)*) – Input symbolic time series.
- **sub_len** (*int*) – Maximum subword length. Default is *len(x) - 1*.
- **normalize** (*bool*) – Normalize result. Default is *False*.
- **iterationss** (*int*) – Number of iterations to perform randomization. Default is *100*.

Returns

- **normal_ci** (*float*) – The computed omplexity index after normalization against the randomization procedure.
- **ci** (*float*) – The computed complexity index.
- **spectrum** (*array-like*) – A list of the number of distinct subwords of length 1, up to the size of the input symbolic time series.

1.1.6.3 dyconnmap.ts.cv module

Coefficient of Variation

dyconnmap.ts.cv.**cv**(*x: numpy.ndarray*) → float
Coefficient of Variation

Parameters **x** (*array-like, shape(n_samples)*) – Input time series

Returns **cv** – The computed coefficient of variation.

Return type float

1.1.6.4 dyconnmap.ts.dcorr module

Distance Correlation

Notes

Snippet adapted from: https://gist.github.com/Satra/aa3d19a12b74e9ab7941

dyconnmap.ts.dcorr.**dcorr**(*x: numpy.ndarray*, *y: numpy.ndarray*) → float
Distance Correlation

Parameters

- **x** (*array-like, shape(n_samples)*) – Input time series.
- **y** (*array-like, shape(N)*) – Input time series.

Returns **val** – The computed distance correlation.

Return type float

1.1.6.5 dyconnmap.ts.embed_delay module

When dealing with non-linear time series analysis, it is common to reconstruct hem as time delay vectors in phase space. This new reconstruction, describes the tmporal evolution of a system in a state space; a trajectory of interchanging states. The need for this state space, stems from the fact that the original system may contain latent and unobserved variables that we would like to expose. Thus, we construct m-dimensional phase vectors from τ-time delayed samples (**Takens1981_**):

$$s_n = (s_(n - (m - 1)), s_(n - (m - 2)), \ldots, s_n)$$

This new space, is shown to preserve the dynamics properties of the original phase space. For more on the subject, the interested readers are encouraged to consult the work of Bradley and Kantz (*Bradley2015*).

---

dyconnmap.ts.embed_delay.**embed_delay**(*ts: np.ndarray[np.float32]*, *dim: int*, *tau: int*) → Optional[np.ndarray[np.float32]]
Embed delay

Parameters

- **ts** (*array-like, shape(n_samples)*) – One-dimensional symbolic time series.
- **dim** (*int*) – The embedding dimension.
- **tau** (*int*) – Time delay factor.

Returns **y** – The embedded timeseries.

Return type array-like

1.1.6.6 dyconnmap.ts.entropy module

Entropy

dyconnmap.ts.entropy.**entropy**(*x: np.ndarray[np.float32]*) → float

Entropy

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns **entropy** – The computed entropy.

Return type float

1.1.6.7 dyconnmap.ts.fisher_score module

Fisher Scoring Algorithm

---

dyconnmap.ts.fisher_score.**fisher_score**(*x: numpy.ndarray*, *y: numpy.ndarray*) → numpy.ndarray

Parameters

- **x** –
- **y** –

1.1.6.8 dyconnmap.ts.fisher_z module

Fisher's Z Transformation

---

dyconnmap.ts.fisher_z.**fisher_z**(*data*)

Fisher's z-transformation

For a given dataset p bound to $[0.0, 1.0]$, we can use Fisher's z-transformation to normalize it in an approximately Gaussian distribution.

This transformation is computed as follows:

$$z_p := \frac{1}{2}\ln\left(\frac{1+p}{1-p}\right) = \operatorname{arctanh}(p)$$

Parameters **data** –

dyconnmap.ts.fisher_z.**fisher_z_plv**(*data*)

$$z_j^p = sin^{-1}(2 * PLV_j - 1)$$

Returns

- |
- ——
- .. *[Mormann2005] Mormann, F., Fell, J., Axmacher, N., Weber, B., Lehnertz, K., Elger, C. E., & Fernández, G. (2005). Phase/amplitude reset and theta–gamma interaction in the human medial temporal lobe during a continuous word recognition memory task. Hippocampus, 15(7), 890-900.*

1.1.6.9 dyconnmap.ts.fnn module

False Nearest Neighbors

---

dyconnmap.ts.fnn.**fnn**(*ts: np.ndarray[np.float32]*, *tau: int*, *max_dim: Optional[int] = 20*, *neighbors_reduction: Optional[float] = 0.1*, *rtol: Optional[float] = 15.0*, *atol: Optional[float] = 2.0*) → Optional[int]

False Nearest Neighbors

Notes

The execution stops either when the maxium number of embedding dimensions is reached, or the the number of neighbors is reduced to specific percentage.

Parameters

- **ts** (*array-like, 1d*) –
- **tau** (*int*) – Time-delay parameter.
- **max_dim** (*int*) – Maximum embedding dimension.
- **neighbors_reduction** (*float*) – Maximum percentage of neighbors reduction. Default '0.10' (10%).
- **rtol** (*float*) – First threshold, criterion to identify a false neighbor. (Neighborhood size)
- **atol** (*float*) – Second threshold, criterion to identify a false neighbor.

Returns **min_dimension** – Minimum embedding dimension.

Return type int

1.1.6.10 dyconnmap.ts.icc module

Intra-class Correlation (3, 1)

Notes

- Based on the code available at <https://github.com/ekmolloy/fmri_test-retest>

dyconnmap.ts.icc.**icc_31**(*X: np.ndarray[np.float32]*) → float

ICC (3,1)

Parameters **X** – Input data

Returns **icc** – Intra-class correlation.

Return type float

1.1.6.11 dyconnmap.ts.markov_matrix module

Markov matrix

Generation of markov matrix and some related state transition features.

dyconnmap.ts.markov_matrix.**markov_matrix**(*symts: np.ndarray[np.int32]*, *states_from_length: Optional[bool] = True*) → np.ndarray[np.float32]

Markov Matrix

Markov matrix (also refered as "transition matrix") is a square matrix that tabulates the observed transition probabilities between symbols for a finite Markov Chain. It is a first-order descriptor by which the next symbol depends only on the current symbol (and not on the previous ones); a Markov Chain model.

A transition matrix is formally depicted as:

Given the probability $Pr(j|i)$ of moving between i and j elements, the transition matrix is depicted as:

$$P = \begin{pmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,j} & \dots & P_{1,S} \\ P_{2,1} & P_{2,2} & \dots & P_{2,j} & \dots & P_{2,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i,1} & P_{i,2} & \dots & P_{i,j} & \dots & P_{i,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{S,1} & P_{S,2} & \dots & P_{S,j} & \dots & P_{S,S} \end{pmatrix}$$

Since the transition matrix is row-normalized, so as the total transition probability from state i to all the others must be equal to 1.

For more properties consult, among other links **WolframMathWorld_** and **WikipediaMarkovMatrix_**.

Parameters

- **symts** (*array-like, shape(N)*) – One-dimensional discrete time series.

- **states_from_length** (*bool or int, optional*) – Used to account symbolic time series in which not all the symbols are present. That may happen when for example the symbols are drawn from different distributions. Default *True*, the size of the resulting Markov Matrix is equal to the number of unique symbols present in the time series. If *False*, the size will be the *highest symbolic state + 1*. You may also speficy the highest (inclusive) symbolic state.

Returns **mtx** – The transition matrix. The size depends the parameter *states_from_length*.

