brainconn

Release 0.0.1

Oct 27, 2018

Contents

brainconn is a Python package for graph theoretic analysis of neuroimaging data.

CHAPTER 1

Contents

1.1 What is graph theory?

Graph theory refers to methods for measures graphs.

1.2 What is brainconn?

brainconn is a Python package for the calculation of graph theoretic metrics from neuroimaging data.

1.3 API Reference

1.3.1 `brainconn.centrality`: Centrality

Metrics which identify the most important nodes in graphs.

brainconn.centrality	Metrics which identify the most important nodes in graphs.
brainconn.centrality.betweenness_bin(G)	Node betweenness centrality is the fraction of all shortest paths in the network that contain a given node.
brainconn.centrality.betweenness_wei(G)	Node betweenness centrality is the fraction of all shortest paths in the network that contain a given node.
brainconn.centrality.diversity_coef_sign(W, ci)	The Shannon entropy-based diversity coefficient measures the diversity of intermodular connections of individual nodes and ranges from 0 to 1.
brainconn.centrality.edge_betweenness_bin(G)	Edge betweenness centrality is the fraction of all shortest paths in the network that contain a given edge.

Continued on next page

Table 1 – continued from previous page

brainconn.centrality.edge_betweenness_wei(G)	Edge betweenness centrality is the fraction of all shortest paths in the network that contain a given edge.
brainconn.centrality.eigenvector_centrality_und(CIJ)	Eigenector centrality is a self-referential measure of centrality: nodes have high eigenvector centrality if they connect to other nodes that have high eigenvector centrality.
brainconn.centrality.erange(CIJ)	Shortcuts are central edges which significantly reduce the characteristic path length in the network.
brainconn.centrality.flow_coef_bd(CIJ)	Computes the flow coefficient for each node and averaged over the network, as described in Honey et al.
brainconn.centrality.gateway_coef_sign(W, ci)	The gateway coefficient is a variant of participation coefficient.
brainconn.centrality.kcoreness_centrality_bd(CIJ)	The k-core is the largest subgraph comprising nodes of degree at least k.
brainconn.centrality.kcoreness_centrality_bu(CIJ)	The k-core is the largest subgraph comprising nodes of degree at least k.
brainconn.centrality.module_degree_zscore(W, ci)	The within-module degree z-score is a within-module version of degree centrality.
brainconn.centrality.pagerank_centrality(A, d)	The PageRank centrality is a variant of eigenvector centrality.
brainconn.centrality.participation_coef(W, ci)	Participation coefficient is a measure of diversity of intermodular connections of individual nodes.
brainconn.centrality.participation_coef_sign(W, ci)	Participation coefficient is a measure of diversity of intermodular connections of individual nodes.
brainconn.centrality.subgraph_centrality(CIJ)	The subgraph centrality of a node is a weighted sum of closed walks of different lengths in the network starting and ending at the node.

brainconn.centrality.betweenness_bin

betweenness_bin(*G*)

Node betweenness centrality is the fraction of all shortest paths in the network that contain a given node. Nodes with high values of betweenness centrality participate in a large number of shortest paths.

Parameters

- **A** (NxN numpy.ndarray) – binary directed/undirected connection matrix
- **BC** (Nx1 numpy.ndarray) – node betweenness centrality vector

Notes

Betweenness centrality may be normalised to the range [0,1] as BC/[(N-1)(N-2)], where N is the number of nodes in the network.

Examples using brainconn.centrality.betweenness_bin

- *Calculate centrality measures*

brainconn.centrality.betweenness_wei

betweenness_wei(*G*)

Node betweenness centrality is the fraction of all shortest paths in the network that contain a given node. Nodes with high values of betweenness centrality participate in a large number of shortest paths.

Parameters **L** (NxN numpy.ndarray) – directed/undirected weighted connection matrix

Returns **BC** – node betweenness centrality vector

Return type Nx1 numpy.ndarray

Notes

The input matrix must be a connection-length matrix, typically obtained via a mapping from weight to length. For instance, in a weighted correlation network higher correlations are more naturally interpreted as shorter distances and the input matrix should consequently be some inverse of the connectivity matrix.

Betweenness centrality may be normalised to the range [0,1] as BC/[(N-1)(N-2)], where N is the number of nodes in the network.

Examples using brainconn.centrality.betweenness_wei

- *Calculate centrality measures*

brainconn.centrality.diversity_coef_sign

diversity_coef_sign(*W*, *ci*)

The Shannon entropy-based diversity coefficient measures the diversity of intermodular connections of individual nodes and ranges from 0 to 1.

Parameters

- **W** (NxN numpy.ndarray) – undirected connection matrix with positive and negative weights
- **ci** (Nx1 numpy.ndarray) – community affiliation vector

Returns

- **Hpos** (Nx1 numpy.ndarray) – diversity coefficient based on positive connections
- **Hneg** (Nx1 numpy.ndarray) – diversity coefficient based on negative connections

brainconn.centrality.edge_betweenness_bin

edge_betweenness_bin(*G*)

Edge betweenness centrality is the fraction of all shortest paths in the network that contain a given edge. Edges with high values of betweenness centrality participate in a large number of shortest paths.

Parameters **A** (NxN numpy.ndarray) – binary directed/undirected connection matrix

Returns

- **EBC** (NxN numpy.ndarray) – edge betweenness centrality matrix
- **BC** (Nx1 numpy.ndarray) – node betweenness centrality vector

Notes

Betweenness centrality may be normalised to the range [0,1] as BC/[(N-1)(N-2)], where N is the number of nodes in the network.

Examples using `brainconn.centrality.edge_betweenness_bin`

- *Calculate centrality measures*

brainconn.centrality.edge_betweenness_wei

edge_betweenness_wei(*G*)

Edge betweenness centrality is the fraction of all shortest paths in the network that contain a given edge. Edges with high values of betweenness centrality participate in a large number of shortest paths.

Parameters **L** (NxN `numpy.ndarray`) – directed/undirected weighted connection matrix

Returns

- **EBC** (NxN `numpy.ndarray`) – edge betweenness centrality matrix
- **BC** (Nx1 `numpy.ndarray`) – nodal betweenness centrality vector

Notes

The input matrix must be a connection-length matrix, typically obtained via a mapping from weight to length. For instance, in a weighted correlation network higher correlations are more naturally interpreted as shorter distances and the input matrix should consequently be some inverse of the connectivity matrix.

Betweenness centrality may be normalised to the range [0,1] as BC/[(N-1)(N-2)], where N is the number of nodes in the network.

Examples using `brainconn.centrality.edge_betweenness_wei`

- *Calculate centrality measures*

brainconn.centrality.eigenvector_centrality_und

eigenvector_centrality_und(*CIJ*)

Eigenector centrality is a self-referential measure of centrality: nodes have high eigenvector centrality if they connect to other nodes that have high eigenvector centrality. The eigenvector centrality of node i is equivalent to the ith element in the eigenvector corresponding to the largest eigenvalue of the adjacency matrix.

Parameters

- **CIJ** (NxN `numpy.ndarray`) – binary/weighted undirected adjacency matrix
- **v** (Nx1 `numpy.ndarray`) – eigenvector associated with the largest eigenvalue of the matrix

brainconn.centrality.erange

erange(*CIJ*)

Shortcuts are central edges which significantly reduce the characteristic path length in the network.

Parameters **CIJ** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **Erange** (NxN numpy.ndarray) – range for each edge, i.e. the length of the shortest path from i to j for edge c(i,j) after the edge has been removed from the graph
- **eta** (*float*) – average range for the entire graph
- **Eshort** (NxN numpy.ndarray) – entries are ones for shortcut edges
- **fs** (*float*) – fractions of shortcuts in the graph

Notes

Follows the treatment of 'shortcuts' by Duncan Watts

brainconn.centrality.flow_coef_bd

flow_coef_bd(*CIJ*)

Computes the flow coefficient for each node and averaged over the network, as described in Honey et al. (2007) PNAS. The flow coefficient is similar to betweenness centrality, but works on a local neighborhood. It is mathematically related to the clustering coefficient (cc) at each node as, fc+cc <= 1.

Parameters **CIJ** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **fc** (Nx1 numpy.ndarray) – flow coefficient for each node
- **FC** (*float*) – average flow coefficient over the network
- **total_flo** (*int*) – number of paths that "flow" across the central node

brainconn.centrality.gateway_coef_sign

gateway_coef_sign(*W*, *ci*, *centrality_type='degree'*)

The gateway coefficient is a variant of participation coefficient. It is weighted by how critical the connections are to intermodular connectivity (e.g. if a node is the only connection between its module and another module, it will have a higher gateway coefficient, unlike participation coefficient).

Parameters

- **W** (NxN numpy.ndarray) – undirected signed connection matrix
- **ci** (Nx1 numpy.ndarray) – community affiliation vector
- **centrality_type** (*enum*) – 'degree' - uses the weighted degree (i.e, node strength) 'betweenness' - uses the betweenness centrality

Returns

- **Gpos** (Nx1 numpy.ndarray) – gateway coefficient for positive weights
- **Gneg** (Nx1 numpy.ndarray) – gateway coefficient for negative weights

References

brainconn.centrality.kcoreness_centrality_bd

kcoreness_centrality_bd(*CIJ*)

The k-core is the largest subgraph comprising nodes of degree at least k. The coreness of a node is k if the node belongs to the k-core but not to the (k+1)-core. This function computes k-coreness of all nodes for a given binary directed connection matrix.

Parameters **CIJ** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **coreness** ((N,) numpy.ndarray) – node coreness
- **kn** ((N,) numpy.ndarray) – size of k-core

brainconn.centrality.kcoreness_centrality_bu

kcoreness_centrality_bu(*CIJ*)

The k-core is the largest subgraph comprising nodes of degree at least k. The coreness of a node is k if the node belongs to the k-core but not to the (k+1)-core. This function computes the coreness of all nodes for a given binary undirected connection matrix.

Parameters **CIJ** (NxN numpy.ndarray) – binary undirected connection matrix

Returns

- **coreness** ((N,) numpy.ndarray) – node coreness
- **kn** ((N,) numpy.ndarray) – size of k-core

brainconn.centrality.module_degree_zscore

module_degree_zscore(*W*, *ci*, *flag=0*)

The within-module degree z-score is a within-module version of degree centrality.

Parameters

- **W** (NxN numpy.ndarray) – binary/weighted directed/undirected connection matrix
- **ci** (*Nx1 np.array_like*) – community affiliation vector
- **flag** (*int*) –

  Graph type. 0: undirected graph (default) 1: directed graph in degree 2: directed graph out degree 3: directed graph in and out degree

Returns **Z** – within-module degree Z-score

Return type Nx1 numpy.ndarray

brainconn.centrality.pagerank_centrality

pagerank_centrality(*A*, *d*, *falff=None*)

The PageRank centrality is a variant of eigenvector centrality. This function computes the PageRank centrality of each vertex in a graph.

Formally, PageRank is defined as the stationary distribution achieved by instantiating a Markov chain on a graph. The PageRank centrality of a given vertex, then, is proportional to the number of steps (or amount of time) spent at that vertex as a result of such a process.

The PageRank index gets modified by the addition of a damping factor, d. In terms of a Markov chain, the damping factor specifies the fraction of the time that a random walker will transition to one of its current state's neighbors. The remaining fraction of the time the walker is restarted at a random vertex. A common value for the damping factor is d = 0.85.

Parameters

- **A** (NxN numpy.ndarray) – adjacency matrix
- **d** (*float*) – damping factor (see description)
- **falff** (Nx1 numpy.ndarray or None) – Initial page rank probability, non-negative values. Default value is None. If not specified, a naive bayesian prior is used.

Returns **r** – vectors of page rankings

Return type Nx1 numpy.ndarray

Notes

The algorithm will work well for smaller matrices (number of nodes around 1000 or less)

brainconn.centrality.participation_coef

participation_coef(*W*, *ci*, *degree='undirected'*)

Participation coefficient is a measure of diversity of intermodular connections of individual nodes.

Parameters

- **W** (NxN numpy.ndarray) – binary/weighted directed/undirected connection matrix
- **ci** (Nx1 numpy.ndarray) – community affiliation vector
- **degree** (*{'undirected', 'in', 'out'}, optional*) – Flag to describe nature of graph. 'undirected': For undirected graphs, 'in': Uses the in-degree, 'out': Uses the out-degree

Returns **P** – participation coefficient

Return type Nx1 numpy.ndarray

brainconn.centrality.participation_coef_sign

participation_coef_sign(*W*, *ci*)

Participation coefficient is a measure of diversity of intermodular connections of individual nodes.

Parameters

- **W** (NxN numpy.ndarray) – undirected connection matrix with positive and negative weights
- **ci** (Nx1 numpy.ndarray) – community affiliation vector

Returns

- **Ppos** (Nx1 numpy.ndarray) – participation coefficient from positive weights

- **Pneg** (Nx1 numpy.ndarray) – participation coefficient from negative weights

brainconn.centrality.subgraph_centrality

subgraph_centrality(*CIJ*)

The subgraph centrality of a node is a weighted sum of closed walks of different lengths in the network starting and ending at the node. This function returns a vector of subgraph centralities for each node of the network.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary adjacency matrix
- **Cs** (Nx1 numpy.ndarray) – subgraph centrality

1.3.2 brainconn.clustering: Clustering

Metrics which group nodes within graphs into clusters.

brainconn.clustering	Metrics which group nodes within graphs into clusters.
brainconn.clustering.agreement(ci[, buffsz])	Takes as input a set of vertex partitions CI of dimensions [vertex x partition].
brainconn.clustering.agreement_weighted(ci, wts)	D = AGREEMENT_WEIGHTED(CI,WTS) is identical to AGREEMENT, with the exception that each partitions contribution is weighted according to the corresponding scalar value stored in the vector WTS.
brainconn.clustering.clustering_coef_bd(A)	The clustering coefficient is the fraction of triangles around a node (equiv.
brainconn.clustering.clustering_coef_bu(G)	The clustering coefficient is the fraction of triangles around a node (equiv.
brainconn.clustering.clustering_coef_wd(W)	The weighted clustering coefficient is the average "intensity" of triangles around a node.
brainconn.clustering.clustering_coef_wu(W)	The weighted clustering coefficient is the average "intensity" of triangles around a node.
brainconn.clustering.clustering_coef_wu_sign(W)	Returns the weighted clustering coefficient generalized or separated for positive and negative weights.
brainconn.clustering.consensus_und(D, tau[, ...])	This algorithm seeks a consensus partition of the agreement matrix D.
brainconn.clustering.get_components(A[, ...])	Returns the components of an undirected graph specified by the binary and undirected adjacency matrix adj.
brainconn.clustering.get_components_old(A[, ...])	Returns the components of an undirected graph specified by the binary and undirected adjacency matrix adj.
brainconn.clustering.number_of_components(A)	
brainconn.clustering.transitivity_bd(A)	Transitivity is the ratio of 'triangles to triplets' in the network.
brainconn.clustering.transitivity_bu(A)	Transitivity is the ratio of 'triangles to triplets' in the network.
brainconn.clustering.transitivity_wd(W)	Transitivity is the ratio of 'triangles to triplets' in the network.
brainconn.clustering.transitivity_wu(W)	Transitivity is the ratio of 'triangles to triplets' in the network.

brainconn.clustering.agreement

agreement(*ci*, *buffsz=1000*)

Takes as input a set of vertex partitions CI of dimensions [vertex x partition]. Each column in CI contains the assignments of each vertex to a class/community/module. This function aggregates the partitions in CI into a square [vertex x vertex] agreement matrix D, whose elements indicate the number of times any two vertices were assigned to the same class.

In the case that the number of nodes and partitions in CI is large (greater than ~1000 nodes or greater than ~1000 partitions), the script can be made faster by computing D in pieces. The optional input BUFFSZ determines the size of each piece. Trial and error has found that BUFFSZ ~ 150 works well.

Parameters

- **ci** (NxM numpy.ndarray) – set of M (possibly degenerate) partitions of N nodes
- **buffsz** (*int | None*) – sets buffer size. If not specified, defaults to 1000

Returns **D** – agreement matrix

Return type NxN numpy.ndarray

brainconn.clustering.agreement_weighted

agreement_weighted(*ci*, *wts*)

D = AGREEMENT_WEIGHTED(CI,WTS) is identical to AGREEMENT, with the exception that each partitions contribution is weighted according to the corresponding scalar value stored in the vector WTS. As an example, suppose CI contained partitions obtained using some heuristic for maximizing modularity. A possible choice for WTS might be the Q metric (Newman's modularity score). Such a choice would add more weight to higher modularity partitions.

NOTE: Unlike AGREEMENT, this script does not have the input argument BUFFSZ.

Parameters

- **ci** (MxN numpy.ndarray) – set of M (possibly degenerate) partitions of N nodes
- **wts** (Mx1 numpy.ndarray) – relative weight of each partition

Returns **D** – weighted agreement matrix

Return type NxN numpy.ndarray

brainconn.clustering.clustering_coef_bd

clustering_coef_bd(*A*)

The clustering coefficient is the fraction of triangles around a node (equiv. the fraction of nodes neighbors that are neighbors of each other).

Parameters **A** (NxN numpy.ndarray) – binary directed connection matrix

Returns **C** – clustering coefficient vector

Return type Nx1 numpy.ndarray

Notes

Methodological note: In directed graphs, 3 nodes generate up to 8 triangles (2*2*2 edges). The number of existing triangles is the main diagonal of S^3/2. The number of all (in or out) neighbour pairs is K(K-1)/2. Each

neighbour pair may generate two triangles. "False pairs" are i<->j edge pairs (these do not generate triangles). The number of false pairs is the main diagonal of A^2. Thus the maximum possible number of triangles =

= (2 edges)*([ALL PAIRS] - [FALSE PAIRS]) = 2 * (K(K-1)/2 - diag(A^2)) = K(K-1) - 2(diag(A^2))

brainconn.clustering.clustering_coef_bu

clustering_coef_bu(*G*)

The clustering coefficient is the fraction of triangles around a node (equiv. the fraction of nodes neighbors that are neighbors of each other).

Parameters **A** (NxN numpy.ndarray) – binary undirected connection matrix

Returns **C** – clustering coefficient vector

Return type Nx1 numpy.ndarray

brainconn.clustering.clustering_coef_wd

clustering_coef_wd(*W*)

The weighted clustering coefficient is the average "intensity" of triangles around a node.

Parameters **W** (NxN numpy.ndarray) – weighted directed connection matrix

Returns **C** – clustering coefficient vector

Return type Nx1 numpy.ndarray

Notes

Methodological note (also see clustering_coef_bd) The weighted modification is as follows: - The numerator: adjacency matrix is replaced with weights matrix ^ 1/3 - The denominator: no changes from the binary version

The above reduces to symmetric and/or binary versions of the clustering coefficient for respective graphs.

brainconn.clustering.clustering_coef_wu

clustering_coef_wu(*W*)

The weighted clustering coefficient is the average "intensity" of triangles around a node.

Parameters **W** (NxN numpy.ndarray) – weighted undirected connection matrix

Returns **C** – clustering coefficient vector

Return type Nx1 numpy.ndarray

brainconn.clustering.clustering_coef_wu_sign

clustering_coef_wu_sign(*W*, *coef_type='default'*)

Returns the weighted clustering coefficient generalized or separated for positive and negative weights.

Three Algorithms are supported; herefore referred to as default, zhang, and constantini.

1. Default (Onnela et al.), as in the traditional clustering coefficient computation. Computed separately for positive and negative weights.

2. Zhang & Horvath. Similar to Onnela formula except weight information incorporated in denominator. Reduces sensitivity of the measure to weights directly connected to the node of interest. Computed separately for positive and negative weights.
3. Constantini & Perugini generalization of Zhang & Horvath formula. Takes both positive and negative weights into account simultaneously. Particularly sensitive to non-redundancy in path information based on sign. Returns only one value.

