BCTPY function arguments and return values

March 8, 2018

This document is simply just an overview of the `Brain Connectivity Toolbox` written in Python (`bct-py`). The document lists all the comments of the functions, and thus gives an overview of which arguments each function takes, and what they return.

Contents

1 Centrality

1.1 betweenness_bin

table of contents

```
def 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 np.ndarray
        binary directed/undirected connection matrix

    BC : Nx1 np.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.
    '''
```

1.2 betweenness_wei

table of contents

```
def 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 np.ndarray
        directed/undirected weighted connection matrix

    Returns
    -------
    BC : Nx1 np.ndarray
        node 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.
    '''
```

1.3 diversity_coef_sign

table of contents

```
def 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 np.ndarray
        undirected connection matrix with positive and negative weights
    ci : Nx1 np.ndarray
        community affiliation vector

    Returns
    -------
    Hpos : Nx1 np.ndarray
        diversity coefficient based on positive connections
    Hneg : Nx1 np.ndarray
        diversity coefficient based on negative connections
    '''
```

1.4 edge_betweenness_bin

table of contents

```
def 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 np.ndarray
        binary directed/undirected connection matrix

    Returns
    -------
    EBC : NxN np.ndarray
        edge betweenness centrality matrix
    BC : Nx1 np.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.
    '''
```

1.5 edge_betweenness_wei

table of contents

```
def 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 np.ndarray
        directed/undirected weighted connection matrix

    Returns
    -------
    EBC : NxN np.ndarray
        edge betweenness centrality matrix
    BC : Nx1 np.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.
    '''
```

1.6 eigenvector_centrality_und

table of contents

```
def 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 np.ndarray
        binary/weighted undirected adjacency matrix

    v : Nx1 np.ndarray
        eigenvector associated with the largest eigenvalue of the matrix
    '''
```

1.7 erange

table of contents

```
def erange(CIJ):
    '''
    Shortcuts are central edges which significantly reduce the
    characteristic path length in the network.

    Parameters
    ----------
    CIJ : NxN np.ndarray
        binary directed connection matrix

    Returns
    -------
    Erange : NxN np.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 np.ndarray
```

```
        entries are ones for shortcut edges
    fs : float
        fractions of shortcuts in the graph

    Follows the treatment of 'shortcuts' by Duncan Watts
    '''
```

1.8 flow_coef_bd

table of contents

```
def 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 np.ndarray
        binary directed connection matrix

    Returns
    -------
    fc : Nx1 np.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
    '''
```

1.9 gateway_coef_sign

table of contents

```
def 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 np.ndarray
        undirected signed connection matrix
    ci : Nx1 np.ndarray
        community affiliation vector
    centrality_type : enum
        'degree' - uses the weighted degree (i.e, node strength)
        'betweenness' - uses the betweenness centrality

    Returns
    -------
    Gpos : Nx1 np.ndarray
        gateway coefficient for positive weights
    Gneg : Nx1 np.ndarray
        gateway coefficient for negative weights

    Reference:
        Vargas ER, Wahl LM, Eur Phys J B (2014) 87:1-10
    '''
```

1.10 kcoreness_centrality_bd

table of contents

```
def 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 np.ndarray
        binary directed connection matrix

    Returns
    -------
    coreness : Nx1 np.ndarray
        node coreness
    kn : int
        size of k-core
    '''
```

1.11 kcoreness_centrality_bu

table of contents

```
def 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 np.ndarray
        binary undirected connection matrix

    Returns
    -------
    coreness : Nx1 np.ndarray
        node coreness
    kn : int
        size of k-core
    '''
```

1.12 module_degree_zscore

table of contents

```
def module_degree_zscore(W, ci, flag=0):
    '''
    The within-module degree z-score is a within-module version of degree
    centrality.

    Parameters
    ----------
    W : NxN np.narray
        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 : Nx1 np.ndarray
        within-module degree Z-score
    '''
```