Return type matrix

dyconnmap.ts.markov_matrix.**occupancy_time**(*symts: np.ndarray[np.int32]*, *symbol_states: numpy.int32 = None*, *weight: Optional[Union[numpy.float32, np.ndarray[np.float32]]] = None*) → Tuple[numpy.float64, np.ndarray[np.int32]]

Occupancy Time

Parameters

- **symts** –
- **symbol_states** (*int*) – The maximum number of symbols. This is useful to define in case your symbolic timeseries skips some states, in which case would produce a matrix of different size.
- **weight** (*float*) – The weights of the reuslting transition symbols. Default *len(symts)*.

Returns

- *oc*
- *symbols*

dyconnmap.ts.markov_matrix.**transition_rate**(*symts: np.ndarray[np.int32]*, *weight: Optional[Union[numpy.float32, np.ndarray[np.float32]]] = None*) → float

Transition Rate

The total sum of transition between symbols.

Parameters

- **symts** –
- **weight** (*float*) –

1.1.6.12 dyconnmap.ts.ordinal_pattern_similarity module

Ordinal Pattern Similarity

dyconnmap.ts.ordinal_pattern_similarity.**ordinal_pattern_similarity**(*signal1: np.ndarray[np.int32]*, *signal2: np.ndarray[np.int32]*, *m: int*, *tau: int*) → Tuple[float, np.ndarray[np.int32], np.ndarray[np.float32]]

Ordinal Pattern Similarity

Parameters

- **signal1** –
- **signal2** –
- **m** (*int*) – Embedding dimension.
- **tau** (*int*) – Time delay parameter.

Notes

- The results may vary from the original MATLAB script because of the permutations' order.
- The permutations are generated from $[1, dim + 1]$ so there are no occurances of 0.
- The extra $+1$ in the lines .. python: I = sklearn.preprocessing.normalize(I + 1) is in order to avoid :math:`0`s.

Returns

- **dissimilarity** (*float*) – The dissimilarity index as computed from the ordinal patterns.
- **ordinal_patterns** (*array*) – The time series of ordinal patterns for input signals.
- **patterns_distribution** (*array*) – Distribution of the patterns.

1.1.6.13 dyconnmap.ts.permutation_entropy module

Permutation Entropy

---

dyconnmap.ts.permutation_entropy.**permutation_entropy**(*signal: np.ndarray[np.int32]*, *m: int*, *tau: int*) → float

Permutation Entropy

Parameters

- **signal** (*array-like, shape(N)*) – Symblic time series (1D).
- **m** (*int*) – Embedding dimension.
- **tau** (*int*) – Time delay parameter.

Returns

- **pe** (*float*) – Permutation entropy.

- **npe** (*float*) – Normalized permutation entropy.

1.1.6.14 dyconnmap.ts.rr_order_patterns module

Order Reccurence Plot

---

dyconnmap.ts.rr_order_patterns.**rr_order_patterns**(*signal1: np.ndarray[np.int32]*, *signal2: np.ndarray[np.int32]*, *m: int*, *tau: int*) → float

Parameters

- **signal1** –
- **signal2** –
- **m** (*int*) – Embedding dimension.
- **tau** (*int*) – Time delay parameter.

Notes

- The results may vary from the original MATLAB script because of the permutations' order.
- The results may vary from the original MATLAB script because of the order of the indices in the **:python:`numpy.where`**.
- The permutations are generated from $[1, dim + 1]$ so there are no occurances of 0.
- The extra $+1$ in the lines .. python: I = sklearn.preprocessing.normalize(I + 1) is in order to avoid :math:`0`s.

Returns **cstr** – Coupling strength.

Return type float

1.1.6.15 dyconnmap.ts.sampen module

Sample Entropy

Notes

Based on https://nl.mathworks.com/matlabcentral/fileexchange/35784-sample-entropy

---

dyconnmap.ts.sampen.**sample_entropy**(*data: np.ndarray[np.int32]*, *dim: Optional[int] = 2*, *tau: Optional[int] = None*, *r: Optional[float] = None*) → float

Sample Entropy

Parameters

- **data** (*array-like, shape(n_samples)*) – Symbolic time series.
- **dim** (*int*) – Embedding dimension. Default *2*.
- **tau** (*int*) – Delay time for downsampling. Is *None*, *1* is passed.
- **r** (*float*) – Tolerance factor. If *None*, *0.2 * std(data)* is passed.

Returns **SampEn** – Sample entropy.

Return type float

See also:

dyconnmap.ts.embed_ts Embedded timeseries

1.1.6.16 dyconnmap.ts.ste module

Symbolic Transfer Entropy

dyconnmap.ts.ste.**entropy_reduction_rate**(*sym_ts: np.ndarray[np.float32]*) → float

Entropy Reduction Rate

Parameters **sym_ts** (*array-like, shape(n_samples)*) – Input symblic time series.

Returns **entredrate** – The estimated entropy reduction rate.

Return type float

dyconnmap.ts.ste.**symoblic_transfer_entropy**(*x: np.ndarray[np.int32]*, *y: np.ndarray[np.int32]*, *s: Optional[int] = 1*, *delay: Optional[int] = 0*, *verbose: Optional[bool] = False*) → Tuple[float, float, float]

Symbolic Tranfer Entropy

Parameters

- **x** (*array-like, shape(N)*) – Symblic time series (1D).
- **y** (*array-like, shape(N)*) – Symbolic time series (1D).
- **s** (*int*) – Embedding dimension.
- **delay** (*int*) – Time delay parameter
- **verbose** (*boolean*) – Print computation messages.

Returns

- **tent_diff** (*float*) – The difference of the tranfer entropies of the two time series.
- **tentxy** (*float*) – The estimated tranfer entropy of x -> y.
- **tentyx** (*float*) – The estimated tranfer entropy of y -> x.

1.1.6.17 dyconnmap.ts.surrogates module

Surrogate Analysis

dyconnmap.ts.surrogates.**aaft**(*ts: np.ndarray[np.float32]*, *num_surr: Optional[int] = 1*, *rng: Optional[numpy.random.mtrand.RandomState] = None*) → np.ndarray[np.float32]

Amplitude Adjusted Fourier Transform

Parameters

- **ts** –
- **num_surr** –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

dyconnmap.ts.surrogates.**fdr**(*p_values: np.ndarray[np.float32]*, *q: Optional[float] = 0.01*, *method: Optional[str] = 'pdep'*) → Tuple[bool, float]

False Discovery Rate

Parameters

- **p_values** –
- **q** –
- **method** –

dyconnmap.ts.surrogates.**phase_rand**(*data*, *num_surr: Optional[int] = 1*, *rng: Optional[numpy.random.mtrand.RandomState] = None*) → np.ndarray[np.float32]

Phase-randomized suggorates

Parameters

- **data** –
- **num_surr** –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

dyconnmap.ts.surrogates.**surrogate_analysis**(*ts1: np.ndarray[np.float32]*, *ts2: np.ndarray[np.float32]*, *num_surr: Optional[int] = 1000*, *estimator_func: Optional[Callable[[np.ndarray[np.float32], np.ndarray[np.float32]], float]] = None*, *ts1_no_surr: bool = False*, *rng: Optional[numpy.random.mtrand.RandomState] = None*) → Tuple[float, np.ndarray[np.int32], np.ndarray[np.float32], float]

Surrogate Analysis

Parameters

- **ts1** –
- **ts2** –
- **num_surr** (*int*) –
- **estimator_func** (*function*) –
- **ts1_no_surr** (*boolean*) –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

Returns

- **p_val** (*float*)
- *corr_surr*
- *surrogates*
- **r_value** (*float*)

1.1.6.18 dyconnmap.ts.teager_kaiser_energy module

Teager–Kaiser Operator

ref: https://www.c-motion.com/v3dwiki/index.php/Teager_Kaiser_Energy

dyconnmap.ts.teager_kaiser_energy.**teager_kaiser_energy**(*ts: np.ndarray[np.float32]*) → np.ndarray[np.float32]

Teager Kaiser Energy

Parameters **ts** (*array of size [1 x samples]*) –

1.1.6.19 dyconnmap.ts.wald module

Wald test

dyconnmap.ts.wald.**wald**(*x: numpy.ndarray*, *y: numpy.ndarray*) → Tuple[float, float, List[List[int]], List[List[float]]]

Parameters

- **x** –
- **y** –

Returns

- **w** (*float*)
- **r** (*float*)
- **edges** (*array-like*)
- **weights** (*array-like*)

Notes

The input time series will be padded with zeros if needed.