Parameters

- **W** (NxN numpy.ndarray) – weighted undirected connection matrix
- **corr_type** (*{'default', 'zhang', 'constantini'}*) – Allowed values are 'default', 'zhang', 'constantini'

Returns

- **Cpos** (Nx1 numpy.ndarray) – Clustering coefficient vector for positive weights
- **Cneg** (Nx1 numpy.ndarray) – Clustering coefficient vector for negative weights, unless coef_type == 'constantini'.
- *References* – Onnela et al. (2005) Phys Rev E 71:065103 Zhang & Horvath (2005) Stat Appl Genet Mol Biol 41:1544-6115 Costantini & Perugini (2014) PLOS ONE 9:e88669

brainconn.clustering.consensus_und

consensus_und (*D*, *tau*, *reps=1000*)

This algorithm seeks a consensus partition of the agreement matrix D. The algorithm used here is almost identical to the one introduced in Lancichinetti & Fortunato (2012): The agreement matrix D is thresholded at a level TAU to remove an weak elements. The resulting matrix is then partitions REPS number of times using the Louvain algorithm (in principle, any clustering algorithm that can handle weighted matrixes is a suitable alternative to the Louvain algorithm and can be substituted in its place). This clustering produces a set of partitions from which a new agreement is built. If the partitions have not converged to a single representative partition, the above process repeats itself, starting with the newly built agreement matrix.

NOTE: In this implementation, the elements of the agreement matrix must be converted into probabilities.

NOTE: This implementation is slightly different from the original algorithm proposed by Lanchichinetti & Fortunato. In its original version, if the thresholding produces singleton communities, those nodes are reconnected to the network. Here, we leave any singleton communities disconnected.

Parameters

- **D** (NxN numpy.ndarray) – agreement matrix with entries between 0 and 1 denoting the probability of finding node i in the same cluster as node j
- **tau** (*float*) – threshold which controls the resolution of the reclustering
- **reps** (*int*) – number of times the clustering algorithm is reapplied. default value is 1000.

Returns **ciu** – consensus partition

Return type Nx1 numpy.ndarray

brainconn.clustering.get_components

get_components (*A*, *no_depend=False*)

Returns the components of an undirected graph specified by the binary and undirected adjacency matrix adj.

Components and their constitutent nodes are assigned the same index and stored in the vector, comps. The vector, comp_sizes, contains the number of nodes beloning to each component.

Parameters

- **A** (NxN numpy.ndarray) – binary undirected adjacency matrix
- **no_depend** (*Any*) – Does nothing, included for backwards compatibility

Returns

- **comps** (Nx1 numpy.ndarray) – vector of component assignments for each node
- **comp_sizes** (Mx1 numpy.ndarray) – vector of component sizes

Notes

Note: disconnected nodes will appear as components with a component size of 1

Note: The identity of each component (i.e. its numerical value in the result) is not guaranteed to be identical the value returned in BCT, matlab code, although the component topology is.

Many thanks to Nick Cullen for providing this implementation

brainconn.clustering.get_components_old

get_components_old(*A*, *no_depend=False*)

Returns the components of an undirected graph specified by the binary and undirected adjacency matrix adj. Components and their constitutent nodes are assigned the same index and stored in the vector, comps. The vector, comp_sizes, contains the number of nodes beloning to each component.

Parameters

- **adj** (NxN numpy.ndarray) – binary undirected adjacency matrix
- **no_depend** (*bool*) – If true, doesn't import networkx to do the calculation. Default value is false.

Returns

- **comps** (Nx1 numpy.ndarray) – vector of component assignments for each node
- **comp_sizes** (Mx1 numpy.ndarray) – vector of component sizes

Notes

Note: disconnected nodes will appear as components with a component size of 1

Note: The identity of each component (i.e. its numerical value in the result) is not guaranteed to be identical the value returned in BCT, although the component topology is.

Note: networkx is used to do the computation efficiently. If networkx is not available a breadth-first search that does not depend on networkx is used instead, but this is less efficient. The corresponding BCT function does the computation by computing the Dulmage-Mendelsohn decomposition. I don't know what a Dulmage-Mendelsohn decomposition is and there doesn't appear to be a python equivalent. If you think of a way to implement this better, let me know.

brainconn.clustering.number_of_components

number_of_components(*A*)

brainconn.clustering.transitivity_bd

transitivity_bd(*A*)

Transitivity is the ratio of 'triangles to triplets' in the network. (A classical version of the clustering coefficient).

Parameters **A** (NxN numpy.ndarray) – binary directed connection matrix

Returns **T** – transitivity scalar

Return type float

Notes

Methodological note: In directed graphs, 3 nodes generate up to 8 triangles (2*2*2 edges). The number of existing triangles is the main

diagonal of S^3/2. The number of all (in or out) neighbour pairs is K(K-1)/2. Each neighbour pair may generate two triangles. "False pairs" are i<->j edge pairs (these do not generate triangles). The number of false pairs is the main diagonal of A^2. Thus the maximum possible number of triangles = (2 edges)*([ALL PAIRS] - [FALSE PAIRS])

= 2 * (K(K-1)/2 - diag(A^2)) = K(K-1) - 2(diag(A^2))

brainconn.clustering.transitivity_bu

transitivity_bu(*A*)

Transitivity is the ratio of 'triangles to triplets' in the network. (A classical version of the clustering coefficient).

Parameters **A** (NxN numpy.ndarray) – binary undirected connection matrix

Returns **T** – transitivity scalar

Return type float

brainconn.clustering.transitivity_wd

transitivity_wd(*W*)

Transitivity is the ratio of 'triangles to triplets' in the network. (A classical version of the clustering coefficient).

Parameters **W** (NxN numpy.ndarray) – weighted directed connection matrix

Returns

- **T** (*int*) – transitivity scalar
- *Methodological note (also see note for clustering_coef_bd)*
- *The weighted modification is as follows*
- **- The numerator** (*adjacency matrix is replaced with weights matrix ^ 1/3*)
- **- The denominator** (*no changes from the binary version*)
- *The above reduces to symmetric and/or binary versions of the clustering*

- *coefficient for respective graphs.*

brainconn.clustering.transitivity_wu

transitivity_wu(*W*)

Transitivity is the ratio of 'triangles to triplets' in the network. (A classical version of the clustering coefficient).

Parameters **W** (NxN numpy.ndarray) – weighted undirected connection matrix

Returns **T** – transitivity scalar

Return type int

1.3.3 brainconn.core: Core

Metrics which identify the most important nodes in graphs.

brainconn.core	Metrics which identify the most important nodes in graphs.
brainconn.core.assortativity_bin(CIJ[, flag])	The assortativity coefficient is a correlation coefficient between the degrees of all nodes on two opposite ends of a link.
brainconn.core.assortativity_wei(CIJ[, flag])	The assortativity coefficient is a correlation coefficient between the strengths (weighted degrees) of all nodes on two opposite ends of a link.
brainconn.core.core_periphery_dir(W[, gamma, C0])	The optimal core/periphery subdivision is a partition of the network into two nonoverlapping groups of nodes, a core group and a periphery group.
brainconn.core.kcore_bd(CIJ, k[, peel])	The k-core is the largest subnetwork comprising nodes of degree at least k.
brainconn.core.kcore_bu(CIJ, k[, peel])	The k-core is the largest subnetwork comprising nodes of degree at least k.
brainconn.core.local_assortativity_wu_sign(W)	Local assortativity measures the extent to which nodes are connected to nodes of similar strength.
brainconn.core.rich_club_bd(CIJ[, klevel])	The rich club coefficient, R, at level k is the fraction of edges that connect nodes of degree k or higher out of the maximum number of edges that such nodes might share.
brainconn.core.rich_club_bu(CIJ[, klevel])	The rich club coefficient, R, at level k is the fraction of edges that connect nodes of degree k or higher out of the maximum number of edges that such nodes might share.
brainconn.core.rich_club_wd(CIJ[, klevel])	**param CIJ** weighted directed connection matrix
brainconn.core.rich_club_wu(CIJ[, klevel])	**param CIJ** weighted undirected connection matrix
brainconn.core.score_wu(CIJ, s)	The s-core is the largest subnetwork comprising nodes of strength at least s.

brainconn.core.assortativity_bin

assortativity_bin(*CIJ*, *flag=0*)

The assortativity coefficient is a correlation coefficient between the degrees of all nodes on two opposite ends of a link. A positive assortativity coefficient indicates that nodes tend to link to other nodes with the same or similar degree.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary directed/undirected connection matrix
- **flag** (*int*) – 0 : undirected graph; degree/degree correlation 1 : directed graph; out-degree/in-degree correlation 2 : directed graph; in-degree/out-degree correlation 3 : directed graph; out-degree/out-degree correlation 4 : directed graph; in-degree/in-degreen correlation

Returns **r** – assortativity coefficient

Return type float

Notes

The function accepts weighted networks, but all connection weights are ignored. The main diagonal should be empty. For flag 1 the function computes the directed assortativity described in Rubinov and Sporns (2010) NeuroImage.

brainconn.core.assortativity_wei

assortativity_wei(*CIJ*, *flag=0*)

The assortativity coefficient is a correlation coefficient between the strengths (weighted degrees) of all nodes on two opposite ends of a link. A positive assortativity coefficient indicates that nodes tend to link to other nodes with the same or similar strength.

Parameters

- **CIJ** (NxN numpy.ndarray) – weighted directed/undirected connection matrix
- **flag** (*int*) – 0 : undirected graph; strength/strength correlation 1 : directed graph; out-strength/in-strength correlation 2 : directed graph; in-strength/out-strength correlation 3 : directed graph; out-strength/out-strength correlation 4 : directed graph; in-strength/in-strengthn correlation

Returns **r** – assortativity coefficient

Return type float

Notes

The main diagonal should be empty. For flag 1 the function computes the directed assortativity described in Rubinov and Sporns (2010) NeuroImage.

brainconn.core.core_periphery_dir

core_periphery_dir(*W*, *gamma=1*, *C0=None*)

The optimal core/periphery subdivision is a partition of the network into two nonoverlapping groups of nodes,

a core group and a periphery group. The number of core-group edges is maximized, and the number of within periphery edges is minimized.

The core-ness is a statistic which quantifies the goodness of the optimal core/periphery subdivision (with arbitrary relative value).

The algorithm uses a variation of the Kernighan-Lin graph partitioning algorithm to optimize a core-structure objective described in Borgatti & Everett (2000) Soc Networks 21:375-395

See Rubinov, Ypma et al. (2015) PNAS 112:10032-7

Parameters

- **W** (NxN numpy.ndarray) – directed connection matrix
- **gamma** (*core-ness resolution parameter*) – Default value = 1 gamma > 1 detects small core, large periphery 0 < gamma < 1 detects large core, small periphery
- **C0** (NxN numpy.ndarray) – Initial core structure

brainconn.core.kcore_bd

kcore_bd(*CIJ*, *k*, *peel=False*)

The k-core is the largest subnetwork comprising nodes of degree at least k. This function computes the k-core for a given binary directed connection matrix by recursively peeling off nodes with degree lower than k, until no such nodes remain.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary directed adjacency matrix
- **k** (*int*) – level of k-core
- **peel** (*bool*) – If True, additionally calculates peelorder and peellevel. Defaults to False.

Returns

- **CIJkcore** (NxN numpy.ndarray) – connection matrix of the k-core. This matrix only contains nodes of degree at least k.
- **kn** (*int*) – size of k-core
- **peelorder** (Nx1 numpy.ndarray) – indices in the order in which they were peeled away during k-core decomposition. only returned if peel is specified.
- **peellevel** (Nx1 numpy.ndarray) – corresponding level - nodes in at the same level have been peeled away at the same time. only return if peel is specified

Notes

'peelorder' and 'peellevel' are similar the the k-core sub-shells described in Modha and Singh (2010).

brainconn.core.kcore_bu

kcore_bu(*CIJ*, *k*, *peel=False*)

The k-core is the largest subnetwork comprising nodes of degree at least k. This function computes the k-core for a given binary undirected connection matrix by recursively peeling off nodes with degree lower than k, until no such nodes remain.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary undirected connection matrix
- **k** (*int*) – level of k-core
- **peel** (*bool*) – If True, additionally calculates peelorder and peellevel. Defaults to False.

Returns

- **CIJkcore** (NxN numpy.ndarray) – connection matrix of the k-core. This matrix only contains nodes of degree at least k.
- **kn** (*int*) – size of k-core
- **peelorder** (Nx1 numpy.ndarray) – indices in the order in which they were peeled away during k-core decomposition. only returned if peel is specified.
- **peellevel** (Nx1 numpy.ndarray) – corresponding level - nodes in at the same level have been peeled away at the same time. only return if peel is specified

Notes

'peelorder' and 'peellevel' are similar the the k-core sub-shells described in Modha and Singh (2010).

brainconn.core.local_assortativity_wu_sign

local_assortativity_wu_sign(*W*)

Local assortativity measures the extent to which nodes are connected to nodes of similar strength. Adapted from Thedchanamoorthy et al. 2014 formula to allowed weighted/signed networks.

Parameters **W** (NxN numpy.ndarray) – undirected connection matrix with positive and negative weights

Returns

- **loc_assort_pos** (Nx1 numpy.ndarray) – local assortativity from positive weights
- **loc_assort_neg** (Nx1 numpy.ndarray) – local assortativity from negative weights

brainconn.core.rich_club_bd

rich_club_bd(*CIJ*, *klevel=None*)

The rich club coefficient, R, at level k is the fraction of edges that connect nodes of degree k or higher out of the maximum number of edges that such nodes might share.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary directed connection matrix
- **klevel** (*int | None*) – sets the maximum level at which the rich club coefficient will be calculated. If None (default), the maximum level is set to the maximum degree of the adjacency matrix

Returns

- **R** (Kx1 numpy.ndarray) – vector of rich-club coefficients for levels 1 to klevel
- **Nk** (*int*) – number of nodes with degree > k
- **Ek** (*int*) – number of edges remaining in subgraph with degree > k

brainconn.core.rich_club_bu

rich_club_bu(*CIJ*, *klevel=None*)

The rich club coefficient, R, at level k is the fraction of edges that connect nodes of degree k or higher out of the maximum number of edges that such nodes might share.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary undirected connection matrix
- **klevel** (*int | None*) – sets the maximum level at which the rich club coefficient will be calculated. If None (default), the maximum level is set to the maximum degree of the adjacency matrix

Returns

- **R** (Kx1 numpy.ndarray) – vector of rich-club coefficients for levels 1 to klevel
- **Nk** (*int*) – number of nodes with degree > k
- **Ek** (*int*) – number of edges remaining in subgraph with degree > k

brainconn.core.rich_club_wd

rich_club_wd(*CIJ*, *klevel=None*)

Parameters

- **CIJ** (NxN numpy.ndarray) – weighted directed connection matrix
- **klevel** (*int | None*) – sets the maximum level at which the rich club coefficient will be calculated. If None (default), the maximum level is set to the maximum degree of the adjacency matrix

Returns **Rw** – vector of rich-club coefficients for levels 1 to klevel

Return type Kx1 numpy.ndarray

brainconn.core.rich_club_wu

rich_club_wu(*CIJ*, *klevel=None*)

Parameters

- **CIJ** (NxN numpy.ndarray) – weighted undirected connection matrix
- **klevel** (*int | None*) – sets the maximum level at which the rich club coefficient will be calculated. If None (default), the maximum level is set to the maximum degree of the adjacency matrix

Returns **Rw** – vector of rich-club coefficients for levels 1 to klevel

Return type Kx1 numpy.ndarray

brainconn.core.score_wu

score_wu(*CIJ*, *s*)

The s-core is the largest subnetwork comprising nodes of strength at least s. This function computes the s-core for a given weighted undirected connection matrix. Computation is analogous to the more widely used k-core, but is based on node strengths instead of node degrees.

Parameters

- **CIJ** (NxN numpy.ndarray) – weighted undirected connection matrix
- **s** (*float*) – level of s-core. Note that can take on any fractional value.

Returns

- **CIJscore** (NxN numpy.ndarray) – connection matrix of the s-core. This matrix contains only nodes with a strength of at least s.
- **sn** (*int*) – size of s-core

1.3.4 brainconn.degree: Degree

Metrics which identify the most important nodes in graphs.

brainconn.degree	Metrics which identify the most important nodes in graphs.
brainconn.degree.degrees_dir(CIJ)	Node degree is the number of links connected to the node.
brainconn.degree.degrees_und(CIJ)	Node degree is the number of links connected to the node.
brainconn.degree.jdegree(CIJ)	This function returns a matrix in which the value of each element (u,v) corresponds to the number of nodes that have u outgoing connections and v incoming connections.
brainconn.degree.strengths_dir(CIJ)	Node strength is the sum of weights of links connected to the node.
brainconn.degree.strengths_und(CIJ)	Node strength is the sum of weights of links connected to the node.
brainconn.degree.strengths_und_sign(W)	Node strength is the sum of weights of links connected to the node.

brainconn.degree.degrees_dir

degrees_dir(*CIJ*)

Node degree is the number of links connected to the node. The indegree is the number of inward links and the outdegree is the number of outward links.

Parameters **CIJ** (NxN numpy.ndarray) – directed binary/weighted connection matrix

Returns

- **in_degree** (Nx1 numpy.ndarray) – node in-degree
- **out_degree** (Nx1 numpy.ndarray) – node out-degree
- **deg** (Nx1 numpy.ndarray) – node degree (in-degree + out-degree)

Notes

Inputs are assumed to be on the columns of the CIJ matrix. Weight information is discarded.

Examples using `brainconn.degree.degrees_dir`

- *Calculate degree measures*

`brainconn.degree.degrees_und`

`degrees_und`(*CIJ*)

Node degree is the number of links connected to the node.

Parameters **`CIJ`** (NxN `numpy.ndarray`) – undirected binary/weighted connection matrix

Returns **deg** – node degree

Return type Nx1 `numpy.ndarray`

Notes

Weight information is discarded.

Examples using `brainconn.degree.degrees_und`

- *Calculate degree measures*

`brainconn.degree.jdegree`

`jdegree`(*CIJ*)

This function returns a matrix in which the value of each element (u,v) corresponds to the number of nodes that have u outgoing connections and v incoming connections.

Parameters **`CIJ`** (NxN `numpy.ndarray`) – directed binary/weighted connnection matrix

Returns

- **J** (ZxZ `numpy.ndarray`) – joint degree distribution matrix (shifted by one, replicates matlab one-based-indexing)
- **J_od** (*int*) – number of vertices with out_degree>in_degree
- **J_id** (*int*) – number of vertices with in_degree>out_degree
- **J_bl** (*int*) – number of vertices with in_degree==out_degree

Notes

Weights are discarded.

Examples using `brainconn.degree.jdegree`

- *Calculate degree measures*

brainconn.degree.strengths_dir

strengths_dir(*CIJ*)

Node strength is the sum of weights of links connected to the node. The instrength is the sum of inward link weights and the outstrength is the sum of outward link weights.

Parameters **CIJ** (NxN numpy.ndarray) – directed weighted connection matrix

Returns

- **is** (Nx1 numpy.ndarray) – node in-strength
- **os** (Nx1 numpy.ndarray) – node out-strength
- **str** (Nx1 numpy.ndarray) – node strength (in-strength + out-strength)

Notes

Inputs are assumed to be on the columns of the CIJ matrix.

Examples using brainconn.degree.strengths_dir

- *Calculate degree measures*

brainconn.degree.strengths_und

strengths_und(*CIJ*)

Node strength is the sum of weights of links connected to the node.

Parameters **CIJ** (NxN numpy.ndarray) – undirected weighted connection matrix

Returns **str** – node strengths

Return type Nx1 numpy.ndarray

Examples using brainconn.degree.strengths_und

- *Calculate degree measures*

brainconn.degree.strengths_und_sign

strengths_und_sign(*W*)

Node strength is the sum of weights of links connected to the node.