1.13 pagerank_centrality

table of contents

```
def 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 np.narray
        adjacency matrix
    d : float
        damping factor (see description)
    falff : Nx1 np.ndarray | None
        Initial page rank probability, non-negative values. Default value is
        None. If not specified, a naive bayesian prior is used.

    Returns
    -------
    r : Nx1 np.ndarray
        vectors of page rankings

    Notes
    -----
    Note: The algorithm will work well for smaller matrices (number of
    nodes around 1000 or less)
    '''
```

1.14 participation_coef

table of contents

```
def participation_coef(W, ci, degree='undirected'):
    '''
    Participation coefficient is a measure of diversity of intermodular
    connections of individual nodes.

    Parameters
    ----------
    W : NxN np.ndarray
        binary/weighted directed/undirected connection matrix
    ci : Nx1 np.ndarray
        community affiliation vector
    degree : str
        Flag to describe nature of graph 'undirected': For undirected graphs
                                         'in': Uses the in-degree
                                         'out': Uses the out-degree

    Returns
    -------
    P : Nx1 np.ndarray
        participation coefficient
    '''
```

1.15 subgraph_centrality

table of contents

```
def 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 np.ndarray
        binary adjacency matrix

    Cs : Nx1 np.ndarray
        subgraph centrality
    '''
```

2 Core

2.1 assortativity_bin

table of contents

```
def 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 np.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 : float
        assortativity coefficient

    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.
    '''
```

2.2 assortativity_wei

table of contents

```
def 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 np.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 : float
        assortativity coefficient

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

2.3 core_periphery_dir

table of contents

```
def 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 np.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 np.ndarray
        Initial core structure
    '''
```

2.4 kcore_bd

table of contents

```
def 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 np.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 np.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 np.ndarray
        indices in the order in which they were peeled away during k-core
        decomposition. only returned if peel is specified.
    peellevel : Nx1 np.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).
    '''
```

2.5 kcore_bu

table of contents

```
def 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 np.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 np.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 np.ndarray
        indices in the order in which they were peeled away during k-core
        decomposition. only returned if peel is specified.
    peellevel : Nx1 np.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).
    '''
```

2.6 local_assortativity_wu_sign

table of contents

```
def 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 np.ndarray
        undirected connection matrix with positive and negative weights

    Returns
    -------
    loc_assort_pos : Nx1 np.ndarray
        local assortativity from positive weights
    loc_assort_neg : Nx1 np.ndarray
        local assortativity from negative weights
    '''
```

2.7 rich_club_bd

table of contents

```
def 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 np.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 np.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
    '''
    # definition of degree as used for RC coefficients
    # degree is taken to be the sum of incoming and outgoing connections
```

2.8 rich_club_bu

table of contents

```
def 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 np.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 np.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
    '''
```

2.9 rich_club_wd

table of contents

```
def rich_club_wd(CIJ, klevel=None):
    '''
    Parameters
    ----------
    CIJ : NxN np.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 : Kx1 np.ndarray
        vector of rich-club coefficients for levels 1 to klevel
    '''
```

2.10 rich_club_wu

table of contents

```
def rich_club_wu(CIJ, klevel=None):
    '''
    Parameters
    ----------
    CIJ : NxN np.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 : Kx1 np.ndarray
        vector of rich-club coefficients for levels 1 to klevel
    '''
```

2.11 score_wu

table of contents

```
def 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 np.ndarray
        weighted undirected connection matrix
    s : float
        level of s-core. Note that can take on any fractional value.