1.1.6.20 Module contents

dyconnmap.ts.**aaft**(*ts: np.ndarray[np.float32]*, *num_surr: Optional[int] = 1*, *rng: Optional[numpy.random.mtrand.RandomState] = None*) → np.ndarray[np.float32]

Amplitude Adjusted Fourier Transform

Parameters

- **ts** –
- **num_surr** –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

dyconnmap.ts.**complexity_index**(*x: np.ndarray[np.int32]*, *sub_len: Optional[int] = - 1*, *normalize: Optional[bool] = False*, *iterations: Optional[int] = 100*) → Union[Tuple[numpy.float32, np.ndarray[np.int32]], Tuple[numpy.float32, numpy.float32, np.ndarray[np.int32]]]

Complexity Index

Parameters

- **x** (*array-like, shape(n_samples)*) – Input symbolic time series.
- **sub_len** (*int*) – Maximum subword length. Default is *len(x) - 1*.
- **normalize** (*bool*) – Normalize result. Default is *False*.
- **iterationss** (*int*) – Number of iterations to perform randomization. Default is *100*.

Returns

- **normal_ci** (*float*) – The computed omplexity index after normalization against the randomization procedure.
- **ci** (*float*) – The computed complexity index.
- **spectrum** (*array-like*) – A list of the number of distinct subwords of length 1, up to the size of the input symbolic time series.

dyconnmap.ts.**cv**(*x: numpy.ndarray*) → float

Coefficient of Variation

Parameters **x** (*array-like, shape(n_samples)*) – Input time series

Returns **cv** – The computed coefficient of variation.

Return type float

dyconnmap.ts.**dcorr**(*x: numpy.ndarray*, *y: numpy.ndarray*) → float

Distance Correlation

Parameters

- **x** (*array-like, shape(n_samples)*) – Input time series.
- **y** (*array-like, shape(N)*) – Input time series.

Returns **val** – The computed distance correlation.

Return type float

dyconnmap.ts.**embed_delay**(*ts: np.ndarray[np.float32]*, *dim: int*, *tau: int*) → Optional[np.ndarray[np.float32]]

Embed delay

Parameters

- **ts** (*array-like, shape(n_samples)*) – One-dimensional symbolic time series.
- **dim** (*int*) – The embedding dimension.
- **tau** (*int*) – Time delay factor.

Returns **y** – The embedded timeseries.

Return type array-like

dyconnmap.ts.**entropy**(*x: np.ndarray[np.float32]*) → float

Entropy

Parameters **x** (*array-like, shape(N)*) – Input symbolic time series.

Returns **entropy** – The computed entropy.

Return type float

dyconnmap.ts.**entropy_reduction_rate**(*sym_ts: np.ndarray[np.float32]*) → float

Entropy Reduction Rate

Parameters **sym_ts** (*array-like, shape(n_samples)*) – Input symblic time series.

Returns **entredrate** – The estimated entropy reduction rate.

Return type float

dyconnmap.ts.**fdr**(*p_values: np.ndarray[np.float32]*, *q: Optional[float] = 0.01*, *method: Optional[str] = 'pdep'*) → Tuple[bool, float]

False Discovery Rate

Parameters

- **p_values** –
- **q** –
- **method** –

dyconnmap.ts.**fisher_score**(*x: numpy.ndarray*, *y: numpy.ndarray*) → numpy.ndarray

Parameters

- **x** –
- **y** –

dyconnmap.ts.**fisher_z**(*data*)

Fisher's z-transformation

For a given dataset p bound to $[0.0, 1.0]$, we can use Fisher's z-transformation to normalize it in an approximately Gaussian distribution.

This transformation is computed as follows:

$$z_p := \frac{1}{2}\ln\left(\frac{1+p}{1-p}\right) = \operatorname{arctanh}(p)$$

Parameters **data** –

dyconnmap.ts.**fisher_z_plv**(*data*)

$$z_j^p = sin^{-1}(2 * PLV_j - 1)$$

Returns

- |
- ——
- .. *[Mormann2005] Mormann, F., Fell, J., Axmacher, N., Weber, B., Lehnertz, K., Elger, C. E., & Fernández, G. (2005). Phase/amplitude reset and theta–gamma interaction in the human medial temporal lobe during a continuous word recognition memory task. Hippocampus, 15(7), 890-900.*

dyconnmap.ts.**fnn**(*ts: np.ndarray[np.float32]*, *tau: int*, *max_dim: Optional[int] = 20*, *neighbors_reduction: Optional[float] = 0.1*, *rtol: Optional[float] = 15.0*, *atol: Optional[float] = 2.0*) → Optional[int]

False Nearest Neighbors

Notes

The execution stops either when the maxium number of embedding dimensions is reached, or the the number of neighbors is reduced to specific percentage.

Parameters

- **ts** (*array-like, 1d*) –
- **tau** (*int*) – Time-delay parameter.
- **max_dim** (*int*) – Maximum embedding dimension.
- **neighbors_reduction** (*float*) – Maximum percentage of neighbors reduction. Default '0.10' (10%).
- **rtol** (*float*) – First threshold, criterion to identify a false neighbor. (Neighborhood size)
- **atol** (*float*) – Second threshold, criterion to identify a false neighbor.

Returns **min_dimension** – Minimum embedding dimension.

Return type int

dyconnmap.ts.**icc_31**(*X: np.ndarray[np.float32]*) → float

ICC (3,1)

Parameters **X** – Input data

Returns **icc** – Intra-class correlation.

Return type float

dyconnmap.ts.**markov_matrix**(*symts: np.ndarray[np.int32]*, *states_from_length: Optional[bool] = True*) → np.ndarray[np.float32]

Markov Matrix

Markov matrix (also refered as "transition matrix") is a square matrix that tabulates the observed transition probabilities between symbols for a finite Markov Chain. It is a first-order descriptor by which the next symbol depends only on the current symbol (and not on the previous ones); a Markov Chain model.

A transition matrix is formally depicted as:

Given the probability $Pr(j|i)$ of moving between i and j elements, the transition matrix is depicted as:

$$P = \begin{pmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,j} & \dots & P_{1,S} \\ P_{2,1} & P_{2,2} & \dots & P_{2,j} & \dots & P_{2,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i,1} & P_{i,2} & \dots & P_{i,j} & \dots & P_{i,S} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{S,1} & P_{S,2} & \dots & P_{S,j} & \dots & P_{S,S} \end{pmatrix}$$

Since the transition matrix is row-normalized, so as the total transition probability from state i to all the others must be equal to 1.

For more properties consult, among other links **WolframMathWorld_** and **WikipediaMarkovMatrix_**.