Parameters **W** (NxN numpy.ndarray) – undirected connection matrix with positive and negative weights

Returns

- **Spos** (Nx1 numpy.ndarray) – nodal strength of positive weights
- **Sneg** (Nx1 numpy.ndarray) – nodal strength of positive weights
- **vpos** (*float*) – total positive weight
- **vneg** (*float*) – total negative weight

Examples using `brainconn.degree.strengths_und_sign`

- *Calculate degree measures*

1.3.5 `brainconn.distance`: Distance

Metrics which identify the most important nodes in graphs.

brainconn.distance	Metrics which identify the most important nodes in graphs.
brainconn.distance.breadthdist(CIJ)	The binary reachability matrix describes reachability between all pairs of nodes.
brainconn.distance.breadth(CIJ, source)	Implementation of breadth-first search.
brainconn.distance.charpath(D[, ...])	The characteristic path length is the average shortest path length in the network.
brainconn.distance.cycprob(Pq)	Cycles are paths which begin and end at the same node.
brainconn.distance.distance_bin(G)	The distance matrix contains lengths of shortest paths between all pairs of nodes.
brainconn.distance.distance_wei(G)	The distance matrix contains lengths of shortest paths between all pairs of nodes.
brainconn.distance.distance_wei_floyd(adjacency)	Computes the topological length of the shortest possible path connecting every pair of nodes in the network.
brainconn.distance.retrieve_shortest_path(s, ...)	Returns nodes comprising shortest path between *s* and *t*
brainconn.distance.efficiency_bin(G[, local])	The global efficiency is the average of inverse shortest path length, and is inversely related to the characteristic path length.
brainconn.distance.efficiency_wei(Gw[, local])	The global efficiency is the average of inverse shortest path length, and is inversely related to the characteristic path length.
brainconn.distance.findpaths(CIJ, qmax, sources)	Paths are sequences of linked nodes, that never visit a single node more than once.
brainconn.distance.findwalks(CIJ)	Walks are sequences of linked nodes, that may visit a single node more than once.
brainconn.distance.mean_first_passage_time(...)	Calculates mean first passage time of *adjacency*
brainconn.distance.reachdist(CIJ[, ...])	The binary reachability matrix describes reachability between all pairs of nodes.
brainconn.distance.search_information(adjacency)	Calculates search information of *adjacency*.

`brainconn.distance.breadthdist`

`breadthdist`(*CIJ*)

The binary reachability matrix describes reachability between all pairs of nodes. An entry (u,v)=1 means that there exists a path from node u to node v; alternatively (u,v)=0.

The distance matrix contains lengths of shortest paths between all pairs of nodes. An entry (u,v) represents the length of shortest path from node u to node v. The average shortest path length is the characteristic path length of the network.

Parameters **`CIJ`** (NxN `numpy.ndarray`) – binary directed/undirected connection matrix

Returns

- **R** (NxN numpy.ndarray) – binary reachability matrix
- **D** (NxN numpy.ndarray) – distance matrix

Notes

slower but less memory intensive than "reachdist.m".

brainconn.distance.breadth

breadth(*CIJ*, *source*)

Implementation of breadth-first search.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary directed/undirected connection matrix
- **source** (*int*) – source vertex

Returns

- **distance** (Nx1 numpy.ndarray) – vector of distances between source and ith vertex (0 for source)
- **branch** (Nx1 numpy.ndarray) – vertex that precedes i in the breadth-first search (-1 for source)

Notes

Breadth-first search tree does not contain all paths (or all shortest paths), but allows the determination of at least one path with minimum distance. The entire graph is explored, starting from source vertex 'source'.

brainconn.distance.charpath

charpath(*D*, *include_diagonal=False*, *include_infinite=True*)

The characteristic path length is the average shortest path length in the network. The global efficiency is the average inverse shortest path length in the network.

Parameters

- **D** (NxN numpy.ndarray) – distance matrix
- **include_diagonal** (*bool*) – If True, include the weights on the diagonal. Default value is False.
- **include_infinite** (*bool*) – If True, include infinite distances in calculation

Returns

- **lambda** (*float*) – characteristic path length
- **efficiency** (*float*) – global efficiency
- **ecc** (Nx1 numpy.ndarray) – eccentricity at each vertex
- **radius** (*float*) – radius of graph
- **diameter** (*float*) – diameter of graph

Notes

The input distance matrix may be obtained with any of the distance functions, e.g. distance_bin, distance_wei. Characteristic path length is calculated as the global mean of the distance matrix D, excludings any 'Infs' but including distances on the main diagonal.

brainconn.distance.cycprob

cycprob(*Pq*)

Cycles are paths which begin and end at the same node. Cycle probability for path length d, is the fraction of all paths of length d-1 that may be extended to form cycles of length d.

Parameters **Pq** (NxNxQ numpy.ndarray) – Path matrix with Pq[i,j,q] = number of paths from i to j of length q. Produced by findpaths()

Returns

- **fcyc** (Qx1 numpy.ndarray) – fraction of all paths that are cycles for each path length q
- **pcyc** (Qx1 numpy.ndarray) – probability that a non-cyclic path of length q-1 can be extended to form a cycle of length q for each path length q

brainconn.distance.distance_bin

distance_bin(*G*)

The distance matrix contains lengths of shortest paths between all pairs of nodes. An entry (u,v) represents the length of shortest path from node u to node v. The average shortest path length is the characteristic path length of the network.

Parameters **A** (NxN numpy.ndarray) – binary directed/undirected connection matrix

Returns **D** – distance matrix

Return type NxN

Notes

Lengths between disconnected nodes are set to Inf. Lengths on the main diagonal are set to 0. Algorithm: Algebraic shortest paths.

brainconn.distance.distance_wei

distance_wei(*G*)

The distance matrix contains lengths of shortest paths between all pairs of nodes. An entry (u,v) represents the length of shortest path from node u to node v. The average shortest path length is the characteristic path length of the network.

Parameters **L** (NxN numpy.ndarray) – Directed/undirected connection-length matrix. NB L is not the adjacency matrix. See below.

Returns

- **D** (NxN numpy.ndarray) – distance (shortest weighted path) matrix
- **B** (NxN numpy.ndarray) – matrix of number of edges in shortest weighted path

Notes

The input matrix must be a connection-length matrix, typically

obtained via a mapping from weight to length. For instance, in a weighted correlation network higher correlations are more naturally interpreted as shorter distances and the input matrix should consequently be some inverse of the connectivity matrix.

The number of edges in shortest weighted paths may in general

exceed the number of edges in shortest binary paths (i.e. shortest paths computed on the binarized connectivity matrix), because shortest weighted paths have the minimal weighted distance, but not necessarily the minimal number of edges.

Lengths between disconnected nodes are set to Inf. Lengths on the main diagonal are set to 0.

Algorithm: Dijkstra's algorithm.

brainconn.distance.distance_wei_floyd

distance_wei_floyd(*adjacency*, *transform=None*)

Computes the topological length of the shortest possible path connecting every pair of nodes in the network.

Parameters

- **D** (*(N x N) array_like*) – Weighted/unweighted, direct/undirected connection weight/length array
- **transform** (*str, optional*) – If *adjacency* is a connection weight array, specify a transform to map input connection weights to connection lengths. Options include ['log', 'inv'], where 'log' is *-np.log(adjacency)* and 'inv' is *1/adjacency*. Default: None

Returns

- **SPL** (*(N x N) ndarray*) – Weighted/unweighted shortest path-length array. If *D* is a directed graph, then *SPL* is not symmetric
- **hops** (*(N x N) ndarray*) – Number of edges in the shortest path array. If *D* is unweighted, *SPL* and *hops* are identical.
- **Pmat** (*(N x N) ndarray*) – Element *[i,j]* of this array indicates the next node in the shortest path between *i* and *j*. This array is used as an input argument for function *retrieve_shortest_path()*, which returns as output the sequence of nodes comprising the shortest path between a given pair of nodes.

Notes

There may be more than one shortest path between any pair of nodes in the network. Non-unique shortest paths are termed shortest path degeneracies and are most likely to occur in unweighted networks. When the shortest-path is degenerate, the elements of *Pmat* correspond to the first shortest path discovered by the algorithm.

The input array may be either a connection weight or length array. The connection length array is typically obtained with a mapping from weight to length, such that higher weights are mapped to shorter lengths (see argument *transform*, above).

Originally written in Matlab by Andrea Avena-Koenigsberger (IU, 2012) *[1] [2] [3] [4]*.

References

brainconn.distance.retrieve_shortest_path

retrieve_shortest_path(*s*, *t*, *hops*, *Pmat*)

Returns nodes comprising shortest path between *s* and *t*

This function finds the sequence of nodes that comprise the shortest path between a given source and target node.

Parameters

- **s** (*int*) – Source node, i.e. node where the shortest path begins
- **t** (*int*) – Target node, i.e. node where the shortest path ends
- **hops** (*(N x N) array_like*) – Number of edges in the path. This array may be obtained as the second output argument of the function *distance_wei_floyd*.
- **Pmat** (*(N x N) array_like*) – Array whose elements *Pmat[k,t]* indicate the next node in the shortest path between nodes *k* and *t*. This array may be obtained as the third output of the function *distance_wei_floyd*.

Returns **path** – Nodes (indices) comprising the shortest path between *s* and *t*

Return type ndarray

Notes

Originally written in Matlab by Andrea Avena-Koenigsberger and Joaquin Goni (IU, 2012)

brainconn.distance.efficiency_bin

efficiency_bin(*G*, *local=False*)

The global efficiency is the average of inverse shortest path length, and is inversely related to the characteristic path length.

The local efficiency is the global efficiency computed on the neighborhood of the node, and is related to the clustering coefficient.

Parameters

- **A** (NxN numpy.ndarray) – binary undirected connection matrix
- **local** (*bool*) – If True, computes local efficiency instead of global efficiency. Default value = False.

Returns

- **Eglob** (*float*) – global efficiency, only if local=False
- **Eloc** (Nx1 numpy.ndarray) – local efficiency, only if local=True

brainconn.distance.efficiency_wei

efficiency_wei(*Gw*, *local=False*)

The global efficiency is the average of inverse shortest path length, and is inversely related to the characteristic path length.

The local efficiency is the global efficiency computed on the neighborhood of the node, and is related to the clustering coefficient.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted connection matrix (all weights in W must be between 0 and 1)
- **local** (*bool*) – If True, computes local efficiency instead of global efficiency. Default value = False.

Returns

- **Eglob** (*float*) – global efficiency, only if local=False
- **Eloc** (Nx1 numpy.ndarray) – local efficiency, only if local=True

Notes

The efficiency is computed using an auxiliary connection-length

matrix L, defined as L_ij = 1/W_ij for all nonzero L_ij; This has an intuitive interpretation, as higher connection weights intuitively correspond to shorter lengths.

The weighted local efficiency broadly parallels the weighted

clustering coefficient of Onnela et al. (2005) and distinguishes the influence of different paths based on connection weights of the corresponding neighbors to the node in question. In other words, a path between two neighbors with strong connections to the node in question contributes more to the local efficiency than a path between two weakly connected neighbors. Note that this weighted variant of the local efficiency is hence not a strict generalization of the binary variant.

Algorithm: Dijkstra's algorithm

brainconn.distance.findpaths

findpaths(*CIJ*, *qmax*, *sources*, *savepths=False*)

Paths are sequences of linked nodes, that never visit a single node more than once. This function finds all paths that start at a set of source nodes, up to a specified length. Warning: very memory-intensive.

Parameters

- **CIJ** (NxN numpy.ndarray) – binary directed/undirected connection matrix
- **qmax** (*int*) – maximal path length
- **sources** (Nx1 numpy.ndarray) – source units from which paths are grown
- **savepths** (*bool*) – True if all paths are to be collected and returned. This functionality is currently not enabled.

Returns

- **Pq** (NxNxQ numpy.ndarray) – Path matrix with P[i,j,jq] = number of paths from i to j with length q
- **tpath** (*int*) – total number of paths found
- **plq** (Qx1 numpy.ndarray) – path length distribution as a function of q
- **qstop** (*int*) – path length at which findpaths is stopped

- **allpths** (*None*) – a matrix containing all paths up to qmax. This function is extremely complicated and reimplementing it in bctpy is not straightforward.
- **util** (NxQ numpy.ndarray) – node use index

Notes

Note that Pq(:,:,N) can only carry entries on the diagonal, as all "legal" paths of length N-1 must terminate. Cycles of length N are possible, with all vertices visited exactly once (except for source and target). 'qmax = N' can wreak havoc (due to memory problems).

Note: Weights are discarded. Note: I am certain that this algorithm is rather inefficient - suggestions for improvements are welcome.

brainconn.distance.findwalks

findwalks(*CIJ*)

Walks are sequences of linked nodes, that may visit a single node more than once. This function finds the number of walks of a given length, between any two nodes.

Parameters **CIJ** (NxN numpy.ndarray) – binary directed/undirected connection matrix

Returns

- **Wq** (NxNxQ numpy.ndarray) – Wq[i,j,q] is the number of walks from i to j of length q
- **twalk** (*int*) – total number of walks found
- **wlq** (Qx1 numpy.ndarray) – walk length distribution as a function of q

Notes

Wq grows very quickly for larger N,K,q. Weights are discarded.

brainconn.distance.mean_first_passage_time

mean_first_passage_time(*adjacency*)

Calculates mean first passage time of *adjacency*

The first passage time from i to j is the expected number of steps it takes a random walker starting at node i to arrive for the first time at node j. The mean first passage time is not a symmetric measure: *mfpt(i,j)* may be different from *mfpt(j,i)*.

Parameters **adjacency** (*(N x N) array_like*) – Weighted/unweighted, direct/undirected connection weight/length array

Returns **MFPT** – Pairwise mean first passage time array

Return type (N x N) ndarray

References

brainconn.distance.reachdist

reachdist(*CIJ*, *ensure_binary=True*)

The binary reachability matrix describes reachability between all pairs of nodes. An entry (u,v)=1 means that there exists a path from node u to node v; alternatively (u,v)=0.

The distance matrix contains lengths of shortest paths between all pairs of nodes. An entry (u,v) represents the length of shortest path from node u to node v. The average shortest path length is the characteristic path length of the network.

Parameters

- **CIJ** (NxN `numpy.ndarray`) – binary directed/undirected connection matrix
- **ensure_binary** (*bool*) – Binarizes input. Defaults to true. No user who is not testing something will ever want to not use this, use distance_wei instead for unweighted matrices.

Returns

- **R** (NxN `numpy.ndarray`) – binary reachability matrix
- **D** (NxN `numpy.ndarray`) – distance matrix

Notes

faster but more memory intensive than "breadthdist.m".

brainconn.distance.search_information

search_information(*adjacency*, *transform=None*, *has_memory=False*)

Calculates search information of *adjacency*.

Computes the amount of information (measured in bits) that a random walker needs to follow the shortest path between a given pair of nodes *[1] [2]*.

Parameters

- **adjacency** (*(N x N) array_like*) – Weighted/unweighted, direct/undirected connection weight/length array
- **transform** (*str, optional*) – If *adjacency* is a connection weight array, specify a transform to map input connection weights to connection lengths. Options include ['log', 'inv'], where 'log' is *-np.log(adjacency)* and 'inv' is *1/adjacency*. Default: None
- **has_memory** (*bool, optional*) – This flag defines whether or not the random walker "remembers" its previous step, which has the effect of reducing the amount of information needed to find the next state. Default: False

Returns **SI** – Pair-wise search information array. Note that *SI[i,j]* may be different from *SI[j,i]*'; hence, *SI* is not a symmetric matrix even when *adjacency* is symmetric.

Return type (N x N) ndarray

References

1.3.6 brainconn.generative: Generative

Metrics which identify the most important nodes in graphs.

brainconn.generative	Metrics which identify the most important nodes in graphs.
brainconn.generative.generative_model(A, D, ...)	Generates synthetic networks using the models described in Betzel et al.
brainconn.generative.evaluate_generative_model(A, ...)	Generates synthetic networks with parameters provided and evaluates their energy function.

brainconn.generative.generative_model

generative_model(*A*, *D*, *m*, *eta*, *gamma=None*, *model_type='matching'*, *model_var='powerlaw'*, *epsilon=1e-06*, *copy=True*)

Generates synthetic networks using the models described in Betzel et al. (2016) Neuroimage. See this paper for more details.

Succinctly, the probability of forming a connection between nodes u and v is P(u,v) = E(u,v)**eta * K(u,v)**gamma where eta and gamma are hyperparameters, E(u,v) is the euclidean or similar distance measure, and K(u,v) is the algorithm that defines the model.

This describes the power law formulation, an alternative formulation uses the exponential function P(u,v) = exp(E(u,v)*eta) * exp(K(u,v)*gamma)

Parameters

- **A** (*numpy.ndarray*) – Binary network of seed connections
- **D** (*numpy.ndarray*) – Matrix of euclidean distances or other distances between nodes
- **m** (*int*) – Number of connections that should be present in the final synthetic network
- **eta** (*numpy.ndarray*) – A vector describing a range of values to estimate for eta, the hyperparameter describing exponential weighting of the euclidean distance.
- **gamma** (*numpy.ndarray*) – A vector describing a range of values to estimate for theta, the hyperparameter describing exponential weighting of the basis algorithm. If model_type='euclidean' or another distance metric, this can be None.
- **model_type** (*Enum(str)*) –

  euclidean [Uses only euclidean distances to generate connection] probabilities

  neighbors : count of common neighbors matching : matching index, the normalized overlap in neighborhoods clu-avg : Average clustering coefficient clu-min : Minimum clustering coefficient clu-max : Maximum clustering coefficient clu-diff : Difference in clustering coefficient clu-prod : Product of clustering coefficient deg-avg : Average degree deg-min : Minimum degree deg-max : Maximum degree deg-diff : Difference in degree deg-prod : Product of degrees
- **model_var** (*Enum(str)*) – Default value is powerlaw. If so, uses formulation of P(u,v) as described above. Alternate value is exponential. If so, uses P(u,v) = exp(E(u,v)*eta) * exp(K(u,v)*gamma)
- **epsilon** (*float*) – A small positive value added to all P(u,v). The default value is 1e-6

- **copy** (*bool*) – Some algorithms add edges directly to the input matrix. Set this flag to make a copy of the input matrix instead. Defaults to True.

brainconn.generative.evaluate_generative_model

evaluate_generative_model(*A*, *Atgt*, *D*, *eta*, *gamma=None*, *model_type='matching'*, *model_var='powerlaw'*, *epsilon=1e-06*)

Generates synthetic networks with parameters provided and evaluates their energy function. The energy function is defined as in Betzel et al. 2016. Basically it takes the Kolmogorov-Smirnov statistics of 4 network measures; comparing the degree distributions, clustering coefficients, betweenness centrality, and Euclidean distances between connected regions.

The energy is globally low if the synthetic network matches the target. Energy is defined as the maximum difference across the four statistics.

1.3.7 brainconn.modularity: Modularity

Metrics which identify the most important nodes in graphs.

brainconn.modularity	Metrics which identify the most important nodes in graphs.
brainconn.modularity.ci2ls(ci)	Convert from a community index vector to a 2D python list of modules The list is a pure python list, not requiring numpy.
brainconn.modularity.ls2ci(ls[, zeroindexed])	Convert from a 2D python list of modules to a community index vector.
brainconn.modularity.community_louvain(W[, ...])	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes which maximizes the number of within-group edges and minimizes the number of between-group edges.
brainconn.modularity.link_communities(W[, ...])	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes which maximizes the number of within-group edges and minimizes the number of between-group edges.
brainconn.modularity.modularity_dir(A[, ...])	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_finetune_dir(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_finetune_und(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_finetune_und_sign(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.

Continued on next page

Table 7 – continued from previous page

brainconn.modularity.modularity_louvain_dir(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_louvain_und(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_louvain_und_sign(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_probtune_und_sign(W)	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_und(A[, ...])	The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges.
brainconn.modularity.modularity_und_sign(W, ci)	This function simply calculates the signed modularity for a given partition.
brainconn.modularity.partition_distance(cx, cy)	This function quantifies the distance between pairs of community partitions with information theoretic measures.

brainconn.modularity.ci2ls

ci2ls(*ci*)

Convert from a community index vector to a 2D python list of modules The list is a pure python list, not requiring numpy.

Parameters

- **ci** (Nx1 numpy.ndarray) – the community index vector
- **zeroindexed** (*bool*) – If True, ci uses zero-indexing (lowest value is 0). Defaults to False.