    Returns
    -------
    CIJscore : NxN np.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
    '''
```

3 Clustering

3.1 agreement

table of contents

```
def agreement(ci, buffsz=None):
    '''
    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 : MxN np.ndarray
        set of M (possibly degenerate) partitions of N nodes
    buffsz : int | None
        sets buffer size. If not specified, defaults to 1000

    Returns
    -------
    D : NxN np.ndarray
```

```
        agreement matrix
    '''
```

3.2 agreement_weighted

table of contents

```
def 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 np.ndarray
        set of M (possibly degenerate) partitions of N nodes
    wts : Mx1 np.ndarray
        relative weight of each partition

    Returns
    -------
    D : NxN np.ndarray
        weighted agreement matrix
    '''
```

3.3 clustering_coef_bd

table of contents

```
def 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 np.ndarray
        binary directed connection matrix
```

```
    Returns
    -------
    C : Nx1 np.ndarray
        clustering coefficient vector

    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))
    '''
```

3.4 clustering_coef_bu

table of contents

```
def 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 np.ndarray
        binary undirected connection matrix

    Returns
    -------
    C : Nx1 np.ndarray
        clustering coefficient vector
    '''
```

3.5 clustering_coef_wd

table of contents

```
def clustering_coef_wd(W):
    '''
```

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

    Parameters
    ----------
    W : NxN np.ndarray
        weighted directed connection matrix

    Returns
    -------
    C : Nx1 np.ndarray
        clustering coefficient vector

    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.
    '''
```

3.6 clustering_coef_wu

table of contents

```
def clustering_coef_wu(W):
    '''
    The weighted clustering coefficient is the average "intensity" of
    triangles around a node.

    Parameters
    ----------
    W : NxN np.ndarray
        weighted undirected connection matrix

    Returns
    -------
    C : Nx1 np.ndarray
        clustering coefficient vector
    '''
```

3.7 clustering_coef_wu_sign

table of contents

```
def 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 np.ndarray
        weighted undirected connection matrix
    corr_type : enum
        Allowed values are 'default', 'zhang', 'constantini'

    Returns
    -------
    Cpos : Nx1 np.ndarray
        Clustering coefficient vector for positive weights
    Cneg : Nx1 np.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
    '''
```

3.8 consensus_und

table of contents

```
def 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 np.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 : Nx1 np.ndarray
        consensus partition
    '''
```

3.9 get_components

table of contents

```
def 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 np.ndarray
        binary undirected adjacency matrix
    no_depend : Any
        Does nothing, included for backwards compatibility

    Returns
    -------
    comps : Nx1 np.ndarray
        vector of component assignments for each node
    comp_sizes : Mx1 np.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
    '''
```

3.10 get_components_old

table of contents

```
def 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 np.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 np.ndarray
        vector of component assignments for each node
    comp_sizes : Mx1 np.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.
    '''
```

3.11 transitivity_bd

table of contents

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

    Parameters
    ----------
    A : NxN np.ndarray
```

```
    binary directed connection matrix

Returns
-------
T : float
    transitivity scalar

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))
'''
```

3.12 transitivity_bu

table of contents

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

    Parameters
    ----------
    A : NxN np.ndarray
        binary undirected connection matrix

    Returns
    -------
    T : float
        transitivity scalar
    '''
```

3.13 transitivity_wd

table of contents

```
def transitivity_wd(W):
```

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

    Parameters
    ----------
    W : NxN np.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
```

3.14 transitivity_wu

table of contents

```
def transitivity_wu(W):
    '''
    Transitivity is the ratio of 'triangles to triplets' in the network.
    (A classical version of the clustering coefficient).

    Parameters
    ----------
    W : NxN np.ndarray
        weighted undirected connection matrix

    Returns
    -------
    T : int
        transitivity scalar
    '''
```

4 Degree

4.1 degrees_dir

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix

    Returns
    -------
    id : Nx1 np.ndarray
        node in-degree
    od : Nx1 np.ndarray
        node out-degree
    deg : Nx1 np.ndarray
        node degree (in-degree + out-degree)

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

4.2 degrees_und

table of contents

```
def degrees_und(CIJ):
    '''
    Node degree is the number of links connected to the node.