Parameters

- **symts** (*array-like, shape(N)*) – One-dimensional discrete time series.
- **states_from_length** (*bool or int, optional*) – Used to account symbolic time series in which not all the symbols are present. That may happen when for example the symbols are drawn from different distributions. Default *True*, the size of the resulting Markov Matrix is equal to the number of unique symbols present in the time series. If *False*, the size will be the *highest symbolic state + 1*. You may also speficy the highest (inclusive) symbolic state.

Returns **mtx** – The transition matrix. The size depends the parameter *states_from_length*.

Return type matrix

dyconnmap.ts.**occupancy_time**(*symts: np.ndarray[np.int32]*, *symbol_states: numpy.int32 = None*, *weight: Optional[Union[numpy.float32, np.ndarray[np.float32]]] = None*) → Tuple[numpy.float64, np.ndarray[np.int32]]

Occupancy Time

Parameters

- **symts** –
- **symbol_states** (*int*) – The maximum number of symbols. This is useful to define in case your symbolic timeseries skips some states, in which case would produce a matrix of different size.
- **weight** (*float*) – The weights of the reuslting transition symbols. Default *len(symts)*.

Returns

- *oc*
- *symbols*

dyconnmap.ts.**ordinal_pattern_similarity**(*signal1: np.ndarray[np.int32]*, *signal2: np.ndarray[np.int32]*, *m: int*, *tau: int*) → Tuple[float, np.ndarray[np.int32], np.ndarray[np.float32]]

Ordinal Pattern Similarity

Parameters

- **signal1** –
- **signal2** –
- **m** (*int*) – Embedding dimension.
- **tau** (*int*) – Time delay parameter.

Notes

- The results may vary from the original MATLAB script because of the permutations' order.
- The permutations are generated from $[1, dim + 1]$ so there are no occurances of 0.
- The extra $+1$ in the lines .. python: I = sklearn.preprocessing.normalize(I + 1) is in order to avoid :math:`0`s.

Returns

- **dissimilarity** (*float*) – The dissimilarity index as computed from the ordinal patterns.
- **ordinal_patterns** (*array*) – The time series of ordinal patterns for input signals.
- **patterns_distribution** (*array*) – Distribution of the patterns.

dyconnmap.ts.**permutation_entropy**(*signal: np.ndarray[np.int32]*, *m: int*, *tau: int*) → float

Permutation Entropy

Parameters

- **signal** (*array-like, shape(N)*) – Symblic time series (1D).
- **m** (*int*) – Embedding dimension.
- **tau** (*int*) – Time delay parameter.

Returns

- **pe** (*float*) – Permutation entropy.
- **npe** (*float*) – Normalized permutation entropy.

dyconnmap.ts.**phase_rand**(*data*, *num_surr: Optional[int] = 1*, *rng: Optional[numpy.random.mtrand.RandomState] = None*) → np.ndarray[np.float32]

Phase-randomized suggorates

Parameters

- **data** –
- **num_surr** –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

dyconnmap.ts.**rr_order_patterns**(*signal1: np.ndarray[np.int32]*, *signal2: np.ndarray[np.int32]*, *m: int*, *tau: int*) → float

Parameters

- **signal1** –
- **signal2** –
- **m** (*int*) – Embedding dimension.
- **tau** (*int*) – Time delay parameter.

Notes

- The results may vary from the original MATLAB script because of the permutations' order.
- The results may vary from the original MATLAB script because of the order of the indices in the **:python:`numpy.where`**.
- The permutations are generated from $[1, dim + 1]$ so there are no occurances of 0.
- The extra $+1$ in the lines .. python: I = sklearn.preprocessing.normalize(I + 1) is in order to avoid :math:`0`s.

Returns **cstr** – Coupling strength.

Return type float

dyconnmap.ts.**sample_entropy**(*data: np.ndarray[np.int32]*, *dim: Optional[int] = 2*, *tau: Optional[int] = None*, *r: Optional[float] = None*) → float

Sample Entropy

Parameters

- **data** (*array-like, shape(n_samples)*) – Symbolic time series.
- **dim** (*int*) – Embedding dimension. Default *2*.
- **tau** (*int*) – Delay time for downsampling. Is *None*, *1* is passed.
- **r** (*float*) – Tolerance factor. If *None*, *0.2 * std(data)* is passed.

Returns **SampEn** – Sample entropy.

Return type float

See also:

dyconnmap.ts.embed_ts Embedded timeseries

dyconnmap.ts.**surrogate_analysis**(*ts1: np.ndarray[np.float32]*, *ts2: np.ndarray[np.float32]*, *num_surr: Optional[int] = 1000*, *estimator_func: Optional[Callable[[np.ndarray[np.float32], np.ndarray[np.float32]], float]] = None*, *ts1_no_surr: bool = False*, *rng: Optional[numpy.random.mtrand.RandomState] = None*) → Tuple[float, np.ndarray[np.int32], np.ndarray[np.float32], float]

Surrogate Analysis

Parameters

- **ts1** –
- **ts2** –
- **num_surr** (*int*) –
- **estimator_func** (*function*) –
- **ts1_no_surr** (*boolean*) –
- **rng** (*object or None*) – An object of type numpy.random.RandomState

Returns

- **p_val** (*float*)
- *corr_surr*

- *surrogates*
- **r_value** (*float*)

dyconnmap.ts.**symoblic_transfer_entropy**(*x: np.ndarray[np.int32]*, *y: np.ndarray[np.int32]*, *s: Optional[int] = 1*, *delay: Optional[int] = 0*, *verbose: Optional[bool] = False*) → Tuple[float, float, float]

Symbolic Tranfer Entropy

Parameters

- **x** (*array-like, shape(N)*) – Symblic time series (1D).
- **y** (*array-like, shape(N)*) – Symbolic time series (1D).
- **s** (*int*) – Embedding dimension.
- **delay** (*int*) – Time delay parameter
- **verbose** (*boolean*) – Print computation messages.

Returns

- **tent_diff** (*float*) – The difference of the tranfer entropies of the two time series.
- **tentxy** (*float*) – The estimated tranfer entropy of x -> y.
- **tentyx** (*float*) – The estimated tranfer entropy of y -> x.

dyconnmap.ts.**teager_kaiser_energy**(*ts: np.ndarray[np.float32]*) → np.ndarray[np.float32]

Teager Kaiser Energy

Parameters **ts** (*array of size [1 x samples]*) –

dyconnmap.ts.**transition_rate**(*symts: np.ndarray[np.int32]*, *weight: Optional[Union[numpy.float32, np.ndarray[np.float32]]] = None*) → float

Transition Rate

The total sum of transition between symbols.

Parameters

- **symts** –
- **weight** (*float*) –

dyconnmap.ts.**wald**(*x: numpy.ndarray*, *y: numpy.ndarray*) → Tuple[float, float, List[List[int]], List[List[float]]]

Parameters

- **x** –
- **y** –

Returns

- **w** (*float*)
- **r** (*float*)
- **edges** (*array-like*)
- **weights** (*array-like*)

Notes

The input time series will be padded with zeros if needed.

1.2 Submodules

1.3 dyconnmap.analytic_signal module

Analytic Signal

For a time series $x(t)$, filtered with a passband N; first we compute its analytic representation (*Cohen1995* , *Freeman2007*):

$$u_j(t) = \frac{1}{\pi}\text{PV}\int_{+\infty}^{-\infty}\frac{V_j(t')}{t-t'}dt'$$

where PV signifies the Cauchy Principal Value. The above equation results into a complex time-series $V_j(t)$, with a real part $v_j(t)$ (the original neuroelectric time-series) and an imaginary part $u_j(t)$, both as functions of time. Where j the EEG recording electrode id.