Returns **ls** – pure python list with lowest value zero-indexed (regardless of zero-indexing parameter)

Return type listof(list)

brainconn.modularity.ls2ci

ls2ci(*ls*, *zeroindexed=False*)

Convert from a 2D python list of modules to a community index vector. The list is a pure python list, not requiring numpy.

Parameters

- **ls** (*listof(list)*) – pure python list with lowest value zero-indexed (regardless of value of zeroindexed parameter)

- **zeroindexed** (*bool*) – If True, ci uses zero-indexing (lowest value is 0). Defaults to False.

Returns **ci** – community index vector

Return type Nx1 numpy.ndarray

brainconn.modularity.community_louvain

community_louvain(*W*, *gamma=1*, *ci=None*, *B='modularity'*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes which maximizes the number of within-group edges and minimizes the number of between-group edges.

This function is a fast an accurate multi-iterative generalization of the louvain community detection algorithm. This function subsumes and improves upon modularity_[louvain,finetune]_[und,dir]() and additionally allows to optimize other objective functions (includes built-in Potts Model i Hamiltonian, allows for custom objective-function matrices).

Parameters

- **W** (*NxN np.array*) – directed/undirected weighted/binary adjacency matrix
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules. ignored if an objective function matrix is specified.
- **ci** (*Nx1 np.arraylike*) – initial community affiliation vector. default value=None
- **B** (*str | NxN np.arraylike*) – string describing objective function type, or provides a custom NxN objective-function matrix. builtin values

  'modularity' uses Q-metric as objective function 'potts' uses Potts model Hamiltonian. 'negative_sym' symmetric treatment of negative weights 'negative_asym' asymmetric treatment of negative weights
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (*Nx1 np.array*) – final community structure
- **q** (*float*) – optimized q-statistic (modularity only)

brainconn.modularity.link_communities

link_communities(*W*, *type_clustering='single'*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes which maximizes the number of within-group edges and minimizes the number of between-group edges.

This algorithm uncovers overlapping community structure via hierarchical clustering of network links. This algorithm is generalized for weighted/directed/fully-connected networks

Parameters

- **W** (*NxN np.array*) – directed weighted/binary adjacency matrix
- **type_clustering** (*str*) – type of hierarchical clustering. 'single' for single-linkage, 'complete' for complete-linkage. Default value='single'

Returns **M** – nodal community affiliation matrix.

Return type CxN numpy.ndarray

brainconn.modularity.modularity_dir

modularity_dir(*A*, *gamma=1*, *kci=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

Parameters

- **W** (NxN numpy.ndarray) – directed weighted/binary connection matrix
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **kci** (Nx1 numpy.ndarray | None) – starting community structure. If specified, calculates the Q-metric on the community structure giving, without doing any optimzation. Otherwise, if not specified, uses a spectral modularity maximization algorithm.

Returns

- **ci** (Nx1 numpy.ndarray) – optimized community structure
- **Q** (*float*) – maximized modularity metric

Notes

This algorithm is deterministic. The matlab function bearing this name incorrectly disclaims that the outcome depends on heuristics involving a random seed. The louvain method does depend on a random seed, but this function uses a deterministic modularity maximization algorithm.

brainconn.modularity.modularity_finetune_dir

modularity_finetune_dir(*W*, *ci=None*, *gamma=1*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

This algorithm is inspired by the Kernighan-Lin fine-tuning algorithm and is designed to refine a previously detected community structure.

Parameters

- **W** (NxN numpy.ndarray) – directed weighted/binary connection matrix
- **ci** (Nx1 numpy.ndarray | None) – initial community affiliation vector
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector

- **Q** (*float*) – optimized modularity metric

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_finetune_und

modularity_finetune_und(*W*, *ci=None*, *gamma=1*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

This algorithm is inspired by the Kernighan-Lin fine-tuning algorithm and is designed to refine a previously detected community structure.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix
- **ci** (Nx1 numpy.ndarray | None) – initial community affiliation vector
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector
- **Q** (*float*) – optimized modularity metric

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_finetune_und_sign

modularity_finetune_und_sign(*W*, *qtype='sta'*, *gamma=1*, *ci=None*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

This algorithm is inspired by the Kernighan-Lin fine-tuning algorithm and is designed to refine a previously detected community structure.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix with positive and negative weights.

- **qtype** (*str*) – modularity type. Can be 'sta' (default), 'pos', 'smp', 'gja', 'neg'. See Rubinov and Sporns (2011) for a description.
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **ci** (Nx1 numpy.ndarray | None) – initial community affiliation vector
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector
- **Q** (*float*) – optimized modularity metric

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_louvain_dir

modularity_louvain_dir(*W*, *gamma=1*, *hierarchy=False*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

The Louvain algorithm is a fast and accurate community detection algorithm (as of writing). The algorithm may also be used to detect hierarchical community structure.

Parameters

- **W** (NxN numpy.ndarray) – directed weighted/binary connection matrix
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **hierarchy** (*bool*) – Enables hierarchical output. Defalut value=False
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector. If hierarchical output enabled, it is an NxH numpy.ndarray instead with multiple iterations
- **Q** (*float*) – optimized modularity metric. If hierarchical output enabled, becomes an Hx1 array of floats instead.

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_louvain_und

modularity_louvain_und(*W*, *gamma=1*, *hierarchy=False*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

The Louvain algorithm is a fast and accurate community detection algorithm (as of writing). The algorithm may also be used to detect hierarchical community structure.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **hierarchy** (*bool*) – Enables hierarchical output. Defalut value=False
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector. If hierarchical output enabled, it is an NxH numpy.ndarray instead with multiple iterations
- **Q** (*float*) – optimized modularity metric. If hierarchical output enabled, becomes an Hx1 array of floats instead.

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_louvain_und_sign

modularity_louvain_und_sign(*W*, *gamma=1*, *qtype='sta'*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

The Louvain algorithm is a fast and accurate community detection algorithm (at the time of writing).

Use this function as opposed to modularity_louvain_und() only if the network contains a mix of positive and negative weights. If the network contains all positive weights, the output will be equivalent to that of modularity_louvain_und().

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix with positive and negative weights
- **qtype** (*str*) – modularity type. Can be 'sta' (default), 'pos', 'smp', 'gja', 'neg'. See Rubinov and Sporns (2011) for a description.
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.

- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector
- **Q** (*float*) – optimized modularity metric

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_probtune_und_sign

modularity_probtune_und_sign(*W*, *qtype='sta'*, *gamma=1*, *ci=None*, *p=0.45*, *seed=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups. High-modularity degeneracy is the presence of many topologically distinct high-modularity partitions of the network.

This algorithm is inspired by the Kernighan-Lin fine-tuning algorithm and is designed to probabilistically refine a previously detected community by incorporating random node moves into a finetuning algorithm.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix with positive and negative weights
- **qtype** (*str*) – modularity type. Can be 'sta' (default), 'pos', 'smp', 'gja', 'neg'. See Rubinov and Sporns (2011) for a description.
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **ci** (Nx1 numpy.ndarray | None) – initial community affiliation vector
- **p** (*float*) – probability of random node moves. Default value = 0.45
- **seed** (*int | None*) – random seed. default value=None. if None, seeds from /dev/urandom.

Returns

- **ci** (Nx1 numpy.ndarray) – refined community affiliation vector
- **Q** (*float*) – optimized modularity metric

Notes

Ci and Q may vary from run to run, due to heuristics in the algorithm. Consequently, it may be worth to compare multiple runs.

brainconn.modularity.modularity_und

modularity_und(*A*, *gamma=1*, *kci=None*)

The optimal community structure is a subdivision of the network into nonoverlapping groups of nodes in a way that maximizes the number of within-group edges, and minimizes the number of between-group edges. The modularity is a statistic that quantifies the degree to which the network may be subdivided into such clearly delineated groups.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix
- **gamma** (*float*) – resolution parameter. default value=1. Values 0 <= gamma < 1 detect larger modules while gamma > 1 detects smaller modules.
- **kci** (Nx1 numpy.ndarray | None) – starting community structure. If specified, calculates the Q-metric on the community structure giving, without doing any optimzation. Otherwise, if not specified, uses a spectral modularity maximization algorithm.

Returns

- **ci** (Nx1 numpy.ndarray) – optimized community structure
- **Q** (*float*) – maximized modularity metric

Notes

This algorithm is deterministic. The matlab function bearing this name incorrectly disclaims that the outcome depends on heuristics involving a random seed. The louvain method does depend on a random seed, but this function uses a deterministic modularity maximization algorithm.

brainconn.modularity.modularity_und_sign

modularity_und_sign(*W*, *ci*, *qtype='sta'*)

This function simply calculates the signed modularity for a given partition. It does not do automatic partition generation right now.

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted/binary connection matrix with positive and negative weights
- **ci** (Nx1 numpy.ndarray) – community partition
- **qtype** (*str*) – modularity type. Can be 'sta' (default), 'pos', 'smp', 'gja', 'neg'. See Rubinov and Sporns (2011) for a description.

Returns

- **ci** (Nx1 numpy.ndarray) – the partition which was input (for consistency of the API)
- **Q** (*float*) – maximized modularity metric

Notes

uses a deterministic algorithm

brainconn.modularity.partition_distance

partition_distance (*cx*, *cy*)

This function quantifies the distance between pairs of community partitions with information theoretic measures.

Parameters

- **cx** (Nx1 numpy.ndarray) – community affiliation vector X
- **cy** (Nx1 numpy.ndarray) – community affiliation vector Y

Returns

- **VIn** (Nx1 numpy.ndarray) – normalized variation of information
- **MIn** (Nx1 numpy.ndarray) – normalized mutual information

Notes

(Definitions: VIn = [H(X) + H(Y) - 2MI(X,Y)]/log(n) MIn = 2MI(X,Y)/[H(X)+H(Y)]

where H is entropy, MI is mutual information and n is number of nodes)

1.3.8 brainconn.motifs: Motifs

Metrics which identify the most important nodes in graphs.

brainconn.motifs	Metrics which identify the most important nodes in graphs.
brainconn.motifs.find_motif34(m[, n])	This function returns all motif isomorphs for a given motif id and class (3 or 4).
brainconn.motifs.make_motif34lib()	This function generates the motif34lib.mat library required for all other motif computations.
brainconn.motifs.motif3funct_bin(A)	Functional motifs are subsets of connection patterns embedded within anatomical motifs.
brainconn.motifs.motif3funct_wei(W)	Functional motifs are subsets of connection patterns embedded within anatomical motifs.
brainconn.motifs.motif3struct_bin(A)	Structural motifs are patterns of local connectivity.
brainconn.motifs.motif3struct_wei(W)	Structural motifs are patterns of local connectivity.
brainconn.motifs.motif4funct_bin(A)	Functional motifs are subsets of connection patterns embedded within anatomical motifs.
brainconn.motifs.motif4funct_wei(W)	Functional motifs are subsets of connection patterns embedded within anatomical motifs.
brainconn.motifs.motif4struct_bin(A)	Structural motifs are patterns of local connectivity.
brainconn.motifs.motif4struct_wei(W)	Structural motifs are patterns of local connectivity.

brainconn.motifs.find_motif34

find_motif34 (*m*, *n=None*)

This function returns all motif isomorphs for a given motif id and class (3 or 4). The function also returns the motif id for a given motif matrix

1. **Input: Motif_id, e.g. 1 to 13, if class is 3** Motif_class, number of nodes, 3 or 4.

Output: Motif_matrices, all isomorphs for the given motif

2. Input: Motif_matrix e.g. [0 1 0; 0 0 1; 1 0 0] Output Motif_id e.g. 1 to 13, if class is 3

Parameters

- **m** (*int | matrix*) – In use case 1, a motif_id which is an integer. In use case 2, the entire matrix of the motif (e.g. [0 1 0; 0 0 1; 1 0 0])
- **n** (*int | None*) – In use case 1, the motif class, which is the number of nodes. This is either 3 or 4. In use case 2, None.

Returns **M** – In use case 1, returns all isomorphs for the given motif In use case 2, returns the motif_id for the specified motif matrix

Return type numpy.ndarray | int

brainconn.motifs.make_motif34lib

make_motif34lib()

This function generates the motif34lib.mat library required for all other motif computations. Not to be called externally.

brainconn.motifs.motif3funct_bin

motif3funct_bin(*A*)

Functional motifs are subsets of connection patterns embedded within anatomical motifs. Motif frequency is the frequency of occurrence of motifs around a node.

Parameters **A** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **F** (13xN numpy.ndarray) – motif frequency matrix
- **f** (13x1 numpy.ndarray) – motif frequency vector (averaged over all nodes)

brainconn.motifs.motif3funct_wei

motif3funct_wei(*W*)

Functional motifs are subsets of connection patterns embedded within anatomical motifs. Motif frequency is the frequency of occurrence of motifs around a node. Motif intensity and coherence are weighted generalizations of motif frequency.

Parameters **W** (NxN numpy.ndarray) – weighted directed connection matrix (all weights between 0 and 1)

Returns

- **I** (13xN numpy.ndarray) – motif intensity matrix
- **Q** (13xN numpy.ndarray) – motif coherence matrix
- **F** (13xN numpy.ndarray) – motif frequency matrix

Notes

Average intensity and coherence are given by I./F and Q./F.

brainconn.motifs.motif3struct_bin

motif3struct_bin(*A*)

Structural motifs are patterns of local connectivity. Motif frequency is the frequency of occurrence of motifs around a node.

Parameters **A** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **F** (13xN numpy.ndarray) – motif frequency matrix
- **f** (13x1 numpy.ndarray) – motif frequency vector (averaged over all nodes)

brainconn.motifs.motif3struct_wei

motif3struct_wei(*W*)

Structural motifs are patterns of local connectivity. Motif frequency is the frequency of occurrence of motifs around a node. Motif intensity and coherence are weighted generalizations of motif frequency.

Parameters **W** (NxN numpy.ndarray) – weighted directed connection matrix (all weights between 0 and 1)

Returns

- **I** (13xN numpy.ndarray) – motif intensity matrix
- **Q** (13xN numpy.ndarray) – motif coherence matrix
- **F** (13xN numpy.ndarray) – motif frequency matrix

Notes

Average intensity and coherence are given by I./F and Q./F.

brainconn.motifs.motif4funct_bin

motif4funct_bin(*A*)

Functional motifs are subsets of connection patterns embedded within anatomical motifs. Motif frequency is the frequency of occurrence of motifs around a node.

Parameters **A** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **F** (199xN numpy.ndarray) – motif frequency matrix
- **f** (199x1 numpy.ndarray) – motif frequency vector (averaged over all nodes)

brainconn.motifs.motif4funct_wei

motif4funct_wei(*W*)

Functional motifs are subsets of connection patterns embedded within anatomical motifs. Motif frequency is the frequency of occurrence of motifs around a node. Motif intensity and coherence are weighted generalizations of motif frequency.

Parameters **W** (NxN numpy.ndarray) – weighted directed connection matrix (all weights between 0 and 1)

Returns

- **I** (199xN numpy.ndarray) – motif intensity matrix
- **Q** (199xN numpy.ndarray) – motif coherence matrix
- **F** (199xN numpy.ndarray) – motif frequency matrix

Notes

Average intensity and coherence are given by I./F and Q./F.

brainconn.motifs.motif4struct_bin

motif4struct_bin(*A*)

Structural motifs are patterns of local connectivity. Motif frequency is the frequency of occurrence of motifs around a node.

Parameters **A** (NxN numpy.ndarray) – binary directed connection matrix

Returns

- **F** (199xN numpy.ndarray) – motif frequency matrix
- **f** (199x1 numpy.ndarray) – motif frequency vector (averaged over all nodes)

brainconn.motifs.motif4struct_wei

motif4struct_wei(*W*)

Structural motifs are patterns of local connectivity. Motif frequency is the frequency of occurrence of motifs around a node. Motif intensity and coherence are weighted generalizations of motif frequency.

Parameters **W** (NxN numpy.ndarray) – weighted directed connection matrix (all weights between 0 and 1)

Returns

- **I** (199xN numpy.ndarray) – motif intensity matrix
- **Q** (199xN numpy.ndarray) – motif coherence matrix
- **F** (199xN numpy.ndarray) – motif frequency matrix

Notes

Average intensity and coherence are given by I./F and Q./F.

1.3.9 brainconn.physical_connectivity: Physical connectivity

Metrics which identify the most important nodes in graphs.

brainconn.physical_connectivity	Metrics which identify the most important nodes in graphs.

Continued on next page

Table 9 – continued from previous page

brainconn.physical_connectivity.density_dir(CIJ)	Density is the fraction of present connections to possible connections.
brainconn.physical_connectivity.density_und(CIJ)	Density is the fraction of present connections to possible connections.
brainconn.physical_connectivity.rentian_scaling(A, ...)	Physical Rentian scaling (or more simply Rentian scaling) is a property of systems that are cost-efficiently embedded into physical space.

brainconn.physical_connectivity.density_dir

density_dir (*CIJ*)

Density is the fraction of present connections to possible connections.

Parameters **CIJ** (NxN numpy.ndarray) – directed weighted/binary connection matrix

Returns

- **kden** (*float*) – density
- **N** (*int*) – number of vertices
- **k** (*int*) – number of edges

Notes

Assumes CIJ is directed and has no self-connections. Weight information is discarded.

brainconn.physical_connectivity.density_und

density_und (*CIJ*)

Density is the fraction of present connections to possible connections.

Parameters **CIJ** (NxN numpy.ndarray) – undirected (weighted/binary) connection matrix

Returns

- **kden** (*float*) – density
- **N** (*int*) – number of vertices
- **k** (*int*) – number of edges

Notes

Assumes CIJ is undirected and has no self-connections. Weight information is discarded.

brainconn.physical_connectivity.rentian_scaling

rentian_scaling (*A*, *xyz*, *n*)

Physical Rentian scaling (or more simply Rentian scaling) is a property of systems that are cost-efficiently embedded into physical space. It is what is called a "topo-physical" property because it combines information regarding the topological organization of the graph with information about the physical placement of connections. Rentian scaling is present in very large scale integrated circuits, the C. elegans neuronal network, and

morphometric and diffusion-based graphs of human anatomical networks. Rentian scaling is determined by partitioning the system into cubes, counting the number of nodes inside of each cube (N), and the number of edges traversing the boundary of each cube (E). If the system displays Rentian scaling, these two variables N and E will scale with one another in loglog space. The Rent's exponent is given by the slope of log10(E) vs. log10(N), and can be reported alone or can be compared to the theoretical minimum Rent's exponent to determine how cost efficiently the network has been embedded into physical space. Note: if a system displays Rentian scaling, it does not automatically mean that the system is cost-efficiently embedded (although it does suggest that). Validation occurs when comparing to the theoretical minimum Rent's exponent for that system.

Parameters

- **A** (NxN `numpy.ndarray`) – unweighted, binary, symmetric adjacency matrix
- **xyz** (Nx3 `numpy.ndarray`) – vector of node placement coordinates
- **n** (*int*) – Number of partitions to compute. Each partition is a data point; you want a large enough number to adequately compute Rent's exponent.

Returns

- **N** (Mx1 `numpy.ndarray`) – Number of nodes in each of the M partitions
- **E** (Mx1 `numpy.ndarray`)

Notes

Subsequent Analysis: Rentian scaling plots are then created by: figure; loglog(E,N,'*'); To determine the Rent's exponent, p, it is important not to use partitions which may be affected by boundary conditions. In Bassett et al. 2010 PLoS CB, only partitions with N<M/2 were used in the estimation of the Rent's exponent. Thus, we can define N_prime = N(find(N<M/2)) and E_prime = E(find(N<M/2)). Next we need to determine the slope of Eprime vs. Nprime in loglog space, which is the Rent's exponent. There are many ways of doing this with more or less statistical rigor. Robustfit in MATLAB is one such option:

[b,stats] = robustfit(log10(N_prime),log10(E_prime))

Then the Rent's exponent is b(1,2) and the standard error of the estimation is given by stats.se(1,2).