    Parameters
    ----------
    CIJ : NxN np.ndarray
        undirected binary/weighted connection matrix

    Returns
    -------
    deg : Nx1 np.ndarray
        node degree

    Notes
    -----
```

```
    Weight information is discarded.
    '''
```

4.3 jdegree

table of contents

```
def 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 np.ndarray
        directed binary/weighted connnection matrix

    Returns
    -------
    J : ZxZ np.ndarray
        joint degree distribution matrix
        (shifted by one, replicates matlab one-based-indexing)
    J_od : int
        number of vertices with od>id
    J_id : int
        number of vertices with id>od
    J_bl : int
        number of vertices with id==od

    Notes
    -----
    Weights are discarded.
    '''
```

4.4 strengths_dir

table of contents

```
def 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 np.ndarray
        directed weighted connection matrix

    Returns
    -------
    is : Nx1 np.ndarray
        node in-strength
    os : Nx1 np.ndarray
        node out-strength
    str : Nx1 np.ndarray
        node strength (in-strength + out-strength)

    Notes
    -----
    Inputs are assumed to be on the columns of the CIJ matrix.
    '''
```

4.5 strengths_und

table of contents

```
def strengths_und(CIJ):
    '''
    Node strength is the sum of weights of links connected to the node.

    Parameters
    ----------
    CIJ : NxN np.ndarray
        undirected weighted connection matrix

    Returns
    -------
    str : Nx1 np.ndarray
        node strengths
    '''
```

4.6 strengths_und_sign

table of contents

```
def strengths_und_sign(W):
    '''
    Node strength is the sum of weights of links connected to the node.

    Parameters
```

```
    ----------
    W : NxN np.ndarray
        undirected connection matrix with positive and negative weights

    Returns
    -------
    Spos : Nx1 np.ndarray
        nodal strength of positive weights
    Sneg : Nx1 np.ndarray
        nodal strength of positive weights
    vpos : float
        total positive weight
    vneg : float
        total negative weight
    '''
```

5 Distance

5.1 breadthdist

table of contents

```
def 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 np.ndarray
        binary directed/undirected connection matrix

    Returns
    -------
    R : NxN np.ndarray
        binary reachability matrix
    D : NxN np.ndarray
        distance matrix
```

```
    Notes
    -----
    slower but less memory intensive than "reachdist.m".
    '''
```

5.2 breadth

table of contents

```
def breadth(CIJ, source):
    '''
    Implementation of breadth-first search.

    Parameters
    ----------
    CIJ : NxN np.ndarray
        binary directed/undirected connection matrix
    source : int
        source vertex

    Returns
    -------
    distance : Nx1 np.ndarray
        vector of distances between source and ith vertex (0 for source)
    branch : Nx1 np.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'.
    '''
```

5.3 charpath

table of contents

```
def 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 np.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 np.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.
    '''
```

5.4 cycprob

table of contents

```
def 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 np.ndarray
        Path matrix with Pq[i,j,q] = number of paths from i to j of length q.
        Produced by findpaths()
```

```
    Returns
    -------
    fcyc : Qx1 np.ndarray
        fraction of all paths that are cycles for each path length q
    pcyc : Qx1 np.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
    '''
```

5.5 distance_bin

table of contents

```
def 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 np.ndarray
        binary directed/undirected connection matrix

    Returns
    -------
    D : NxN
        distance matrix

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

5.6 distance_wei

table of contents

```
def 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 np.ndarray
        Directed/undirected connection-length matrix.
        NB L is not the adjacency matrix. See below.

    Returns
    -------
    D : NxN np.ndarray
        distance (shortest weighted path) matrix
    B : NxN np.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.
    '''
```

5.7 efficiency_bin

table of contents

```
def 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 np.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 np.ndarray
        local efficiency, only if local=True
    '''
```

5.8 efficiency_wei

table of contents

```
def 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 np.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 np.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
    '''
```

5.9 findpaths

table of contents

```
def 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 np.ndarray
        binary directed/undirected connection matrix
    qmax : int
        maximal path length
    sources : Nx1 np.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 np.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 np.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 np.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.

'''
```