From these parts, we are capable to determine the Instantaneous Amplitude:

$$A_j(t) = \sqrt{v_j^2(t) + u_j^2(t)}$$

and its Instantaneous Phase counterpart, from:

$$\phi_j(t) = atan\frac{u_j(t)}{v_j(t)}$$

The values in $\phi_j(t)$ are originally bound to $[-\pi, \pi]$, however we employed an *unwrap* transformation (a phase correction algorithm) in order to eliminate the discontinuities (**Dimitriadis2010_**, *Freeman2002*).

---

dyconnmap.analytic_signal.**analytic_signal**(*signal*, *fb=None*, *fs=None*, *order=3*)

Passband filtering and Hilbert transformation

Parameters

- **signal** (*real array-like, shape(n_rois, n_samples)*) – Input signal
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*int, optional.*) – Sampling frequency.
- **order** (*int, optional*) – The Filter order. Default *3*.

Returns

- **hilberted_signal** (*complex array-like, shape(n_rois, n_samples)*) – The Hilbert representation of the input signal.
- **unwrapped_phase** (*real array-like, shape(n_rois, n_samples)*) – The unwrapped phase of the Hilbert representation.
- **filtered_signal** (*real array-like, shape(n_rois, n_samples)*) – The input signal, filtered within the given frequencies. This is returned only if *fb* and *fs* are passed.

Notes

Internally, we use SciPy's Butterworth implementation (*scipy.signal.butter*) and the two-pass filter *scipy.signal.filtfilt* to achieve results identical to MATLAB.

1.4 dyconnmap.bands module

Commonly used band frequencies

For your convenience we have predefined some widely adopted brain rhythms. You can access them with

```
from dyconnmap.bands import *
print(bands['alpha'])
```

brainwave	frequency (Hz)	variable/index
	[1.0, 4.0]	bands['delta']
	[4.0, 8.0]	bands['theta']
1	[7.0, 10.0]	bands['alpha1']
2	[10.0, 13.0]	bands['alpha2']
	[7.0, 13.0]	bands['alpha']
	[8.0, 13.0]	band['mu']
	[13.0, 25.0]	bands['beta']
	[25.0, 40.0]	bands['gamma']

1.5 dyconnmap.sliding_window module

Sliding Window

dyconnmap.sliding_window.**sliding_window**(*data*, *estimator_instance*, *window_length=25*, *step=1*, *pairs=None*)

dyconnmap.sliding_window.**sliding_window_indx**(*data*, *window_length*, *overlap=0.75*, *pairs=None*)

Compute the indices and pairs using a sliding window.

Slide a window over `data`, and return the indices and offsets.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **window_length** (*int*) – Number of samples to be used in the computation of the connectivity.
- **overlap** (*float*) – Percentage of the `window_length` by which the window will overlap when sliding forward.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **indices** – Indices of pairs.

Return type array - like, shape(n_windows, start_offset, end_offset, n_channels, n_channels)

1.6 dyconnmap.tvfcgs module

Time-Varying Functional Connectivity Graphs

Time-varying functional connectivity graphs (TVFCGs) (**Dimitriadis2010_**, *Falani2008*) introduce the idea of processing overlapping segments of neuroelectric signals by defining a frequency-dependent time window in which the synchronization is estimated; and then tabulating the results as adjacency matrices. These matrices have a natural graph-based representation called "functional connectivity graphs" (FCGs).

An important aspect of the TVFCGs is the "cycle-criterion" (CC) (*Cohen2008*). It regulates the amount of the oscillation cycles that will be considered in measuring the phase synchrony. In the original proposal $CC = 2.0$ was introduced, resulting into a time-window with width twice the lower period. TVFCGs on the other, consider the given lower frequency that correspond to the possibly synchronized oscillations of each brain rhythm and the sampling frequency. This newly defined frequency-depedent time-window is sliding over the time series and the network connectivity is estimated. The overlapping is determined by an arbitrary step parameter.

Given a multi-channel recording data matrix $X^{m \times n}$ of size $m \times n$ (with m channels, and n samples), a frequency range with F_{up} and F_{lo} the upper and lower limits, fs the sampling frequency, $step$ and CC, the computation of these graphs proceeds as follows:

Firstly, based on the CC and the specified frequency range (F_{lo} and fs) the window size calculated:

$$w_{len} = \frac{CC}{F_{lo}} fs$$

Then, this window is moving per $step$ samples and the average synchronization is computed (between the channels, in a pairwise manner) resulting into $\frac{n}{step}$ adjacency matrices of size $n \times n$.

---

dyconnmap.tvfcgs.**tvfcg**(*data*, *estimator_instance*, *fb*, *fs*, *cc=2.0*, *step=5.0*, *pairs=None*)

Time-Varying Functional Connectivity Graphs

The TVFCGs are computed from the input `data` by employing the given synchronization estimator (`estimator_instance`).

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **estimator_instance** (*object*) – An object of type *dyconnmap.fc.Estimator*.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – The amount of samples the window will move/slide over the time series.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **fcgs** – The computed FCGs.

Return type array-like, shape(n_windows, n_channels, n_channels)

dyconnmap.tvfcgs.**tvfcg_cfc**(*data*, *estimator_instance*, *fb_lo*, *fb_hi*, *fs=128*, *cc=2.0*, *step=5*, *pairs=None*)

Time-Varying Functional Connectivity Graphs (for Cross frequency Coupling)

The TVFCGs are computed from the input `data` by employing the given cross frequency coupling synchronization estimator (`estimator_instance`).

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **estimator_instance** (*object*) – An object of type *dyconnmap.fc.Estimator*.
- **fb_lo** (*list of length 2*) – The low and high frequencies.
- **fb_hi** (*list of length 2*) – The low and high frequencies.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – The amount of samples the window will move/slide over the time series.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **fcgs** – The computed Cross-Frequency FCGs.

Return type array-like, shape(n_windows, n_channels, n_channels)

Notes

Not all available estimators in the *dyconnmap.fc* are valid for estimating cross frequency coupling.

dyconnmap.tvfcgs.**tvfcg_compute_windows**(*data*, *fb_lo*, *fs*, *cc*, *step*)

Compute TVFCGs Sliding Windows

A helper function that computes the size and number of sliding windows given the parameters.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** –
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – Stepping.

Returns

- **windows** (*int*) – The total number of sliding windows.
- **window_length** (*int*) – The length of a sliding window; number of samples used to estimated the connectivity.

dyconnmap.tvfcgs.**tvfcg_ts**(*ts*, *fb*, *fs=128*, *cc=2.0*, *step=5*, *pairs=None*, *avg_func=<function mean>*)
Time-Varying Function Connectivity Graphs (from time series)

This implementation operates directly on the given estimated synchronization time series (`ts`) and the mean value inside the window is computed.

Parameters

- **ts** (*array-like, shape(n_channels, n_samples)*) – Multichannel synchronization time series.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – The amount of samples the window will move/slide over the time series.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **fcgs** – The computed FCGs.

Return type array-like

1.7 Module contents

dyconnmap.**analytic_signal**(*signal*, *fb=None*, *fs=None*, *order=3*)
Passband filtering and Hilbert transformation

Parameters

- **signal** (*real array-like, shape(n_rois, n_samples)*) – Input signal
- **fb** (*list of length 2, optional*) – The low and high frequencies.
- **fs** (*int, optional.*) – Sampling frequency.
- **order** (*int, optional*) – The Filter order. Default *3*.