Note: n=5000 was used in Bassett et al. 2010 in PLoS CB.

1.3.10 `brainconn.reference`: Reference

Metrics which identify the most important nodes in graphs.

brainconn.reference	Metrics which identify the most important nodes in graphs.
brainconn.reference.latmio_dir_connected(R, itr)	This function "latticizes" a directed network, while preserving the in- and out-degree distributions.
brainconn.reference.latmio_dir(R, itr[, D])	This function "latticizes" a directed network, while preserving the in- and out-degree distributions.
brainconn.reference.latmio_und_connected(R, itr)	This function "latticizes" an undirected network, while preserving the degree distribution.
brainconn.reference.latmio_und(R, itr[, D])	This function "latticizes" an undirected network, while preserving the degree distribution.

Continued on next page

Table 10 – continued from previous page

brainconn.reference.makeevenCIJ(n, k, sz_cl)	This function generates a random, directed network with a specified number of fully connected modules linked together by evenly distributed remaining random connections.
brainconn.reference.makefractalCIJ(mx_lvl, ...)	This function generates a directed network with a hierarchical modular organization.
brainconn.reference.makerandCIJdegreesfixed(...)	This function generates a directed random network with a specified in-degree and out-degree sequence.
brainconn.reference.makerandCIJ_dir(n, k)	This function generates a directed random network
brainconn.reference.makerandCIJ_und(n, k)	This function generates an undirected random network
brainconn.reference.makeringlatticeCIJ(n, k)	This function generates a directed lattice network with toroidal boundary counditions (i.e.
brainconn.reference.maketoeplitzCIJ(n, k, s)	This function generates a directed network with a Gaussian drop-off in edge density with increasing distance from the main diagonal.
brainconn.reference.null_model_dir_sign(W[, ...])	This function randomizes an directed network with positive and negative weights, while preserving the degree and strength distributions.
brainconn.reference.null_model_und_sign(W[, ...])	This function randomizes an undirected network with positive and negative weights, while preserving the degree and strength distributions.
brainconn.reference.randmio_dir(R, itr)	This function randomizes a directed network, while preserving the in- and out-degree distributions.
brainconn.reference.randmio_dir_connected(R, itr)	This function randomizes a directed network, while preserving the in- and out-degree distributions.
brainconn.reference.randmio_dir_signed(R, itr)	This function randomizes a directed weighted network with positively and negatively signed connections, while preserving the positive and negative degree distributions.
brainconn.reference.randmio_und(R, itr)	This function randomizes an undirected network, while preserving the degree distribution.
brainconn.reference.randmio_und_connected(R, itr)	This function randomizes an undirected network, while preserving the degree distribution.
brainconn.reference.randmio_und_signed(R, itr)	This function randomizes an undirected weighted network with positive and negative weights, while simultaneously preserving the degree distribution of positive and negative weights.
brainconn.reference.randomize_graph_partial_und(A, ...)	A = RANDOMIZE_GRAPH_PARTIAL_UND(A,B,MAXSWAP) takes adjacency matrices A and B and attempts to randomize matrix A by performing MAXSWAP rewirings.
brainconn.reference.randomizer_bin_und(R, alpha)	This function randomizes a binary undirected network, while preserving the degree distribution.

brainconn.reference.latmio_dir_connected

latmio_dir_connected(*R*, *itr*, *D=None*)

This function "latticizes" a directed network, while preserving the in- and out-degree distributions. In weighted networks, the function preserves the out-strength but not the in-strength distributions. The function also ensures that the randomized network maintains connectedness, the ability for every node to reach every other node in

the network. The input network for this function must be connected.

Parameters

- **R** (NxN numpy.ndarray) – directed binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.
- **D** (numpy.ndarray | None) – distance-to-diagonal matrix. Defaults to the actual distance matrix if not specified.

Returns

- **Rlatt** (NxN numpy.ndarray) – latticized network in original node ordering
- **Rrp** (NxN numpy.ndarray) – latticized network in node ordering used for latticization
- **ind_rp** (Nx1 numpy.ndarray) – node ordering used for latticization
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.latmio_dir

latmio_dir(*R*, *itr*, *D=None*)

This function "latticizes" a directed network, while preserving the in- and out-degree distributions. In weighted networks, the function preserves the out-strength but not the in-strength distributions.

Parameters

- **R** (NxN numpy.ndarray) – directed binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.
- **D** (numpy.ndarray | None) – distance-to-diagonal matrix. Defaults to the actual distance matrix if not specified.

Returns

- **Rlatt** (NxN numpy.ndarray) – latticized network in original node ordering
- **Rrp** (NxN numpy.ndarray) – latticized network in node ordering used for latticization
- **ind_rp** (Nx1 numpy.ndarray) – node ordering used for latticization
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.latmio_und_connected

latmio_und_connected(*R*, *itr*, *D=None*)

This function "latticizes" an undirected network, while preserving the degree distribution. The function does not preserve the strength distribution in weighted networks. The function also ensures that the randomized network maintains connectedness, the ability for every node to reach every other node in the network. The input network for this function must be connected.

Parameters

- **R** (NxN numpy.ndarray) – undirected binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.
- **D** (numpy.ndarray | None) – distance-to-diagonal matrix. Defaults to the actual distance matrix if not specified.

Returns

- **Rlatt** (NxN numpy.ndarray) – latticized network in original node ordering
- **Rrp** (NxN numpy.ndarray) – latticized network in node ordering used for latticization
- **ind_rp** (Nx1 numpy.ndarray) – node ordering used for latticization
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.latmio_und

latmio_und(*R*, *itr*, *D=None*)

This function "latticizes" an undirected network, while preserving the degree distribution. The function does not preserve the strength distribution in weighted networks.

Parameters

- **R** (NxN numpy.ndarray) – undirected binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.
- **D** (numpy.ndarray | None) – distance-to-diagonal matrix. Defaults to the actual distance matrix if not specified.

Returns

- **Rlatt** (NxN numpy.ndarray) – latticized network in original node ordering
- **Rrp** (NxN numpy.ndarray) – latticized network in node ordering used for latticization
- **ind_rp** (Nx1 numpy.ndarray) – node ordering used for latticization
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.makeevenCIJ

makeevenCIJ(*n*, *k*, *sz_cl*)

This function generates a random, directed network with a specified number of fully connected modules linked together by evenly distributed remaining random connections.

Parameters

- **N** (*int*) – number of vertices (must be power of 2)
- **K** (*int*) – number of edges
- **sz_cl** (*int*) – size of clusters (must be power of 2)

Returns **CIJ** – connection matrix

Return type NxN numpy.ndarray

Notes

N must be a power of 2. A warning is generated if all modules contain more edges than K. Cluster size is 2^sz_cl;

brainconn.reference.makefractalCIJ

makefractalCIJ(*mx_lvl*, *E*, *sz_cl*)

This function generates a directed network with a hierarchical modular organization. All modules are fully connected and connection density decays as 1/(E^n), with n = index of hierarchical level.

Parameters

- **mx_lvl** (*int*) – number of hierarchical levels, N = 2^mx_lvl
- **E** (*int*) – connection density fall off per level
- **sz_cl** (*int*) – size of clusters (must be power of 2)

Returns

- **CIJ** (NxN numpy.ndarray) – connection matrix
- **K** (*int*) – number of connections present in output CIJ

brainconn.reference.makerandCIJdegreesfixed

makerandCIJdegreesfixed(*inv*, *outv*)

This function generates a directed random network with a specified in-degree and out-degree sequence.

Parameters

- **inv** (Nx1 numpy.ndarray) – in-degree vector
- **outv** (Nx1 numpy.ndarray) – out-degree vector

Returns **CIJ**

Return type NxN numpy.ndarray

Notes

Necessary conditions include:

length(in) = length(out) = n sum(in) = sum(out) = k in(i), out(i) < n-1 in(i) + out(j) < n+2 in(i) + out(i) < n

No connections are placed on the main diagonal

The algorithm used in this function is not, technically, guaranteed to terminate. If a valid distribution of in and out degrees is provided, this function will find it in bounded time with probability 1-(1/(2*(k^2))). This turns out to be a serious problem when computing infinite degree matrices, but offers good performance otherwise.

brainconn.reference.makerandCIJ_dir

makerandCIJ_dir(*n*, *k*)

This function generates a directed random network

Parameters

- **N** (*int*) – number of vertices
- **K** (*int*) – number of edges

Returns **CIJ** – directed random connection matrix

Return type NxN numpy.ndarray

Notes

no connections are placed on the main diagonal.

brainconn.reference.makerandCIJ_und

makerandCIJ_und(*n*, *k*)

This function generates an undirected random network

Parameters

- **N** (*int*) – number of vertices
- **K** (*int*) – number of edges

Returns **CIJ** – undirected random connection matrix

Return type NxN numpy.ndarray

Notes

no connections are placed on the main diagonal.

brainconn.reference.makeringlatticeCIJ

makeringlatticeCIJ(*n*, *k*)

This function generates a directed lattice network with toroidal boundary counditions (i.e. with ring-like "wrapping around").

Parameters

- **N** (*int*) – number of vertices
- **K** (*int*) – number of edges

Returns **CIJ** – connection matrix

Return type NxN numpy.ndarray

Notes

The lattice is made by placing connections as close as possible to the main diagonal, with wrapping around. No connections are made on the main diagonal. In/Outdegree is kept approx. constant at K/N.

brainconn.reference.maketoeplitzCIJ

maketoeplitzCIJ(*n*, *k*, *s*)

This function generates a directed network with a Gaussian drop-off in edge density with increasing distance from the main diagonal. There are toroidal boundary counditions (i.e. no ring-like "wrapping around").

Parameters

- **N** (*int*) – number of vertices

- **K** (*int*) – number of edges
- **s** (*float*) – standard deviation of toeplitz

Returns **CIJ** – connection matrix

Return type NxN numpy.ndarray

Notes

no connections are placed on the main diagonal.

brainconn.reference.null_model_dir_sign

null_model_dir_sign(*W*, *bin_swaps=5*, *wei_freq=0.1*)

This function randomizes an directed network with positive and negative weights, while preserving the degree and strength distributions. This function calls randmio_dir.m

Parameters

- **W** (NxN numpy.ndarray) – directed weighted connection matrix
- **bin_swaps** (*int*) – average number of swaps in each edge binary randomization. Default value is 5. 0 swaps implies no binary randomization.
- **wei_freq** (*float*) – frequency of weight sorting in weighted randomization. 0<=wei_freq<1. wei_freq == 1 implies that weights are sorted at each step. wei_freq == 0.1 implies that weights sorted each 10th step (faster,

  default value)

  wei_freq == 0 implies no sorting of weights (not recommended)

Returns

- **W0** (NxN numpy.ndarray) – randomized weighted connection matrix
- **R** (*4-tuple of floats*) – Correlation coefficients between strength sequences of input and output connection matrices, rpos_in, rpos_out, rneg_in, rneg_out

Notes

The value of bin_swaps is ignored when binary topology is fully connected (e.g. when the network has no negative weights).

Randomization may be better (and execution time will be slower) for higher values of bin_swaps and wei_freq. Higher values of bin_swaps may enable a more random binary organization, and higher values of wei_freq may enable a more accurate conservation of strength sequences.

R are the correlation coefficients between positive and negative in-strength and out-strength sequences of input and output connection matrices and are used to evaluate the accuracy with which strengths were preserved. Note that correlation coefficients may be a rough measure of strength-sequence accuracy and one could implement more formal tests (such as the Kolmogorov-Smirnov test) if desired.

brainconn.reference.null_model_und_sign

null_model_und_sign (*W*, *bin_swaps=5*, *wei_freq=0.1*)

This function randomizes an undirected network with positive and negative weights, while preserving the degree and strength distributions. This function calls randmio_und.m

Parameters

- **W** (NxN numpy.ndarray) – undirected weighted connection matrix
- **bin_swaps** (*int*) – average number of swaps in each edge binary randomization. Default value is 5. 0 swaps implies no binary randomization.
- **wei_freq** (*float*) – frequency of weight sorting in weighted randomization. 0<=wei_freq<1. wei_freq == 1 implies that weights are sorted at each step. wei_freq == 0.1 implies that weights sorted each 10th step (faster,

  default value)

  wei_freq == 0 implies no sorting of weights (not recommended)

Returns

- **W0** (NxN numpy.ndarray) – randomized weighted connection matrix
- **R** (*4-tuple of floats*) – Correlation coefficients between strength sequences of input and output connection matrices, rpos_in, rpos_out, rneg_in, rneg_out

Notes

The value of bin_swaps is ignored when binary topology is fully connected (e.g. when the network has no negative weights).

Randomization may be better (and execution time will be slower) for higher values of bin_swaps and wei_freq. Higher values of bin_swaps may enable a more random binary organization, and higher values of wei_freq may enable a more accurate conservation of strength sequences.

R are the correlation coefficients between positive and negative strength sequences of input and output connection matrices and are used to evaluate the accuracy with which strengths were preserved. Note that correlation coefficients may be a rough measure of strength-sequence accuracy and one could implement more formal tests (such as the Kolmogorov-Smirnov test) if desired.

brainconn.reference.randmio_dir

randmio_dir (*R*, *itr*)

This function randomizes a directed network, while preserving the in- and out-degree distributions. In weighted networks, the function preserves the out-strength but not the in-strength distributions.

Parameters

- **W** (NxN numpy.ndarray) – directed binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.

Returns

- **R** (NxN numpy.ndarray) – randomized network
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.randmio_dir_connected

randmio_dir_connected(*R*, *itr*)

This function randomizes a directed network, while preserving the in- and out-degree distributions. In weighted networks, the function preserves the out-strength but not the in-strength distributions. The function also ensures that the randomized network maintains connectedness, the ability for every node to reach every other node in the network. The input network for this function must be connected.

Parameters

- **W** (NxN numpy.ndarray) – directed binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.

Returns

- **R** (NxN numpy.ndarray) – randomized network
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.randmio_dir_signed

randmio_dir_signed(*R*, *itr*)

This function randomizes a directed weighted network with positively and negatively signed connections, while preserving the positive and negative degree distributions. In weighted networks by default the function preserves the out-degree strength but not the in-strength distributions

Parameters

- **W** (NxN numpy.ndarray) – directed binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.

Returns

- **R** (NxN numpy.ndarray) – randomized network
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.randmio_und

randmio_und(*R*, *itr*)

This function randomizes an undirected network, while preserving the degree distribution. The function does not preserve the strength distribution in weighted networks.

Parameters

- **W** (NxN numpy.ndarray) – undirected binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.

Returns

- **R** (NxN numpy.ndarray) – randomized network
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.randmio_und_connected

randmio_und_connected(*R*, *itr*)

This function randomizes an undirected network, while preserving the degree distribution. The function does not preserve the strength distribution in weighted networks. The function also ensures that the randomized network maintains connectedness, the ability for every node to reach every other node in the network. The input network for this function must be connected.

NOTE the changes to the BCT matlab function of the same name made in the Jan 2016 release have not been propagated to this function because of substantially decreased time efficiency in the implementation. Expect these changes to be merged eventually.

Parameters

- **W** (NxN numpy.ndarray) – undirected binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.

Returns

- **R** (NxN numpy.ndarray) – randomized network
- **eff** (*int*) – number of actual rewirings carried out

brainconn.reference.randmio_und_signed

randmio_und_signed(*R*, *itr*)

This function randomizes an undirected weighted network with positive and negative weights, while simultaneously preserving the degree distribution of positive and negative weights. The function does not preserve the strength distribution in weighted networks.

Parameters

- **W** (NxN numpy.ndarray) – undirected binary/weighted connection matrix
- **itr** (*int*) – rewiring parameter. Each edge is rewired approximately itr times.

Returns **R** – randomized network

Return type NxN numpy.ndarray

brainconn.reference.randomize_graph_partial_und

randomize_graph_partial_und(*A*, *B*, *maxswap*)

A = RANDOMIZE_GRAPH_PARTIAL_UND(A,B,MAXSWAP) takes adjacency matrices A and B and attempts to randomize matrix A by performing MAXSWAP rewirings. The rewirings will avoid any spots where matrix B is nonzero.

Parameters

- **A** (NxN numpy.ndarray) – undirected adjacency matrix to randomize
- **B** (NxN numpy.ndarray) – mask; edges to avoid
- **maxswap** (*int*) – number of rewirings

Returns **A** – randomized matrix

Return type NxN numpy.ndarray

Notes

1. Graph may become disconnected as a result of rewiring. Always

   important to check.

2. A can be weighted, though the weighted degree sequence will not be

   preserved.

3. A must be undirected.

brainconn.reference.randomizer_bin_und

randomizer_bin_und(*R*, *alpha*)

This function randomizes a binary undirected network, while preserving the degree distribution. The function directly searches for rewirable edge pairs (rather than trying to rewire edge pairs at random), and hence avoids long loops and works especially well in dense matrices.

Parameters

- **A** (NxN numpy.ndarray) – binary undirected connection matrix
- **alpha** (*float*) – fraction of edges to rewire

Returns **R** – randomized network

Return type NxN numpy.ndarray

1.3.11 brainconn.similarity: Similarity

Metrics which identify the most important nodes in graphs.

brainconn.similarity	Metrics which identify the most important nodes in graphs.
brainconn.similarity.corr_flat_dir(a1, a2)	Returns the correlation coefficient between two flattened adjacency matrices.
brainconn.similarity.corr_flat_und(a1, a2)	Returns the correlation coefficient between two flattened adjacency matrices.
brainconn.similarity.dice_pairwise_und(a1, a2)	Calculates pairwise dice similarity for each vertex between two matrices.
brainconn.similarity.edge_nei_overlap_bd(CIJ)	This function determines the neighbors of two nodes that are linked by an edge, and then computes their overlap.
brainconn.similarity.edge_nei_overlap_bu(CIJ)	This function determines the neighbors of two nodes that are linked by an edge, and then computes their overlap.
brainconn.similarity.gtom(adj, nr_steps)	The m-th step generalized topological overlap measure (GTOM) quantifies the extent to which a pair of nodes have similar m-th step neighbors.
brainconn.similarity.matching_ind(CIJ)	For any two nodes u and v, the matching index computes the amount of overlap in the connection patterns of u and v.

Continued on next page

Table 11 – continued from previous page

brainconn.similarity.matching_ind_und(CIJ0)	M0 = MATCHING_IND_UND(CIJ) computes matching index for undirected graph specified by adjacency matrix CIJ.

brainconn.similarity.corr_flat_dir

corr_flat_dir(*a1*, *a2*)

Returns the correlation coefficient between two flattened adjacency matrices. Similarity metric for weighted matrices.

Parameters

- **A1** (NxN numpy.ndarray) – directed matrix 1
- **A2** (NxN numpy.ndarray) – directed matrix 2

Returns **r** – Correlation coefficient describing edgewise similarity of a1 and a2

Return type float

brainconn.similarity.corr_flat_und

corr_flat_und(*a1*, *a2*)

Returns the correlation coefficient between two flattened adjacency matrices. Only the upper triangular part is used to avoid double counting undirected matrices. Similarity metric for weighted matrices.

Parameters

- **A1** (NxN numpy.ndarray) – undirected matrix 1
- **A2** (NxN numpy.ndarray) – undirected matrix 2

Returns **r** – Correlation coefficient describing edgewise similarity of a1 and a2

Return type float

brainconn.similarity.dice_pairwise_und

dice_pairwise_und(*a1*, *a2*)

Calculates pairwise dice similarity for each vertex between two matrices. Treats the matrices as binary and undirected.

Parameters

- **A1** (NxN numpy.ndarray) – Matrix 1
- **A2** (NxN numpy.ndarray) – Matrix 2

Returns **D** – dice similarity vector

Return type Nx1 numpy.ndarray

brainconn.similarity.edge_nei_overlap_bd

edge_nei_overlap_bd(*CIJ*)

This function determines the neighbors of two nodes that are linked by an edge, and then computes their overlap. Connection matrix must be binary and directed. Entries of 'EC' that are 'inf' indicate that no edge is present.