5.10 findwalks

table of contents

```
def 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 np.ndarray
        binary directed/undirected connection matrix

    Returns
    -------
    Wq : NxNxQ np.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 np.ndarray
        walk length distribution as a function of q

    Notes
    -----
```

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

5.11 reachdist

table of contents

```
def 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 np.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 np.ndarray
        binary reachability matrix
    D : NxN np.ndarray
        distance matrix

    Notes
    -----
    faster but more memory intensive than "breadthdist.m".
    '''
```

6 Generative

6.1 generative_model

table of contents

```
def generative_model(A, D, m, eta, gamma=None, model_type='matching',
    model_var='powerlaw', epsilon=1e-6, 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 : np.ndarray
        Binary network of seed connections
    D : np.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 : np.ndarray
        A vector describing a range of values to estimate for eta, the
        hyperparameter describing exponential weighting of the euclidean
        distance.
    gamma : np.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.
    '''
```

6.2 evaluate_generative_model

table of contents

```
def evaluate_generative_model(A, Atgt, D, eta, gamma=None,
    model_type='matching', model_var='powerlaw', epsilon=1e-6):
    '''
    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.
    '''
```

7 Modularity

7.1 ci2ls

table of contents

```
def 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 np.ndarray
        the community index vector
    zeroindexed : bool
```

```
        If True, ci uses zero-indexing (lowest value is 0). Defaults to False.

    Returns
    -------
    ls : listof(list)
        pure python list with lowest value zero-indexed
        (regardless of zero-indexing parameter)
    '''
```

7.2 ls2ci

table of contents

```
def 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 : Nx1 np.ndarray
        community index vector
    '''
```

7.3 community_louvain

table of contents

```
def 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)
    '''
```

7.4 link_communities

table of contents

```
def 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 : CxN np.ndarray
        nodal community affiliation matrix.
    '''
```

7.5 modularity_dir

table of contents

```
def 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 np.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 np.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 np.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.
    '''
```

7.6 modularity_finetune_dir

table of contents

```
def 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 np.ndarray
        directed weighted/binary connection matrix
    ci : Nx1 np.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 np.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.
    '''
```

7.7 modularity_finetune_und

table of contents

```
def 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 np.ndarray
        undirected weighted/binary connection matrix
    ci : Nx1 np.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 np.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.
    '''
```

7.8 modularity_finetune_und_sign

table of contents

```
def 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 np.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 np.ndarray | None
        initial community affiliation vector
    seed : int | None
        random seed. default value=None. if None, seeds from /dev/urandom.

    Returns
    -------
    ci : Nx1 np.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.
    '''
```

7.9 modularity_louvain_dir

table of contents

```
def 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 np.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 np.ndarray
        refined community affiliation vector. If hierarchical output enabled,
        it is an NxH np.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.
    '''
```

7.10 modularity_louvain_und

table of contents

```
def 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 np.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 np.ndarray
        refined community affiliation vector. If hierarchical output enabled,
        it is an NxH np.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.
    '''
```

7.11 modularity_louvain_und_sign

table of contents

```
def 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 np.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 np.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.
    '''
```

7.12 modularity_probtune_und_sign

table of contents

```
def modularity_probtune_und_sign(W, qtype='sta', gamma=1, ci=None, p=.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 np.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 np.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 np.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.
    '''
```

7.13 modularity_und

table of contents

```
def 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 np.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 np.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 np.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.
    '''
```

7.14 modularity_und_sign

table of contents

```
def 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 np.ndarray
        undirected weighted/binary connection matrix with positive and
        negative weights
```

```
    ci : Nx1 np.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 np.ndarray
        the partition which was input (for consistency of the API)
    Q : float
        maximized modularity metric

    Notes
    -----
    uses a deterministic algorithm
    '''
```