Returns

- **hilberted_signal** (*complex array-like, shape(n_rois, n_samples)*) – The Hilbert representation of the input signal.
- **unwrapped_phase** (*real array-like, shape(n_rois, n_samples)*) – The unwrapped phase of the Hilbert representation.
- **filtered_signal** (*real array-like, shape(n_rois, n_samples)*) – The input signal, filtered within the given frequencies. This is returned only if *fb* and *fs* are passed.

Notes

Internally, we use SciPy's Butterworth implementation (*scipy.signal.butter*) and the two-pass filter *scipy.signal.filtfilt* to achieve results identical to MATLAB.

dyconnmap.**sliding_window**(*data*, *estimator_instance*, *window_length=25*, *step=1*, *pairs=None*)

dyconnmap.**sliding_window_indx**(*data*, *window_length*, *overlap=0.75*, *pairs=None*)
Compute the indices and pairs using a sliding window.

Slide a window over `data`, and return the indices and offsets.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **window_length** (*int*) – Number of samples to be used in the computation of the connectivity.
- **overlap** (*float*) – Percentage of the `window_length` by which the window will overlap when sliding forward.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **indices** – Indices of pairs.

Return type array - like, shape(n_windows, start_offset, end_offset, n_channels, n_channels)

dyconnmap.**tvfcg**(*data*, *estimator_instance*, *fb*, *fs*, *cc=2.0*, *step=5.0*, *pairs=None*)
Time-Varying Functional Connectivity Graphs

The TVFCGs are computed from the input `data` by employing the given synchronization estimator (`estimator_instance`).

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **estimator_instance** (*object*) – An object of type *dyconnmap.fc.Estimator*.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – The amount of samples the window will move/slide over the time series.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **fcgs** – The computed FCGs.

Return type array-like, shape(n_windows, n_channels, n_channels)

dyconnmap.**tvfcg_cfc**(*data*, *estimator_instance*, *fb_lo*, *fb_hi*, *fs=128*, *cc=2.0*, *step=5*, *pairs=None*)
Time-Varying Functional Connectivity Graphs (for Cross frequency Coupling)

The TVFCGs are computed from the input `data` by employing the given cross frequency coupling synchronization estimator (`estimator_instance`).

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **estimator_instance** (*object*) – An object of type *dyconnmap.fc.Estimator*.
- **fb_lo** (*list of length 2*) – The low and high frequencies.
- **fb_hi** (*list of length 2*) – The low and high frequencies.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – The amount of samples the window will move/slide over the time series.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **fcgs** – The computed Cross-Frequency FCGs.

Return type array-like, shape(n_windows, n_channels, n_channels)

Notes

Not all available estimators in the *dyconnmap.fc* are valid for estimating cross frequency coupling.

dyconnmap.**tvfcg_compute_windows**(*data*, *fb_lo*, *fs*, *cc*, *step*)

Compute TVFCGs Sliding Windows

A helper function that computes the size and number of sliding windows given the parameters.

Parameters

- **data** (*array-like, shape(n_channels, n_samples)*) – Multichannel recording data.
- **fb_lo** –
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – Stepping.

Returns

- **windows** (*int*) – The total number of sliding windows.
- **window_length** (*int*) – The length of a sliding window; number of samples used to estimated the connectivity.

dyconnmap.**tvfcg_ts**(*ts*, *fb*, *fs=128*, *cc=2.0*, *step=5*, *pairs=None*, *avg_func=<function mean>*)

Time-Varying Function Connectivity Graphs (from time series)

This implementation operates directly on the given estimated synchronization time series (`ts`) and the mean value inside the window is computed.

Parameters

- **ts** (*array-like, shape(n_channels, n_samples)*) – Multichannel synchronization time series.
- **fb** (*list of length 2*) – The lower and upper frequency.
- **fs** (*float*) – Sampling frequency.
- **cc** (*float*) – Cycle criterion.
- **step** (*int*) – The amount of samples the window will move/slide over the time series.
- **pairs** (array-like or *None*) –
  - If an *array-like* is given, notice that each element is a tuple of length two.
  - If *None* is passed, complete connectivity will be assumed.

Returns **fcgs** – The computed FCGs.

Return type array-like

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Dimitriadis2019] Dimitriadis, S. I., López, M. E., Maestu, F., & Pereda, E. (2019). Modeling the Switching behavior of Functional Connectivity Microstates (FCstates) as a Novel Biomarker for Mild Cognitive Impairment. Frontiers in Neuroscience, 13.

[Basset2011] Bassett, D. S., Wymbs, N. F., Porter, M. A., Mucha, P. J., Carlson, J. M., & Grafton, S. T. (2011). Dynamic reconfiguration of human brain networks during learning. Proceedings of the National Academy of Sciences, 108(18), 7641-7646.

[Dimitriadis2019] Dimitriadis, S. I., López, M. E., Maestu, F., & Pereda, E. (2019). Modeling the Switching behavior of Functional Connectivity Microstates (FCstates) as a Novel Biomarker for Mild Cognitive Impairment. Frontiers in Neuroscience, 13.

[Fritzke1995] Fritzke, B. (1995). A growing neural gas network learns topologies. In Advances in neural information processing systems (pp. 625-632).

[Strickert2003] Strickert, M., & Hammer, B. (2003, September). Neural gas for sequences. In Proceedings of the Workshop on Self-Organizing Maps (WSOM'03) (pp. 53-57).

[Martinetz1991] Martinetz, T., Schulten, K., et al. A "neural-gas" network learns topologies. University of Illinois at Urbana-Champaign, 1991.

[Laskaris2004] Laskaris, N. A., Fotopoulos, S., & Ioannides, A. A. (2004). Mining information from event-related recordings. Signal Processing Magazine, IEEE, 21(3), 66-77.

[Hammer2007] Hammer, B., & Hasenfuss, A. (2007, September). Relational neural gas. In Annual Conference on Artificial Intelligence (pp. 190-204). Springer, Berlin, Heidelberg.

[Hasenfuss2008] Hasenfuss, A., Hammer, B., & Rossi, F. (2008, July). Patch Relational Neural Gas–Clustering of Huge Dissimilarity Datasets. In IAPR Workshop on Artificial Neural Networks in Pattern Recognition (pp. 1-12). Springer, Berlin, Heidelberg.

[Martinetz1991] Martinetz, T., Schulten, K., et al. A "neural-gas" network learns topologies. University of Illinois at Urbana-Champaign, 1991.

[RayTuri1999] Ray, S., & Turi, R. H. (1999, December). Determination of number of clusters in k-means clustering and application in colour image segmentation. In Proceedings of the 4th international conference on advances in pattern recognition and digital techniques (pp. 137-143).

[Davies1979] Davies, D. L., & Bouldin, D. W. (1979). A cluster separation measure. IEEE transactions on pattern analysis and machine intelligence, (2), 224-227.

[Bruns2000] Bruns, A., Eckhorn, R., Jokeit, H., & Ebner, A. (2000). Amplitude envelope correlation detects coupling among incoherent brain signals. Neuroreport, 11(7), 1509-1514.

[Penny2008] Penny, W. D., Duzel, E., Miller, K. J., & Ojemann, J. G. (2008). Testing for nested oscillation. Journal of neuroscience methods, 174(1), 50-61.

[Friston1996] Friston, K. J. (1997). Another neural code?. Neuroimage, 5(3), 213-220.

[Darvas2009] Darvas, F., Ojemann, J. G., & Sorensen, L. B. (2009). Bi-phase locking—a tool for probing non-linear interaction in the human brain. NeuroImage, 46(1), 123-132.