Entries of 'EC' that are 0 denote "local bridges", i.e. edges that link completely non-overlapping neighborhoods. Low values of EC indicate edges that are "weak ties".

If CIJ is weighted, the weights are ignored. Neighbors of a node can be linked by incoming, outgoing, or reciprocal connections.

Parameters **CIJ** (NxN numpy.ndarray) – directed binary/weighted connection matrix

Returns

- **EC** (NxN numpy.ndarray) – edge neighborhood overlap matrix
- **ec** (Kx1 numpy.ndarray) – edge neighborhood overlap per edge vector
- **degij** (NxN numpy.ndarray) – degrees of node pairs connected by each edge

brainconn.similarity.edge_nei_overlap_bu

edge_nei_overlap_bu(*CIJ*)

This function determines the neighbors of two nodes that are linked by an edge, and then computes their overlap. Connection matrix must be binary and directed. Entries of 'EC' that are 'inf' indicate that no edge is present. Entries of 'EC' that are 0 denote "local bridges", i.e. edges that link completely non-overlapping neighborhoods. Low values of EC indicate edges that are "weak ties".

If CIJ is weighted, the weights are ignored.

Parameters **CIJ** (NxN numpy.ndarray) – undirected binary/weighted connection matrix

Returns

- **EC** (NxN numpy.ndarray) – edge neighborhood overlap matrix
- **ec** (Kx1 numpy.ndarray) – edge neighborhood overlap per edge vector
- **degij** (NxN numpy.ndarray) – degrees of node pairs connected by each edge

brainconn.similarity.gtom

gtom(*adj*, *nr_steps*)

The m-th step generalized topological overlap measure (GTOM) quantifies the extent to which a pair of nodes have similar m-th step neighbors. Mth-step neighbors are nodes that are reachable by a path of at most length m.

This function computes the the M x M generalized topological overlap measure (GTOM) matrix for number of steps, numSteps.

Parameters

- **adj** (NxN numpy.ndarray) – connection matrix
- **nr_steps** (*int*) – number of steps

Returns **gt** – GTOM matrix

Return type NxN numpy.ndarray

Notes

When numSteps is equal to 1, GTOM is identical to the topological overlap measure (TOM) from reference [2]. In that case the 'gt' matrix records, for each pair of nodes, the fraction of neighbors the two nodes share in common, where "neighbors" are one step removed. As 'numSteps' is increased, neighbors that are furter out are

considered. Elements of 'gt' are bounded between 0 and 1. The 'gt' matrix can be converted from a similarity to a distance matrix by taking 1-gt.

brainconn.similarity.matching_ind

matching_ind(*CIJ*)

For any two nodes u and v, the matching index computes the amount of overlap in the connection patterns of u and v. Self-connections and u-v connections are ignored. The matching index is a symmetric quantity, similar to a correlation or a dot product.

Parameters **CIJ** (NxN numpy.ndarray) – adjacency matrix

Returns

- **Min** (NxN numpy.ndarray) – matching index for incoming connections
- **Mout** (NxN numpy.ndarray) – matching index for outgoing connections
- **Mall** (NxN numpy.ndarray) – matching index for all connections

Notes

Does not use self- or cross connections for comparison. Does not use connections that are not present in BOTH u and v. All output matrices are calculated for upper triangular only.

brainconn.similarity.matching_ind_und

matching_ind_und(*CIJ0*)

M0 = MATCHING_IND_UND(CIJ) computes matching index for undirected graph specified by adjacency matrix CIJ. Matching index is a measure of similarity between two nodes' connectivity profiles (excluding their mutual connection, should it exist).

Parameters **CIJ** (NxN numpy.ndarray) – undirected adjacency matrix

Returns **M0** – matching index matrix

Return type NxN numpy.ndarray

1.3.12 brainconn.nbs: Network-based statistic

Network-based statistic calculation.

brainconn.nbs	Network-based statistic calculation.
brainconn.nbs.nbs_bct(x, y, thresh[, k, ...])	Performs the NBS for populations X and Y for a t-statistic threshold of alpha.

brainconn.nbs.nbs_bct

nbs_bct(*x*, *y*, *thresh*, *k=1000*, *tail='both'*, *paired=False*, *verbose=False*)

Performs the NBS for populations X and Y for a t-statistic threshold of alpha.

Parameters

- **x** (NxNxP numpy.ndarray) – matrix representing the first population with P subjects.

must be symmetric.

- **y** (NxNxQ numpy.ndarray) – matrix representing the second population with Q subjects. Q need not equal P. must be symmetric.
- **thresh** (float) – minimum t-value used as threshold
- **k** (int, optional) – number of permutations used to estimate the empirical null distribution
- **tail** (*{'both', 'left', 'right'}, optional*) – enables specification of particular alternative hypothesis 'left' : mean population of X < mean population of Y 'right' : mean population of Y < mean population of X 'both' : means are unequal (default)
- **paired** (bool, optional) – use paired sample t-test instead of population t-test. requires both subject populations to have equal N. default value = False
- **verbose** (bool, optional) – print some extra information each iteration. defaults value = False

Returns

- **pval** (Cx1 numpy.ndarray) – A vector of corrected p-values for each component of the networks identified. If at least one p-value is less than alpha, the omnibus null hypothesis can be rejected at alpha significance. The null hypothesis is that the value of the connectivity from each edge has equal mean across the two populations.
- **adj** (IxIxC numpy.ndarray) – an adjacency matrix identifying the edges comprising each component. edges are assigned indexed values.
- **null** (Kx1 numpy.ndarray) – A vector of K sampled from the null distribution of maximal component size.

Notes

The NBS[R5426218f292b-1]_ is a nonparametric statistical test used to isolate the components of an N x N undirected connectivity matrix that differ significantly between two distinct populations. Each element of the connectivity matrix stores a connectivity value and each member of the two populations possesses a distinct connectivity matrix. A component of a connectivity matrix is defined as a set of interconnected edges.

The NBS is essentially a procedure to control the family-wise error rate, in the weak sense, when the null hypothesis is tested independently at each of the N(N-1)/2 edges comprising the undirected connectivity matrix. The NBS can provide greater statistical power than conventional procedures for controlling the family-wise error rate, such as the false discovery rate, if the set of edges at which the null hypothesis is rejected constitues a large component or components.

The NBS comprises fours steps:

1. Perform a two-sample T-test at each edge indepedently to test the hypothesis that the value of connectivity between the two populations come from distributions with equal means.
2. Threshold the T-statistic available at each edge to form a set of suprathreshold edges.
3. Identify any components in the adjacency matrix defined by the set of suprathreshold edges. These are referred to as observed components. Compute the size of each observed component identified; that is, the number of edges it comprises.
4. Repeat K times steps 1-3, each time randomly permuting members of the two populations and storing the size of the largest component identified for each permuation. This yields an empirical estimate of the null distribution of maximal component size. A corrected p-value for each observed component is then calculated using this null distribution.

References

1.3.13 brainconn.utils: Utility functions

Utility functions.

brainconn.utils	Utility functions.
brainconn.utils.matrix	Other utility functions.
brainconn.utils.visualization	Tools for visualizing graphs.
brainconn.utils.misc	Miscellaneous utility functions.

brainconn.utils.matrix

Other utility functions.

Functions

autofix(W[, copy])	Fix a bunch of common problems.
binarize(W[, copy])	Binarizes an input weighted connection matrix.
invert(W[, copy])	Inverts elementwise the weights in an input connection matrix.
normalize(W[, copy])	Normalizes an input weighted connection matrix.
threshold_absolute(W, thr[, copy])	This function thresholds the connectivity matrix by absolute weight magnitude.
threshold_proportional(W, p[, copy])	This function "thresholds" the connectivity matrix by preserving a proportion p (0<p<1) of the strongest weights.
weight_conversion(W, wcm[, copy])	W_bin = weight_conversion(W, 'binarize'); W_nrm = weight_conversion(W, 'normalize'); L = weight_conversion(W, 'lengths');

brainconn.utils.matrix.autofix

autofix(*W*, *copy=True*)

Fix a bunch of common problems. More specifically, remove Inf and NaN, ensure exact binariness and symmetry (i.e. remove floating point instability), and zero diagonal.

Parameters

- **W** (*numpy.ndarray*) – weighted connectivity matrix
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – connectivity matrix with fixes applied

Return type numpy.ndarray

brainconn.utils.matrix.binarize

binarize(*W*, *copy=True*)

Binarizes an input weighted connection matrix. If copy is not set, this function will *modify W in place.*

Parameters

- **W** (NxN numpy.ndarray) – weighted connectivity matrix
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – binary connectivity matrix

Return type NxN numpy.ndarray

brainconn.utils.matrix.invert

invert(*W*, *copy=True*)

Inverts elementwise the weights in an input connection matrix. In other words, change the from the matrix of internode strengths to the matrix of internode distances.

If copy is not set, this function will *modify W in place.*

Parameters

- **W** (numpy.ndarray) – weighted connectivity matrix
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – inverted connectivity matrix

Return type numpy.ndarray

brainconn.utils.matrix.normalize

normalize(*W*, *copy=True*)

Normalizes an input weighted connection matrix. If copy is not set, this function will *modify W in place.*

Parameters

- **W** (numpy.ndarray) – weighted connectivity matrix
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – normalized connectivity matrix

Return type numpy.ndarray

brainconn.utils.matrix.threshold_absolute

threshold_absolute(*W*, *thr*, *copy=True*)

This function thresholds the connectivity matrix by absolute weight magnitude. All weights below the given threshold, and all weights on the main diagonal (self-self connections) are set to 0.

If copy is not set, this function will *modify W in place.*

Parameters

- **W** (numpy.ndarray) – weighted connectivity matrix
- **thr** (*float*) – absolute weight threshold
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – thresholded connectivity matrix

Return type numpy.ndarray

brainconn.utils.matrix.threshold_proportional

threshold_proportional(*W*, *p*, *copy=True*)

This function "thresholds" the connectivity matrix by preserving a proportion p (0<p<1) of the strongest weights. All other weights, and all weights on the main diagonal (self-self connections) are set to 0.

If copy is not set, this function will *modify W in place.*

Parameters

- **W** (numpy.ndarray) – weighted connectivity matrix
- **p** (*float*) – proportional weight threshold (0<p<1)
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – thresholded connectivity matrix

Return type numpy.ndarray

Notes

The proportion of elements set to 0 is a fraction of all elements in the matrix, whether or not they are already 0. That is, this function has the following behavior:

>> x = np.random.random((10,10)) >> x_25 = threshold_proportional(x, .25) >> np.size(np.where(x_25)) #note this double counts each nonzero element 46 >> x_125 = threshold_proportional(x, .125) >> np.size(np.where(x_125)) 22 >> x_test = threshold_proportional(x_25, .5) >> np.size(np.where(x_test)) 46

That is, the 50% thresholding of x_25 does nothing because >=50% of the elements in x_25 are aleady <=0. This behavior is the same as in BCT. Be careful with matrices that are both signed and sparse.

brainconn.utils.matrix.weight_conversion

weight_conversion(*W*, *wcm*, *copy=True*)

W_bin = weight_conversion(W, 'binarize'); W_nrm = weight_conversion(W, 'normalize'); L = weight_conversion(W, 'lengths');

This function may either binarize an input weighted connection matrix, normalize an input weighted connection matrix or convert an input weighted connection matrix to a weighted connection-length matrix.

Binarization converts all present connection weights to 1.

Normalization scales all weight magnitudes to the range [0,1] and should be done prior to computing some weighted measures, such as the weighted clustering coefficient.

Conversion of connection weights to connection lengths is needed prior to computation of weighted distance-based measures, such as distance and betweenness centrality. In a weighted connection network, higher weights are naturally interpreted as shorter lengths. The connection-lengths matrix here is defined as the inverse of the connection-weights matrix.

If copy is not set, this function will *modify W in place.*

Parameters

- **W** (NxN numpy.ndarray) – weighted connectivity matrix
- **wcm** (*str*) – weight conversion command. 'binarize' : binarize weights 'normalize' : normalize weights 'lengths' : convert weights to lengths (invert matrix)
- **copy** (*bool*) – if True, returns a copy of the matrix. Otherwise, modifies the matrix in place. Default value=True.

Returns **W** – connectivity matrix with specified changes

Return type NxN numpy.ndarray

Notes

This function is included for compatibility with BCT. But there are other functions binarize(), normalize() and invert() which are simpler to call directly.

brainconn.utils.visualization

Tools for visualizing graphs.

Functions

adjacency_plot_und(A, coor[, tube])	This function in matlab is a visualization helper which translates an adjacency matrix and an Nx3 matrix of spatial coordinates, and plots a 3D isometric network connecting the undirected unweighted nodes using a specific plotting format.
align_matrices(m1, m2[, dfun, verbose, H, ...])	This function aligns two matrices relative to one another by reordering the nodes in M2.
backbone_wu(CIJ, avgdeg)	The network backbone contains the dominant connections in the network and may be used to aid network visualization.
grid_communities(c)	(X,Y,INDSORT) = GRID_COMMUNITIES(C) takes a vector of community assignments C and returns three output arguments for visualizing the communities.
reorderMAT(m[, H, cost])	This function reorders the connectivity matrix in order to place more edges closer to the diagonal.
reorder_matrix(m1[, cost, verbose, H, Texp, ...])	This function rearranges the nodes in matrix M1 such that the matrix elements are squeezed along the main diagonal.
reorder_mod(A, ci)	This function reorders the connectivity matrix by modular structure and may hence be useful in visualization of modular structure.

Continued on next page

Table 15 – continued from previous page

writetoPAJ(CIJ, fname, directed)	This function writes a Pajek .net file from a numpy matrix

brainconn.utils.visualization.adjacency_plot_und

adjacency_plot_und(*A*, *coor*, *tube=False*)

This function in matlab is a visualization helper which translates an adjacency matrix and an Nx3 matrix of spatial coordinates, and plots a 3D isometric network connecting the undirected unweighted nodes using a specific plotting format. Including the formatted output is not useful at all for bctpy since matplotlib will not be able to plot it in quite the same way.

Instead of doing this, I have included code that will plot the adjacency matrix onto nodes at the given spatial coordinates in mayavi

This routine is basically a less featureful version of the 3D brain in cvu, the connectome visualization utility which I also maintain. cvu uses freesurfer surfaces and annotations to get the node coordinates (rather than leaving them up to the user) and has many other interactive visualization features not included here for the sake of brevity.

There are other similar visualizations in the ConnectomeViewer and the UCLA multimodal connectivity database.

Note that unlike other bctpy functions, this function depends on mayavi.

brainconn.utils.visualization.align_matrices

align_matrices(*m1*, *m2*, *dfun='sqrdiff'*, *verbose=False*, *H=1000000.0*, *Texp=1*, *T0=0.001*, *Hbrk=10*)

This function aligns two matrices relative to one another by reordering the nodes in M2. The function uses a version of simulated annealing.

Parameters

- **M1** (NxN numpy.ndarray) – first connection matrix
- **M2** (NxN numpy.ndarray) – second connection matrix
- **dfun** (*str*) –

  distance metric to use for matching 'absdiff' : absolute difference 'sqrdiff' : squared difference (default) 'cosang' : cosine of vector angle
- **verbose** (*bool*) – print out cost at each iteration. Default False.
- **H** (*int*) – annealing parameter, default value 1e6
- **Texp** (*int*) – annealing parameter, default value 1. Coefficient of H s.t. Texp0=1-Texp/H
- **T0** (*float*) – annealing parameter, default value 1e-3
- **Hbrk** (*int*) – annealing parameter, default value = 10. Coefficient of H s.t. Hbrk0 = H/Hkbr

Returns

- **Mreordered** (NxN numpy.ndarray) – reordered connection matrix M2
- **Mindices** (Nx1 numpy.ndarray) – reordered indices
- **cost** (*float*) – objective function distance between M1 and Mreordered

Notes

Connection matrices can be weighted or binary, directed or undirected. They must have the same number of nodes. M1 can be entered in any node ordering.

Note that in general, the outcome will depend on the initial condition (the setting of the random number seed). Also, there is no good way to determine optimal annealing parameters in advance - these parameters will need to be adjusted "by hand" (particularly H, Texp, T0, and Hbrk). For large and/or dense matrices, it is highly recommended to perform exploratory runs varying the settings of 'H' and 'Texp' and then select the best values.

Based on extensive testing, it appears that T0 and Hbrk can remain unchanged in most cases. Texp may be varied from 1-1/H to 1-10/H, for example. H is the most important parameter - set to larger values as the problem size increases. Good solutions can be obtained for matrices up to about 100 nodes. It is advisable to run this function multiple times and select the solution(s) with the lowest 'cost'.

If the two matrices are related it may be very helpful to pre-align them by reordering along their largest eigenvectors:

> [v,~] = eig(M1); v1 = abs(v(:,end)); [a1,b1] = sort(v1); [v,~] = eig(M2); v2 = abs(v(:,end)); [a2,b2] = sort(v2); [a,b,c] = overlapMAT2(M1(b1,b1),M2(b2,b2),'dfun',1);

Setting 'Texp' to zero cancels annealing and uses a greedy algorithm instead.

brainconn.utils.visualization.backbone_wu

backbone_wu(*CIJ*, *avgdeg*)

The network backbone contains the dominant connections in the network and may be used to aid network visualization. This function computes the backbone of a given weighted and undirected connection matrix CIJ, using a minimum-spanning-tree based algorithm.

Parameters

- **CIJ** (NxN numpy.ndarray) – weighted undirected connection matrix
- **avgdeg** (*int*) – desired average degree of backbone

Returns

- **CIJtree** (NxN numpy.ndarray) – connection matrix of the minimum spanning tree of CIJ
- **CIJclus** (NxN numpy.ndarray) – connection matrix of the minimum spanning tree plus strongest connections up to some average degree 'avgdeg'. Identical to CIJtree if the degree requirement is already met.

Notes

NOTE: nodes with zero strength are discarded. NOTE: CIJclus will have a total average degree exactly equal to

> (or very close to) 'avgdeg'.

NOTE: 'avgdeg' backfill is handled slightly differently than in Hagmann et al 2008.

brainconn.utils.visualization.grid_communities

grid_communities(*c*)

(X,Y,INDSORT) = GRID_COMMUNITIES(C) takes a vector of community assignments C and returns three

output arguments for visualizing the communities. The third is INDSORT, which is an ordering of the vertices so that nodes with the same community assignment are next to one another. The first two arguments are vectors that, when overlaid on the adjacency matrix using the PLOT function, highlight the communities.

Parameters **c** (Nx1 numpy.ndarray) – community assignments

Returns

- **bounds** (*list*) – list containing the communities
- **indsort** (numpy.ndarray) – indices

Notes

Note: This function returns considerably different values than in matlab due to differences between matplotlib and matlab. This function has been designed to work with matplotlib, as in the following example:

ci,_=modularity_und(adj) bounds,ixes=grid_communities(ci) pylab.imshow(adj[np.ix_(ixes,ixes)],interpolation='none',cmap='B
for b in bounds:

pylab.axvline(x=b,color='red') pylab.axhline(y=b,color='red')

Note that I adapted the idea from the matlab function of the same name, and have not tested the functionality extensively.

brainconn.utils.visualization.reorderMAT

reorderMAT(*m*, *H=5000*, *cost='line'*)

This function reorders the connectivity matrix in order to place more edges closer to the diagonal. This often helps in displaying community structure, clusters, etc.

Parameters

- **MAT** (NxN numpy.ndarray) – connection matrix
- **H** (*int*) – number of reordering attempts
- **cost** (*str*) – 'line' or 'circ' for shape of lattice (linear or ring lattice). Default is linear lattice.

Returns

- **MATreordered** (NxN numpy.ndarray) – reordered connection matrix
- **MATindices** (Nx1 numpy.ndarray) – reordered indices
- **MATcost** (*float*) – objective function cost of reordered matrix

Notes

I'm not 100% sure how the algorithms between this and reorder_matrix differ, but this code looks a ton sketchier and might have had some minor bugs in it. Considering reorder_matrix() does the same thing using a well vetted simulated annealing algorithm, just use that. ~rlaplant

brainconn.utils.visualization.reorder_matrix

reorder_matrix(*m1*, *cost='line'*, *verbose=False*, *H=10000.0*, *Texp=10*, *T0=0.001*, *Hbrk=10*)

This function rearranges the nodes in matrix M1 such that the matrix elements are squeezed along the main diagonal. The function uses a version of simulated annealing.