7.15 partition_distance

table of contents

```
def partition_distance(cx, cy):
    '''
    This function quantifies the distance between pairs of community
    partitions with information theoretic measures.

    Parameters
    ----------
    cx : Nx1 np.ndarray
        community affiliation vector X
    cy : Nx1 np.ndarray
        community affiliation vector Y

    Returns
    -------
    VIn : Nx1 np.ndarray
        normalized variation of information
    MIn : Nx1 np.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)
    '''
```

8 Motifs

8.1 find_motif34

table of contents

```
def 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 : np.ndarray | int
        In use case 1, returns all isomorphs for the given motif
        In use case 2, returns the motif_id for the specified motif matrix
    '''
```

8.2 motif3funct_bin

table of contents

```
def 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 np.ndarray
        binary directed connection matrix

    Returns
    -------
    F : 13xN np.ndarray
        motif frequency matrix
    f : 13x1 np.ndarray
        motif frequency vector (averaged over all nodes)
    '''
```

8.3 motif3funct_wei

table of contents

```
def 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 np.ndarray
        weighted directed connection matrix (all weights between 0 and 1)

    Returns
    -------
    I : 13xN np.ndarray
        motif intensity matrix
    Q : 13xN np.ndarray
        motif coherence matrix
    F : 13xN np.ndarray
        motif frequency matrix

    Notes
    -----
    Average intensity and coherence are given by I./F and Q./F.
    '''
```

8.4 motif3struct_bin

table of contents

```
def 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 np.ndarray
        binary directed connection matrix

    Returns
    -------
    F : 13xN np.ndarray
        motif frequency matrix
    f : 13x1 np.ndarray
        motif frequency vector (averaged over all nodes)
    '''
```

8.5 motif3struct_wei

table of contents

```
def 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 np.ndarray
        weighted directed connection matrix (all weights between 0 and 1)

    Returns
    -------
    I : 13xN np.ndarray
        motif intensity matrix
    Q : 13xN np.ndarray
        motif coherence matrix
    F : 13xN np.ndarray
        motif frequency matrix

    Notes
```

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

8.6 motif4funct_bin

table of contents

```
def 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 np.ndarray
        binary directed connection matrix

    Returns
    -------
    F : 199xN np.ndarray
        motif frequency matrix
    f : 199x1 np.ndarray
        motif frequency vector (averaged over all nodes)
    '''
```

8.7 motif4funct_wei

table of contents

```
def 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 np.ndarray
        weighted directed connection matrix (all weights between 0 and 1)

    Returns
    -------
    I : 199xN np.ndarray
```

```
        motif intensity matrix
    Q : 199xN np.ndarray
        motif coherence matrix
    F : 199xN np.ndarray
        motif frequency matrix

    Notes
    -----
    Average intensity and coherence are given by I./F and Q./F.
    '''
```

8.8 motif4struct_bin

table of contents

```
def 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 np.ndarray
        binary directed connection matrix

    Returns
    -------
    F : 199xN np.ndarray
        motif frequency matrix
    f : 199x1 np.ndarray
        motif frequency vector (averaged over all nodes)
    '''
```

8.9 motif4struct_wei

table of contents

```
def 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 np.ndarray
```

```
    weighted directed connection matrix (all weights between 0 and 1)

Returns
-------
I : 199xN np.ndarray
    motif intensity matrix
Q : 199xN np.ndarray
    motif coherence matrix
F : 199xN np.ndarray
    motif frequency matrix

Notes
-----
Average intensity and coherence are given by I./F and Q./F.
'''
```

9 Physical connectivity

9.1 density_dir

table of contents

```
def density_dir(CIJ):
    '''
    Density is the fraction of present connections to possible connections.

    Parameters
    ----------
    CIJ : NxN np.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.
    '''
```

9.2 density_und

table of contents

```
def density_und(CIJ):
    '''
    Density is the fraction of present connections to possible connections.

    Parameters
    ----------
    CIJ : NxN np.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.
    '''
```

9.3 rentian_scaling

table of contents

```
def 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 np.ndarray
    unweighted, binary, symmetric adjacency matrix
xyz : Nx3 np.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 np.ndarray
    Number of nodes in each of the M partitions
E : Mx1 np.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.
'''
```

10 Reference

10.1 latmio_dir_connected

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.
    D : np.ndarray | None
        distance-to-diagonal matrix. Defaults to the actual distance matrix
        if not specified.