[Nolte2004] Nolte, G., Bai, O., Wheaton, L., Mari, Z., Vorbach, S., & Hallett, M. (2004). Identifying true brain interaction from EEG data using the imaginary part of coherency. Clinical neurophysiology, 115(10), 2292-2307.

[Thatcher2005] Thatcher, R. W., North, D., & Biver, C. (2005). EEG and intelligence: relations between EEG coherence, EEG phase delay and power. Clinical neurophysiology, 116(9), 2129-2141.

[Vinck2011] Vinck, M., Oostenveld, R., van Wingerden, M., Battaglia, F., & Pennartz, C. M. (2011). An improved index of phase-synchronization for electrophysiological data in the presence of volume-conduction, noise and sample-size bias. Neuroimage, 55(4), 1548-1565.

[Stam2012] Stam, C. J., & van Straaten, E. C. (2012). Go with the flow: use of a directed phase lag index (dPLI) to characterize patterns of phase relations in a large-scale model of brain dynamics. Neuroimage, 62(3), 1415-1428.

[Bruns2004] Bruns, A., & Eckhorn, R. (2004). Task-related coupling from high-to low-frequency signals among visual cortical areas in human subdural recordings. International Journal of Psychophysiology, 51(2), 97-116.

[Penny2008] Penny, W. D., Duzel, E., Miller, K. J., & Ojemann, J. G. (2008). Testing for nested oscillation. Journal of neuroscience methods, 174(1), 50-61. Chicago

[Penny2006] Penny, W. D., Friston, K. J., Ashburner, J. T., Kiebel, S. J., & Nichols, T. E. (Eds.). (2011). Statistical parametric mapping: the analysis of functional brain images. Academic press.

[Penny2008] Penny, W. D., Duzel, E., Miller, K. J., & Ojemann, J. G. (2008). Testing for nested oscillation. Journal of neuroscience methods, 174(1), 50-61.

[Sadaghiani2012] Sadaghiani, S., Scheeringa, R., Lehongre, K., Morillon, B., Giraud, A. L., D'Esposito, M., & Kleinschmidt, A. (2012). Alpha-band phase synchrony is related to activity in the fronto-parietal adaptive control network. The Journal of Neuroscience, 32(41), 14305-14310.

[Vinck2011] Vinck, M., Oostenveld, R., van Wingerden, M., Battaglia, F., & Pennartz, C. M. (2011). An improved index of phase-synchronization for electrophysiological data in the presence of volume-conduction, noise and sample-size bias. Neuroimage, 55(4), 1548-1565.

[Penny2008] Penny, W. D., Duzel, E., Miller, K. J., & Ojemann, J. G. (2008). Testing for nested oscillation. Journal of neuroscience methods, 174(1), 50-61. Chicago

[Bruns2004] Bruns, A., & Eckhorn, R. (2004). Task-related coupling from high-to low-frequency signals among visual cortical areas in human subdural recordings. International Journal of Psychophysiology, 51(2), 97-116.

[Hipp2012] Hipp, J. F., Hawellek, D. J., Corbetta, M., Siegel, M., & Engel, A. K. (2012). Large-scale cortical correlation structure of spontaneous oscillatory activity. Nature neuroscience, 15(6), 884-890.

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[Lachaux1998] Lachaux, J., Rodriguez, E., Martinerie, J., Varela, F., & others,. (1999). Measuring phase synchrony in brain signals. Human Brain Mapping, 8(4), 194-208.

[Tass1998] Tass, P., Rosenblum, M. G., Weule, J., Kurths, J., Pikovsky, A., Volkmann, J., ... & Freund, H. J. (1998). Detection of n: m phase locking from noisy data: application to magnetoencephalography. Physical review letters, 81(15), 3291.

[Vinck2011] Vinck, M., Oostenveld, R., van Wingerden, M., Battaglia, F., & Pennartz, C. M. (2011). An improved index of phase-synchronization for electrophysiological data in the presence of volume-conduction, noise and sample-size bias. Neuroimage, 55(4), 1548-1565.

[Hammond2013] Hammond, D. K., Gur, Y., & Johnson, C. R. (2013, December). Graph diffusion distance: A difference measure for weighted graphs based on the graph Laplacian exponential kernel. In Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE (pp. 419-422). IEEE.

[Ipsen2004] Ipsen, M. (2004). Evolutionary reconstruction of networks. In Function and Regulation of Cellular Systems (pp. 241-249). Birkhäuser, Basel.

[Donnat2018] Donnat, C., & Holmes, S. (2018). Tracking Network Dynamics: a review of distances and similarity metrics. arXiv preprint arXiv:1801.07351.

[Fred2005] Fred, A. L., & Jain, A. K. (2005). Combining multiple clusterings using evidence accumulation. IEEE transactions on pattern analysis and machine intelligence, 27(6), 835-850.

[Strehl2002] Strehl, A., & Ghosh, J. (2002). Cluster ensembles—a knowledge reuse framework for combining multiple partitions. Journal of machine learning research, 3(Dec), 583-617.

[Guillon2016] Guillon, J., Attal, Y., Colliot, O., La Corte, V., Dubois, B., Schwartz, D., … & Fallani, F. D. V. (2017). Loss of brain inter-frequency hubs in Alzheimer's disease. Scientific reports, 7(1), 10879.

[Wilson2008] Wilson, R. C., & Zhu, P. (2008). A study of graph spectra for comparing graphs and trees. Pattern Recognition, 41(9), 2833-2841.

[Jakobson2000] Jakobson, D., & Rivin, I. (2000). Extremal metrics on graphs I. arXiv preprint math/0001169.

[Pincombe2007] Pincombe, B. (2007). Detecting changes in time series of network graphs using minimum mean squared error and cumulative summation. ANZIAM Journal, 48, 450-473.

[Alvarez2006] Alvarez-Hamelin, J. I., Dall'Asta, L., Barrat, A., & Vespignani, A. (2006). Large scale networks fingerprinting and visualization using the k-core decomposition. In Advances in neural information processing systems (pp. 41-50).

[Hagman2008] Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C. J., Wedeen, V. J., & Sporns, O. (2008). Mapping the structural core of human cerebral cortex. PLoS biology, 6(7), e159.

[Basset2009] Bassett, D. S., Bullmore, E. T., Meyer-Lindenberg, A., Apud, J. A., Weinberger, D. R., & Coppola, R. (2009). Cognitive fitness of cost-efficient brain functional networks. Proceedings of the National Academy of Sciences, 106(28), 11747-11752.

[Dimitriadis2017a] Dimitriadis, S. I., Salis, C., Tarnanas, I., & Linden, D. E. (2017). Topological Filtering of Dynamic Functional Brain Networks Unfolds Informative Chronnectomics: A Novel Data-Driven Thresholding Scheme Based on Orthogonal Minimal Spanning Trees (OMSTs). Frontiers in neuroinformatics, 11.

[Dimitriadis2017n] Dimitriadis, S. I., Antonakakis, M., Simos, P., Fletcher, J. M., & Papanicolaou, A. C. (2017). Data-driven Topological Filtering based on Orthogonal Minimal Spanning Trees: Application to Multi-Group MEG Resting-State Connectivity. Brain Connectivity, (ja).

[Basset2009] Bassett, D. S., Bullmore, E. T., Meyer-Lindenberg, A., Apud, J. A., Weinberger, D. R., & Coppola, R. (2009). Cognitive fitness of cost-efficient brain functional networks. Proceedings of the National Academy of Sciences, 106(28), 11747-11752.