Parameters

- **M1** (NxN numpy.ndarray) – connection matrix weighted/binary directed/undirected
- **cost** (*str*) – 'line' or 'circ' for shape of lattice (linear or ring lattice). Default is linear lattice.
- **verbose** (*bool*) – print out cost at each iteration. Default False.
- **H** (*int*) – annealing parameter, default value 1e6
- **Texp** (*int*) – annealing parameter, default value 1. Coefficient of H s.t. Texp0=1-Texp/H
- **T0** (*float*) – annealing parameter, default value 1e-3
- **Hbrk** (*int*) – annealing parameter, default value = 10. Coefficient of H s.t. Hbrk0 = H/Hkbr

Returns

- **Mreordered** (NxN numpy.ndarray) – reordered connection matrix
- **Mindices** (Nx1 numpy.ndarray) – reordered indices
- **Mcost** (*float*) – objective function cost of reordered matrix

Notes

Note that in general, the outcome will depend on the initial condition (the setting of the random number seed). Also, there is no good way to determine optimal annealing parameters in advance - these paramters will need to be adjusted "by hand" (particularly H, Texp, and T0). For large and/or dense matrices, it is highly recommended to perform exploratory runs varying the settings of 'H' and 'Texp' and then select the best values.

Based on extensive testing, it appears that T0 and Hbrk can remain unchanged in most cases. Texp may be varied from 1-1/H to 1-10/H, for example. H is the most important parameter - set to larger values as the problem size increases. It is advisable to run this function multiple times and select the solution(s) with the lowest 'cost'.

Setting 'Texp' to zero cancels annealing and uses a greedy algorithm instead.

brainconn.utils.visualization.reorder_mod

reorder_mod(*A*, *ci*)

This function reorders the connectivity matrix by modular structure and may hence be useful in visualization of modular structure.

Parameters

- **A** (NxN numpy.ndarray) – binary/weighted connectivity matrix
- **ci** (Nx1 numpy.ndarray) – module affiliation vector

Returns

- **On** (Nx1 numpy.ndarray) – new node order
- **Ar** (NxN numpy.ndarray) – reordered connectivity matrix

brainconn.utils.visualization.writetoPAJ

writetoPAJ(*CIJ*, *fname*, *directed*)

This function writes a Pajek .net file from a numpy matrix

Parameters

- **CIJ** (NxN numpy.ndarray) – adjacency matrix
- **fname** (*str*) – filename
- **directed** (*bool*) – True if the network is directed and False otherwise. The data format may be required to know this for some reason so I am afraid to just use directed as the default value.

brainconn.utils.misc

Miscellaneous utility functions.

Functions

cuberoot(x)	Correctly handle the cube root for negative weights, instead of uselessly crashing as in python or returning the wrong root as in matlab
dummyvar(cis[, return_sparse])	This is an efficient implementation of matlab's "dummyvar" command using sparse matrices.
get_resource_path()	Returns the path to general resources, terminated with separator.
pick_four_unique_nodes_quickly(n)	This is equivalent to np.random.choice(n, 4, replace=False)
teachers_round(x)	Do rounding such that .5 always rounds to 1, and not bankers rounding.

brainconn.utils.misc.cuberoot

cuberoot(*x*)

Correctly handle the cube root for negative weights, instead of uselessly crashing as in python or returning the wrong root as in matlab

brainconn.utils.misc.dummyvar

dummyvar(*cis*, *return_sparse=False*)

This is an efficient implementation of matlab's "dummyvar" command using sparse matrices.

input: partitions, NxM array-like containing M partitions of N nodes into <=N distinct communities

output: dummyvar, an NxR matrix containing R column variables (indicator variables) with N entries, where R is the total number of communities summed across each of the M partitions.

i.e. r = sum((max(len(unique(partitions[i]))) for i in range(m)))

brainconn.utils.misc.get_resource_path

get_resource_path()
Returns the path to general resources, terminated with separator. Resources are kept outside package folder in "datasets". Based on function by Yaroslav Halchenko used in Neurosynth Python package.

brainconn.utils.misc.pick_four_unique_nodes_quickly

pick_four_unique_nodes_quickly(*n*)
This is equivalent to np.random.choice(n, 4, replace=False)

Another fellow suggested np.random.random(n).argpartition(4) which is clever but still substantially slower.

brainconn.utils.misc.teachers_round

teachers_round(*x*)
Do rounding such that .5 always rounds to 1, and not bankers rounding. This is for compatibility with matlab functions, and ease of testing.

Exceptions

BCTParamError

1.4 Example gallery

1.4.1 Degree

Calculate centrality measures

Centrality is a thing with stuff and things.

```
# sphinx_gallery_thumbnail_number = 3
```

Start with the necessary imports

```
import os.path as op

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

import brainconn
```

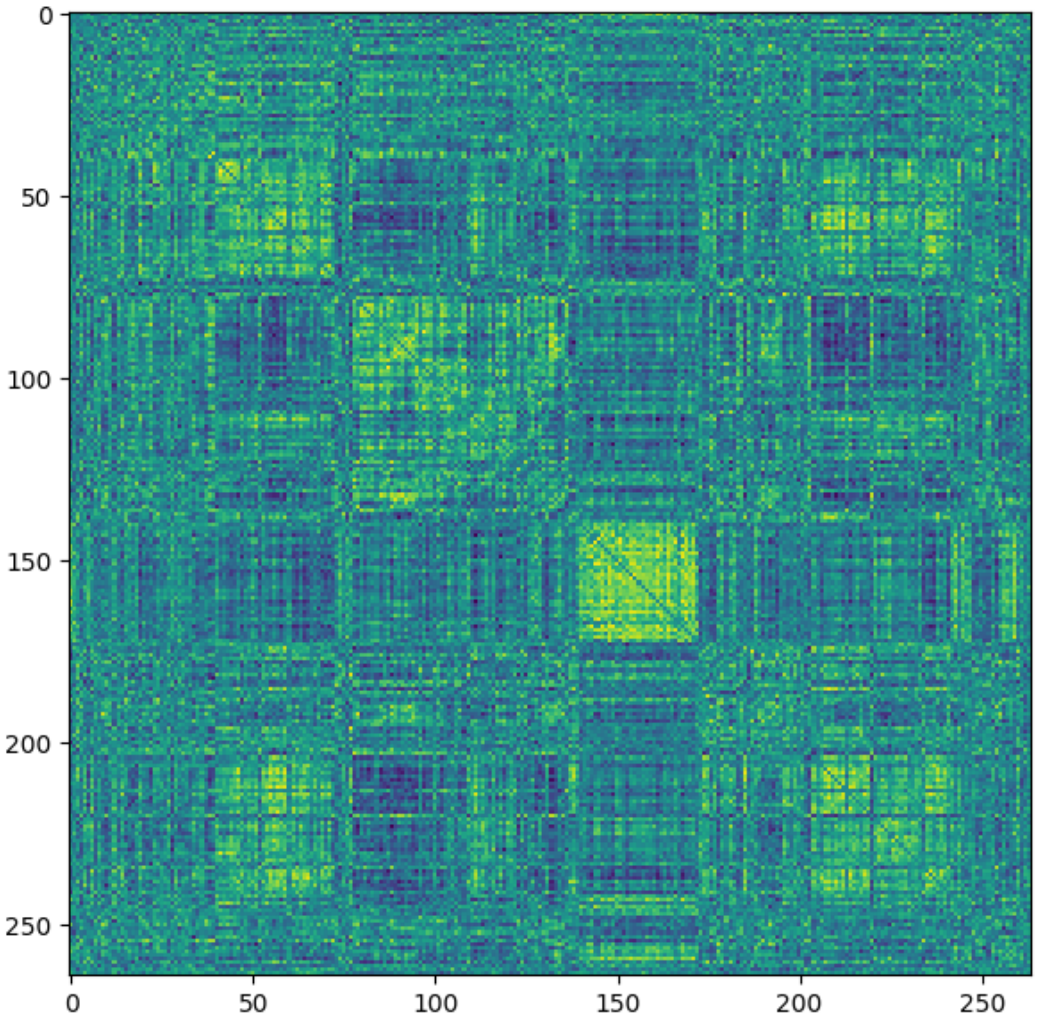
Get some data

```
corr = np.loadtxt(op.join(brainconn.utils.get_resource_path(), 'example_corr.txt'))

# Zero diagonal
adj_wei = corr - np.eye(corr.shape[0])
adj_bin = brainconn.utils.binarize(brainconn.utils.threshold_proportional(adj_wei, 0.
↪2))
```

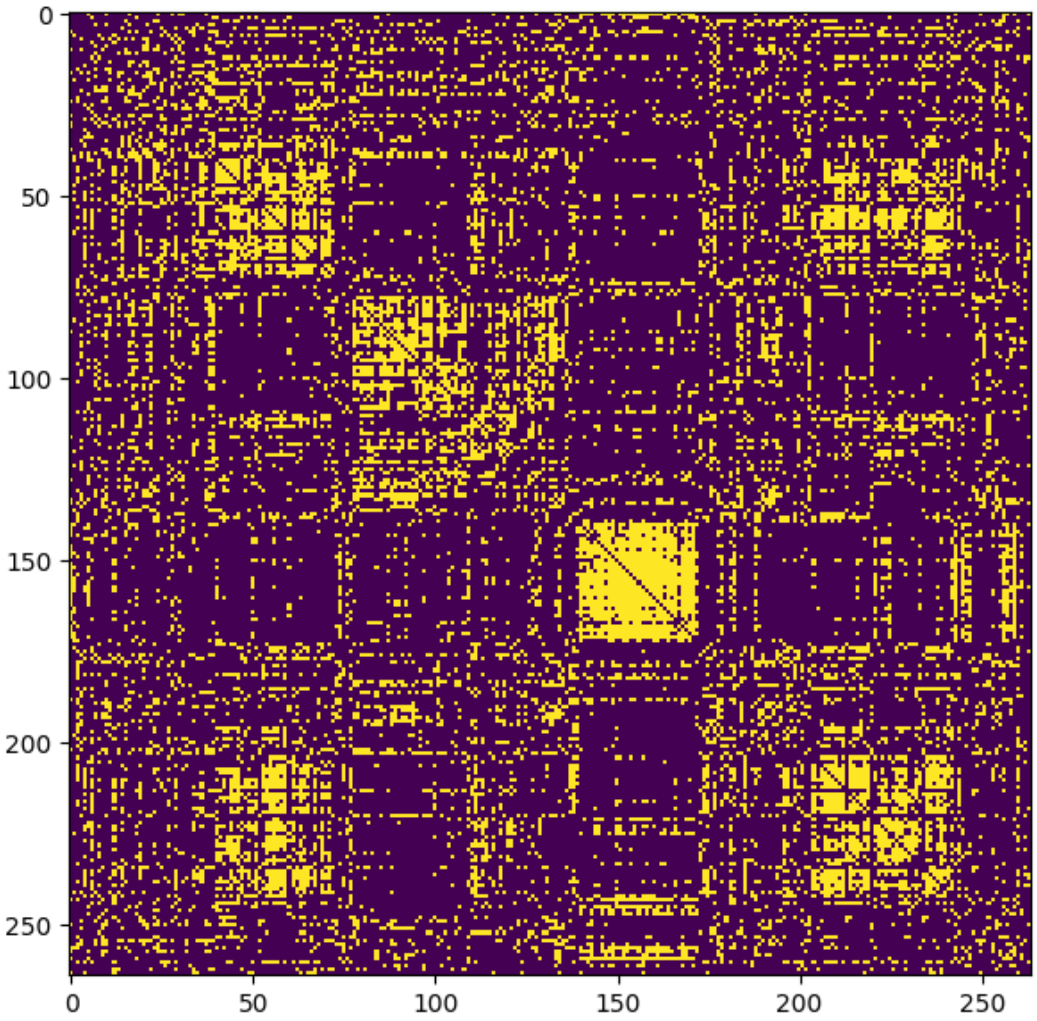
Look at weighted adjacency matrix

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(adj_wei)
fig.show()
```



Look at binary adjacency matrix

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(adj_bin)
fig.show()
```



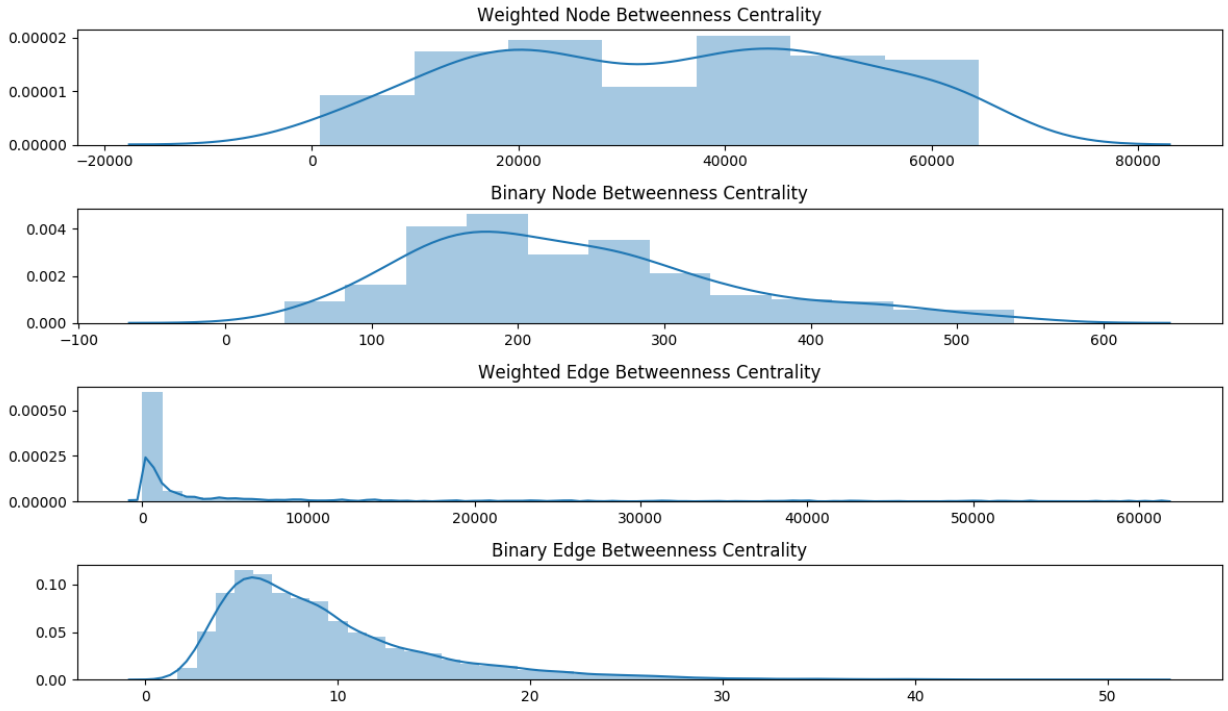
Compute stuff

```
betw_wei = brainconn.centrality.betweenness_wei(adj_wei)
betw_bin = brainconn.centrality.betweenness_bin(adj_bin)
edg_betw_wei = brainconn.centrality.edge_betweenness_wei(adj_wei)[0]
idx = np.triu_indices(edg_betw_wei.shape[0], k=1)
edg_betw_wei = edg_betw_wei[idx]
edg_betw_wei = edg_betw_wei[edg_betw_wei > 0]
edg_betw_bin = brainconn.centrality.edge_betweenness_bin(adj_bin)[0]
idx = np.triu_indices(edg_betw_bin.shape[0], k=1)
edg_betw_bin = edg_betw_bin[idx]
edg_betw_bin = edg_betw_bin[edg_betw_bin > 0]
```

(continues on next page)

(continued from previous page)

```
vals = [betw_wei, betw_bin, edg_betw_wei, edg_betw_bin]
names = ['Weighted Node Betweenness Centrality',
         'Binary Node Betweenness Centrality',
         'Weighted Edge Betweenness Centrality',
         'Binary Edge Betweenness Centrality']
fig, axes = plt.subplots(nrows=4, figsize=(12, 7))
for i in range(4):
    sns.distplot(vals[i], ax=axes[i])
    axes[i].set_title(names[i])
fig.tight_layout()
fig.show()
```



Total running time of the script: (1 minutes 1.232 seconds)

1.4.2 Centrality

Calculate degree measures

Degree is another thing with stuff and things.

```
# sphinx_gallery_thumbnail_number = 5
```

Start with the necessary imports

```
import os.path as op
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from brainconn import degree, utils
```

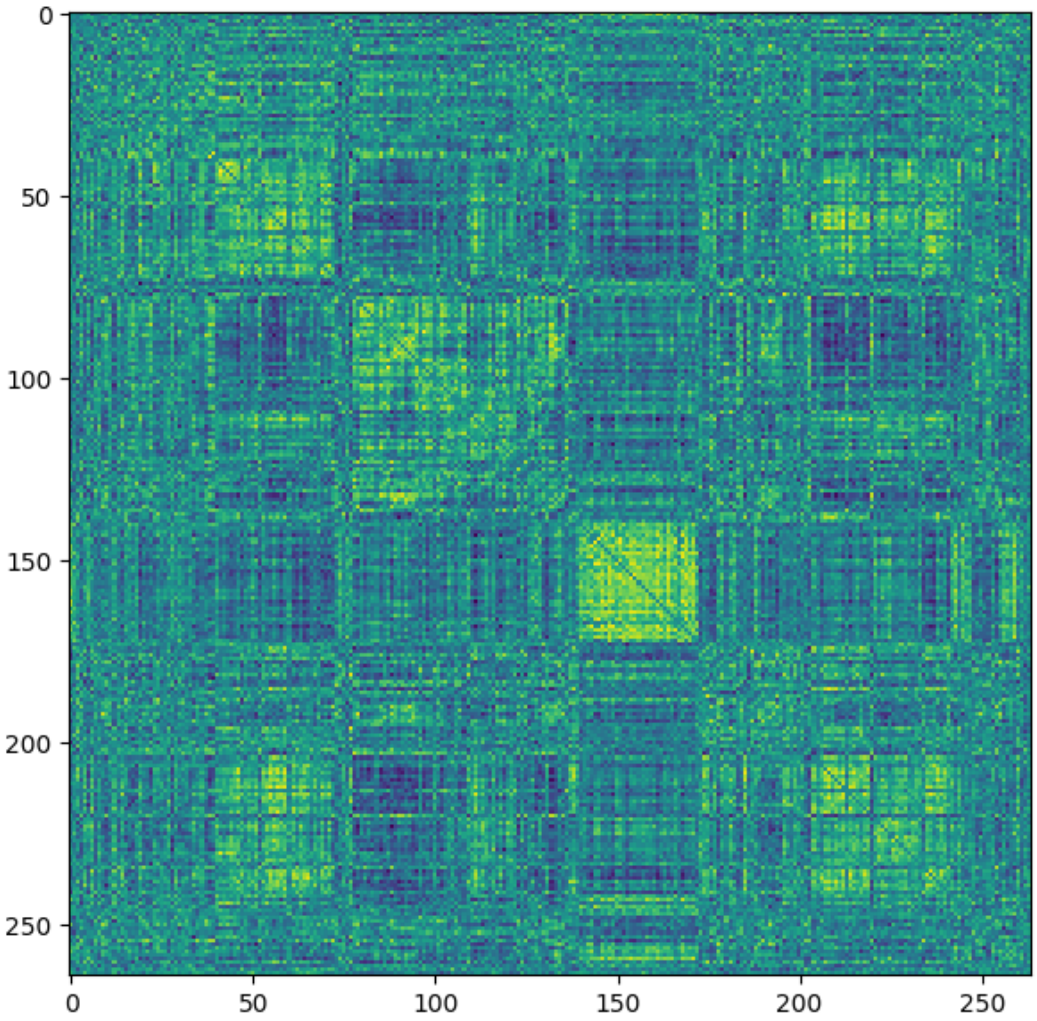
Get some data

```
corr = np.loadtxt(op.join(utils.get_resource_path(), 'example_corr.txt'))

# Zero diagonal
adj_wei = corr - np.eye(corr.shape[0])
adj_wei_dir = adj_wei + (np.triu(adj_wei) / 2)
adj_bin = utils.binarize(utils.threshold_proportional(adj_wei, 0.2))
adj_bin_dir = utils.binarize(utils.threshold_proportional(adj_wei_dir, 0.2))
```