    Returns
    -------
    Rlatt : NxN np.ndarray
        latticized network in original node ordering
    Rrp : NxN np.ndarray
        latticized network in node ordering used for latticization
    ind_rp : Nx1 np.ndarray
        node ordering used for latticization
    eff : int
        number of actual rewirings carried out
    '''
```

10.2 latmio_dir

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.
    D : np.ndarray | None
        distance-to-diagonal matrix. Defaults to the actual distance matrix
        if not specified.

    Returns
    -------
    Rlatt : NxN np.ndarray
        latticized network in original node ordering
    Rrp : NxN np.ndarray
        latticized network in node ordering used for latticization
    ind_rp : Nx1 np.ndarray
        node ordering used for latticization
    eff : int
        number of actual rewirings carried out
    '''
```

10.3 latmio_und_connected

table of contents

```
def 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 np.ndarray
        undirected binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.
    D : np.ndarray | None
        distance-to-diagonal matrix. Defaults to the actual distance matrix
        if not specified.

    Returns
```

```
    -------
    Rlatt : NxN np.ndarray
        latticized network in original node ordering
    Rrp : NxN np.ndarray
        latticized network in node ordering used for latticization
    ind_rp : Nx1 np.ndarray
        node ordering used for latticization
    eff : int
        number of actual rewirings carried out
    '''
```

10.4 latmio_und

table of contents

```
def 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 np.ndarray
        undirected binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.
    D : np.ndarray | None
        distance-to-diagonal matrix. Defaults to the actual distance matrix
        if not specified.

    Returns
    -------
    Rlatt : NxN np.ndarray
        latticized network in original node ordering
    Rrp : NxN np.ndarray
        latticized network in node ordering used for latticization
    ind_rp : Nx1 np.ndarray
        node ordering used for latticization
    eff : int
        number of actual rewirings carried out
    '''
```

10.5 makeevenCIJ

table of contents

```
def 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 : NxN np.ndarray
        connection matrix

    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;
    '''
```

10.6 makefractalCIJ

table of contents

```
def 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 np.ndarray
        connection matrix
    K : int
        number of connections present in output CIJ
    '''
```

10.7 makerandCIJdegreesfixed

table of contents

```
def makerandCIJdegreesfixed(inv, outv):
    '''
    This function generates a directed random network with a specified
    in-degree and out-degree sequence.

    Parameters
    ----------
    inv : Nx1 np.ndarray
        in-degree vector
    outv : Nx1 np.ndarray
        out-degree vector

    Returns
    -------
    CIJ : NxN np.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.
    '''
```

10.8 makerandCIJ_dir

table of contents

```
def makerandCIJ_dir(n, k):
    '''
    This function generates a directed random network

    Parameters
    ----------
    N : int
        number of vertices
    K : int
        number of edges

    Returns
    -------
    CIJ : NxN np.ndarray
        directed random connection matrix

    Notes
    -----
    no connections are placed on the main diagonal.
    '''
```

10.9 makerandCIJ_und

table of contents

```
def makerandCIJ_und(n, k):
    '''
    This function generates an undirected random network

    Parameters
    ----------
    N : int
        number of vertices
    K : int
        number of edges

    Returns
    -------
    CIJ : NxN np.ndarray
        undirected random connection matrix

    Notes
    -----
```

```
    no connections are placed on the main diagonal.
    '''
```

10.10 makeringlatticeCIJ

table of contents

```
def 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 : NxN np.ndarray
        connection matrix

    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.
    '''
```