[Dimitriadis2010] Dimitriadis, S. I., Laskaris, N. A., Tsirka, V., Vourkas, M., Micheloyannis, S., & Fotopoulos, S. (2010). Tracking brain dynamics via time-dependent network analysis. Journal of neuroscience methods, 193(1), 145-155.

[Meilla2007] Meilă, M. (2007). Comparing clusterings—an information based distance. Journal of multivariate analysis, 98(5), 873-895.

[Dimitriadis2009] Dimitriadis, S. I., Laskaris, N. A., Del Rio-Portilla, Y., & Koudounis, G. C. (2009). Characterizing dynamic functional connectivity across sleep stages from EEG. Brain topography, 22(2), 119-133.

[Dimitriadis2012] Dimitriadis, S. I., Laskaris, N. A., Michael Vourkas, V. T., & Micheloyannis, S. (2012). An EEG study of brain connectivity dynamics at the resting state. Nonlinear Dynamics-Psychology and Life Sciences, 16(1), 5.

[Alvarez2006] Alvarez-Hamelin, J. I., Dall'Asta, L., Barrat, A., & Vespignani, A. (2006). Large scale networks fingerprinting and visualization using the k-core decomposition. In Advances in neural information processing systems (pp. 41-50).

[Hagman2008] Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C. J., Wedeen, V. J., & Sporns, O. (2008). Mapping the structural core of human cerebral cortex. PLoS biology, 6(7), e159.

[Basset2009] Bassett, D. S., Bullmore, E. T., Meyer-Lindenberg, A., Apud, J. A., Weinberger, D. R., & Coppola, R. (2009). Cognitive fitness of cost-efficient brain functional networks. Proceedings of the National Academy of Sciences, 106(28), 11747-11752.

[Dimitriadis2017a] Dimitriadis, S. I., Salis, C., Tarnanas, I., & Linden, D. E. (2017). Topological Filtering of Dynamic Functional Brain Networks Unfolds Informative Chronnectomics: A Novel Data-Driven Thresholding Scheme Based on Orthogonal Minimal Spanning Trees (OMSTs). Frontiers in neuroinformatics, 11.

[Dimitriadis2017n] Dimitriadis, S. I., Antonakakis, M., Simos, P., Fletcher, J. M., & Papanicolaou, A. C. (2017). Data-driven Topological Filtering based on Orthogonal Minimal Spanning Trees: Application to Multi-Group MEG Resting-State Connectivity. Brain Connectivity, (ja).

[Basset2009] Bassett, D. S., Bullmore, E. T., Meyer-Lindenberg, A., Apud, J. A., Weinberger, D. R., & Coppola, R. (2009). Cognitive fitness of cost-efficient brain functional networks. Proceedings of the National Academy of Sciences, 106(28), 11747-11752.

[Dimitriadis2010] Dimitriadis, S. I., Laskaris, N. A., Tsirka, V., Vourkas, M., Micheloyannis, S., & Fotopoulos, S. (2010). Tracking brain dynamics via time-dependent network analysis. Journal of neuroscience methods, 193(1), 145-155.

[Yeung] Yeung, N., Bogacz, R., Holroyd, C. B., Nieuwenhuis, S., & Cohen, J. D. (2007). Theta phase resetting and the error-related negativity. Psychophysiology, 44(1), 39-49.

[Makinen] Mäkinen, V., Tiitinen, H., & May, P. (2005). Auditory event-related responses are generated independently of ongoing brain activity. Neuroimage, 24(4), 961-968.

[Janson2004] Janson, S., Lonardi, S., & Szpankowski, W. (2004). On average sequence complexity. Theoretical Computer Science, 326(1-3), 213-227.

[Rapp2007] Rapp, P. E. (2007). Quantitative characterization of animal behavior following blast exposure. Cognitive neurodynamics, 1(4), 287-293.

[Taken1981] Takens, F. (1981). Detecting strange attractors in turbulence. Lecture notes in mathematics, 898(1), 366-381.

[Bradley2015] Bradley, E., & Kantz, H. (2015). Nonlinear time-series analysis revisited. Chaos: An Interdisciplinary Journal of Nonlinear Science, 25(9), 097610.

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[Kennel1992] Kennel, M. B., Brown, R., & Abarbanel, H. D. (1992). Determining embedding dimension for phase-space reconstruction using a geometrical construction. Physical review A, 45(6), 3403.

[Abarbane2012] Abarbanel, H. (2012). Analysis of observed chaotic data. Springer Science & Business Media.

[McGraw1996] McGraw, K. O., & Wong, S. P. (1996). Forming inferences about some intraclass correlation coefficients. Psychological methods, 1(1), 30.

[Birn2013] Birn, R. M., Molloy, E. K., Patriat, R., Parker, T., Meier, T. B., Kirk, G. R., … & Prabhakaran, V. (2013). The effect of scan length on the reliability of resting-state fMRI connectivity estimates. Neuroimage, 83, 550-558.

[WolframMathWorld] http://mathworld.wolfram.com/StochasticMatrix.html

[WikipediaMarkovMatrix] https://en.wikipedia.org/wiki/Stochastic_matrix

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[Stam2007] Stam, C. J., Nolte, G., & Daffertshofer, A. (2007). Phase lag index: assessment of functional connectivity from multi channel EEG and MEG with diminished bias from common sources. Human brain mapping, 28(11), 1178-1193.

[Hardmeier2014] Hardmeier, M., Hatz, F., Bousleiman, H., Schindler, C., Stam, C. J., & Fuhr, P. (2014). Reproducibility of functional connectivity and graph measures based on the phase lag index (PLI) and weighted phase lag index (wPLI) derived from high resolution EEG. PloS one, 9(10), e108648.

[WolframMathWorld] http://mathworld.wolfram.com/StochasticMatrix.html

[WikipediaMarkovMatrix] https://en.wikipedia.org/wiki/Stochastic_matrix

[Cohen1995] Cohen, L. (1995). Time-frequency analysis (Vol. 1, No. 995,299). Prentice hall.

[Freeman2007] Walter J. Freeman (2007) Hilbert transform for brain waves. Scholarpedia, 2(1):1338.

[Dimitriadis2010] Dimitriadis, S. I., Laskaris, N. A., Tsirka, V., Vourkas, M., Micheloyannis, S., & Fotopoulos, S. (2010). Tracking brain dynamics via time-dependent network analysis. Journal of neuroscience methods, 193(1), 145-155.

[Freeman2002] Freeman, W. J., & Rogers, L. J. (2002). Fine temporal resolution of analytic phase reveals episodic synchronization by state transitions in gamma EEGs. Journal of neurophysiology, 87(2), 937-945.

[Butter1930] Butterworth, S. (1930). On the theory of filter amplifiers. Wireless Engineer, 7(6), 536-541.

[Cohen2008] Cohen, M. X. (2008). Assessing transient cross-frequency coupling in EEG data. Journal of neuroscience methods, 168(2), 494-499.

[Dimitriadis2010] Dimitriadis, S. I., Laskaris, N. A., Tsirka, V., Vourkas, M., Micheloyannis, S., & Fotopoulos, S. (2010). Tracking brain dynamics via time-dependent network analysis. Journal of neuroscience methods, 193(1), 145-155.

[Falani2008] Fallani, F. D. V., Latora, V., Astolfi, L., Cincotti, F., Mattia, D., Marciani, M. G., … & Babiloni, F. (2008). Persistent patterns of interconnection in time-varying cortical networks estimated from high-resolution EEG recordings in humans during a simple motor act. Journal of Physics A: Mathematical and Theoretical, 41(22), 224014.

PYTHON MODULE INDEX

d

INDEX

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W