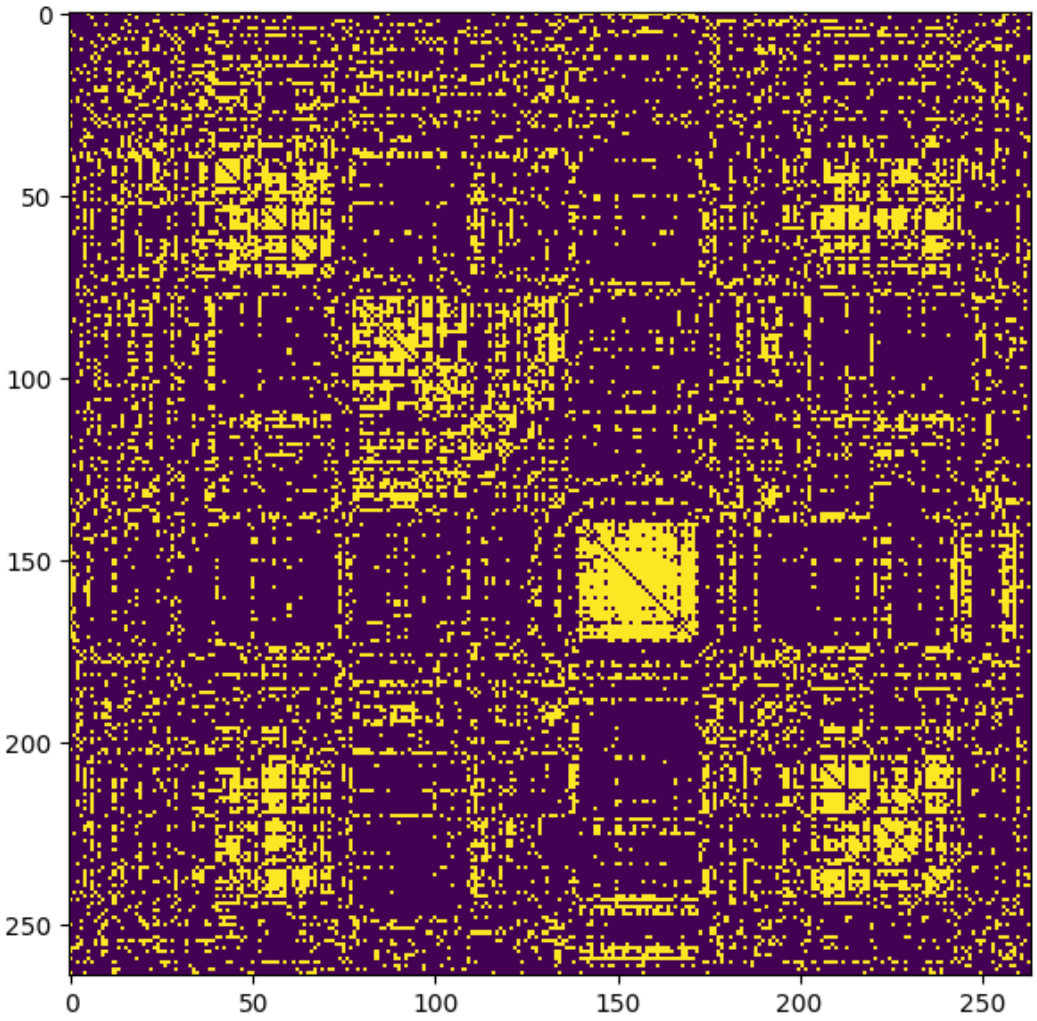
Look at weighted undirected adjacency matrix

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(adj_wei)
fig.show()
```



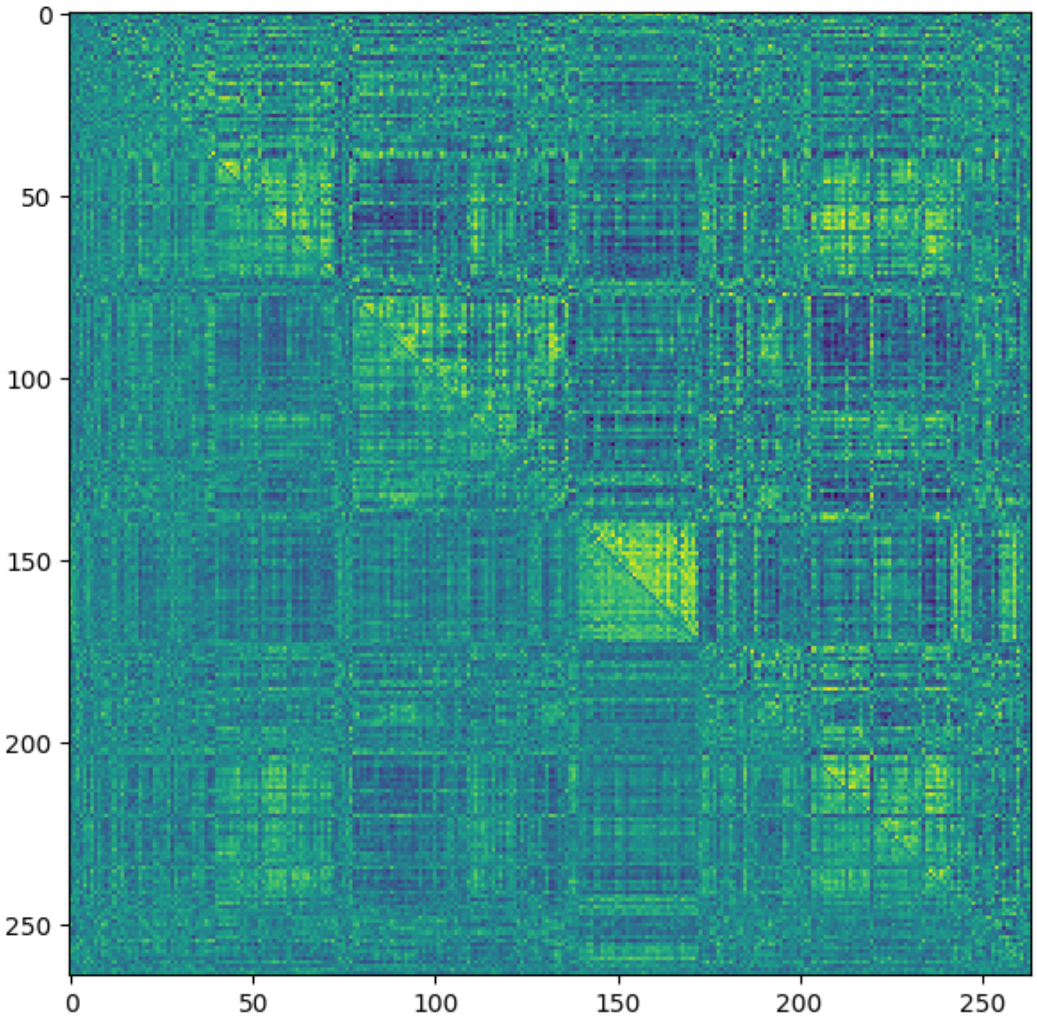
Look at binary undirected adjacency matrix

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(adj_bin)
fig.show()
```



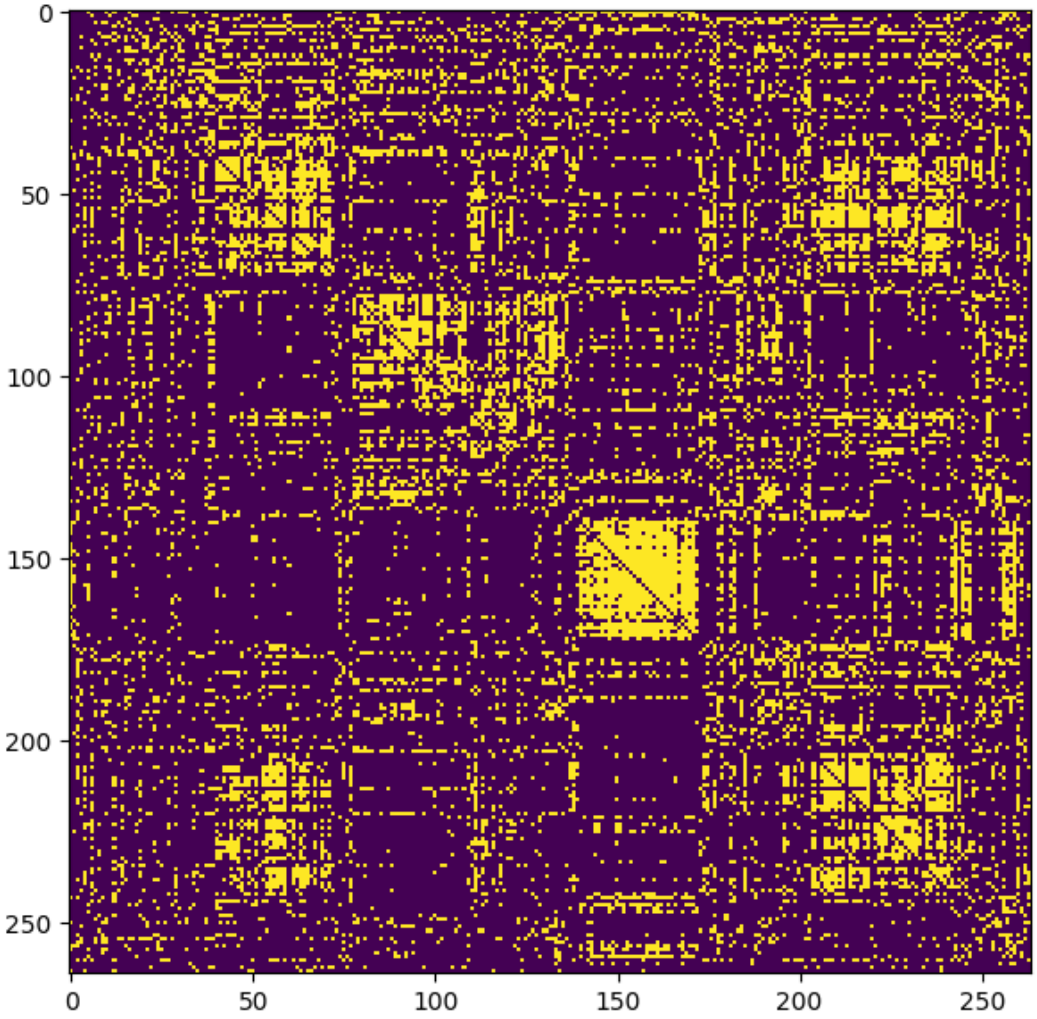
Look at weighted directed adjacency matrix

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(adj_wei_dir)
fig.show()
```



Look at binary directed adjacency matrix

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.imshow(adj_bin_dir)
fig.show()
```



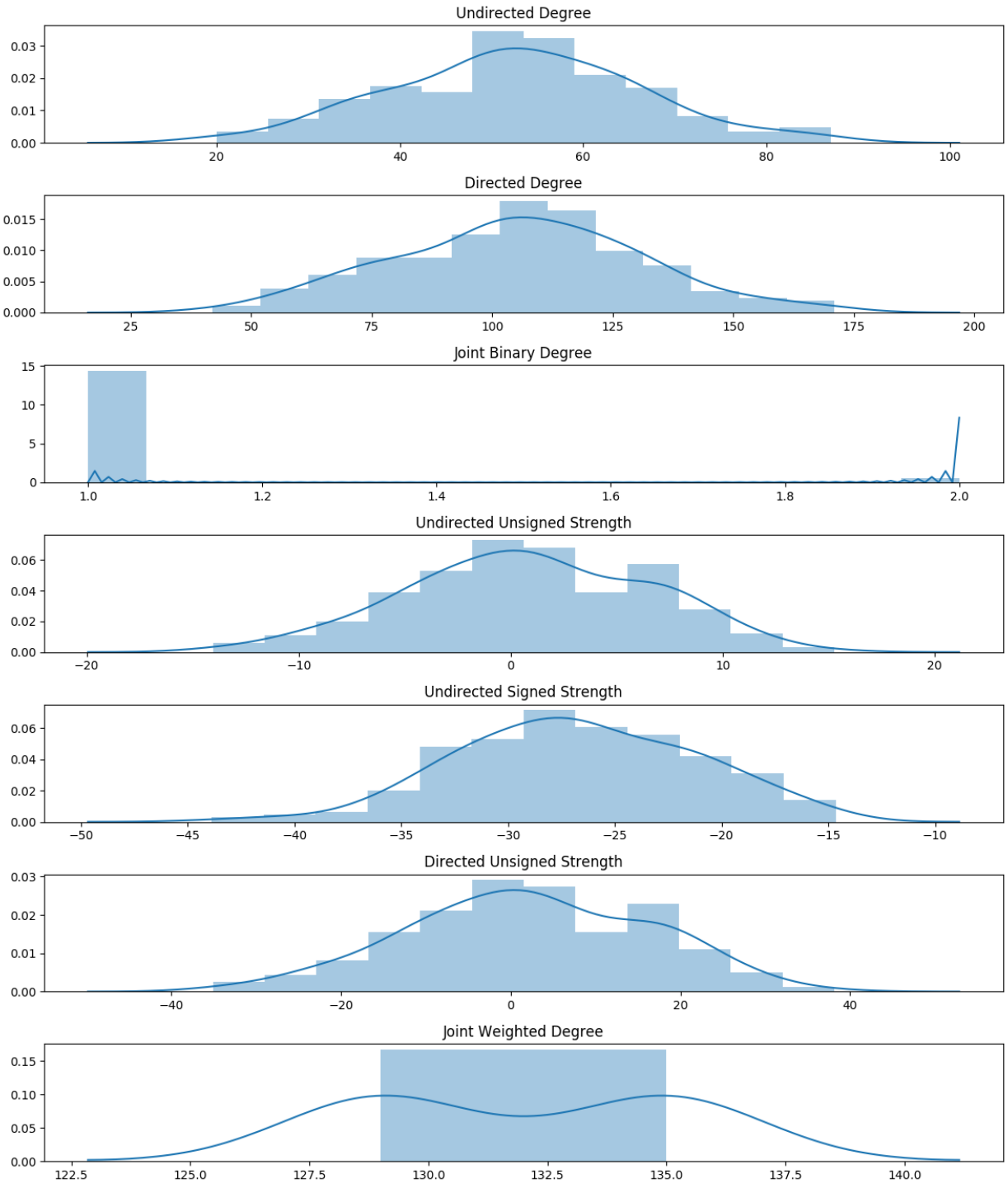
Compute stuff

```
degr_und = degree.degrees_und(adj_bin)
_, _, degr_dir = degree.degrees_dir(adj_bin_dir)
jdeg_bin, _, _, _ = degree.jdegree(adj_bin_dir)
stre_und = degree.strengths_und(adj_wei)
_, stre_und_neg, _, _ = degree.strengths_und_sign(adj_wei)
stre_dir = degree.strengths_dir(adj_wei_dir)
jdeg_wei, _, _, _ = degree.jdegree(adj_wei_dir)
jdeg_bin = jdeg_bin[jdeg_bin > 0]
jdeg_wei = jdeg_wei[jdeg_wei > 0]
```

(continues on next page)

(continued from previous page)

```
vals = [degr_und, degr_dir, jdeg_bin, stre_und, stre_und_neg, stre_dir,
        jdeg_wei]
names = ['Undirected Degree',
         'Directed Degree',
         'Joint Binary Degree',
         'Undirected Unsigned Strength',
         'Undirected Signed Strength',
         'Directed Unsigned Strength',
         'Joint Weighted Degree']
fig, axes = plt.subplots(nrows=len(names), figsize=(12, 2*len(names)))
for i in range(len(names)):
    sns.distplot(vals[i], ax=axes[i])
    axes[i].set_title(names[i])
fig.tight_layout()
fig.show()
```



Total running time of the script: (0 minutes 0.602 seconds)

1.5 History of changes

1.5.1 brainconn 0.0.2 (current)

- Change structure from single `algorithms` submodule to separated submodules: `centrality`, `clustering`, `core`, `degree`, `distance`, `generative`, `modularity`, `motifs`, `physical_connectivity`, `reference`, and `similarity`.

1.5.2 brainconn 0.0.1

- Rename fork of BCT to brainconn for further independent development

1.5.3 BCT 0.4.1

- Refactor code into multiple files
- Fix bug in efficiency_bin
- Fix bugs in modularity_louvain_und
- Fix bugs in `participation_coef_b*`
- Add some test cases

1.5.4 BCT 0.4.0

- Add various new functions from Jan 2015 release of BCT
- Fix various bugs documented in github issues

1.5.5 BCT 0.3.3

- Fix small bug in `latmio_und_connected` causing failure for sparse matrices
- Add non-networkx dependent algorithm to get_components (but less efficient)
- Add an implementation of consensus clustering and fix bug in agreement
- Fix bug causing `clustering_coef_bu` to always return 0
- Remembered to update changelog
- Fix some bugs in `modularity_louvain_dir` and related
- Fix bug in NBS and add optional paired-sample test statistic (sviter)

1.5.6 BCT 0.3.2

- Change several functions including threshold_proportional and binarize have `copy=True` as default argument
- Fix bug in `threshold_proportional` where copying behavior did not work symmetric matrices.
- Fix minor quirk in `threshold_proportional` where `np.round` rounds to nearest even number (optimizes floating point) which is discrepant with BCT
- Add a test suite with some functions

- Fix typo in `rich_club_bu`
- Refactor `x[range(n),range(n)]` to `np.fill_diagonal`
- Fix off-by-one bug in `moduality_[prob/fine]tune_und_sign`

1.5.7 BCT 0.3.1

- Fix bug in NBS
- Fix series of bugs in `null_models`

1.5.8 BCT 0.3

- Added NBS
- Added in all of the new functions from the Dec 2013 release of BCT
- Fixed numerous bugs having to do with indexing errors in modularity
- Fixed several odd bugs with `clustering_coef`, `efficiency`, `distance`
- For the next release, I clearly need a real test suite.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Vargas ER, Wahl LM, Eur Phys J B (2014) 87:1-10

[1] Floyd, R. W. (1962). Algorithm 97: shortest path. Communications of the ACM, 5(6), 345.

[2] Roy, B. (1959). Transitivite et connexite. Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences, 249(2), 216-218.

[3] Warshall, S. (1962). A theorem on boolean matrices. Journal of the ACM (JACM), 9(1), 11-12.

[4] https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

[1] Goni, J., Avena-Koenigsberger, A., de Mendizabal, N. V., van den Heuvel, M. P., Betzel, R. F., & Sporns, O. (2013). Exploring the morphospace of communication efficiency in complex networks. PLoS One, 8(3), e58070.

[1] Goni, J., van den Heuvel, M. P., Avena-Koenigsberger, A., de Mendizabal, N. V., Betzel, R. F., Griffa, A., Hagmann, P., Corominas-Murtra, B., Thiran, J-P., & Sporns, O. (2014). Resting-brain functional connectivity predicted by analytic measures of network communication. Proceedings of the National Academy of Sciences, 111(2), 833-838.

[2] Rosvall, M., Trusina, A., Minnhagen, P., & Sneppen, K. (2005). Networks and cities: An information perspective. Physical Review Letters, 94(2), 028701.

[1] Zalesky A, Fornito A, Bullmore ET (2010) Network-based statistic: Identifying differences in brain networks. NeuroImage. 10.1016/j.neuroimage.2010.06.041

Python Module Index

b

Index

F

G

I

J

K

L

M

N

P

R

S

T

W