10.11 maketoeplitzCIJ

table of contents

```
def 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 : NxN np.ndarray
    connection matrix

Notes
-----
no connections are placed on the main diagonal.
'''
```

10.12 null_model_dir_sign

table of contents

```
def null_model_dir_sign(W, bin_swaps=5, wei_freq=.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 np.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 np.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.
    '''
```

10.13 null_model_und_sign

table of contents

```
def null_model_und_sign(W, bin_swaps=5, wei_freq=.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 np.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 np.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.
    '''
```

10.14 randmio_dir_connected

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    eff : int
        number of actual rewirings carried out
    '''
```

10.15 randmio_dir

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    eff : int
        number of actual rewirings carried out
    '''
```

10.16 randmio_und_connected

table of contents

```
def 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 np.ndarray
```

```
        undirected binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    eff : int
        number of actual rewirings carried out
    '''
```

10.17 randmio_dir_signed

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    eff : int
        number of actual rewirings carried out
    '''
```

10.18 randmio_und

table of contents

```
def 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 np.ndarray
        undirected binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    eff : int
        number of actual rewirings carried out
    '''
```

10.19 randmio_und_signed

table of contents

```
def 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 np.ndarray
        undirected binary/weighted connection matrix
    itr : int
        rewiring parameter. Each edge is rewired approximately itr times.

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    '''
```

10.20 randomize_graph_partial_und

table of contents

```
def 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 np.ndarray
        undirected adjacency matrix to randomize
    B : NxN np.ndarray
        mask; edges to avoid
    maxswap : int
        number of rewirings

    Returns
    -------
    A : NxN np.ndarray
        randomized matrix

    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.
    '''
```

10.21 randomizer_bin_und

table of contents

```
def 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 np.ndarray
        binary undirected connection matrix
    alpha : float
```

```
        fraction of edges to rewire

    Returns
    -------
    R : NxN np.ndarray
        randomized network
    '''
```

11 Similarity

11.1 edge_nei_overlap_bd

table of contents

```
def 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 np.ndarray
        directed binary/weighted connection matrix

    Returns
    -------
    EC : NxN np.ndarray
        edge neighborhood overlap matrix
    ec : Kx1 np.ndarray
        edge neighborhood overlap per edge vector
    degij : NxN np.ndarray
        degrees of node pairs connected by each edge
    '''
```

11.2 edge_nei_overlap_bu

table of contents

```
def 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 np.ndarray
        undirected binary/weighted connection matrix

    Returns
    -------
    EC : NxN np.ndarray
        edge neighborhood overlap matrix
    ec : Kx1 np.ndarray
        edge neighborhood overlap per edge vector
    degij : NxN np.ndarray
        degrees of node pairs connected by each edge
    '''
```

11.3 gtom

table of contents

```
def 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 np.ndarray
        connection matrix
    nr_steps : int
```

```
    number of steps

Returns
-------
gt : NxN np.ndarray
    GTOM matrix

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.
'''
```

11.4 matching_ind

table of contents

```
def 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 np.ndarray
        adjacency matrix

    Returns
    -------
    Min : NxN np.ndarray
        matching index for incoming connections
    Mout : NxN np.ndarray
        matching index for outgoing connections
    Mall : NxN np.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.
    '''
```

11.5 matching_ind_und

table of contents

```
def 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 np.ndarray
        undirected adjacency matrix

    Returns
    -------
    M0 : NxN np.ndarray
        matching index matrix
    '''
```

11.6 dice_pairwise_und

table of contents

```
def dice_pairwise_und(a1, a2):
    '''
    Calculates pairwise dice similarity for each vertex between two
    matrices. Treats the matrices as binary and undirected.

    Paramaters
    ----------
    A1 : NxN np.ndarray
        Matrix 1
    A2 : NxN np.ndarray
        Matrix 2

    Returns
    -------
    D : Nx1 np.ndarray
        dice similarity vector
```

```
    '''
```

11.7 corr_flat_und

table of contents

```
def 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 np.ndarray
        undirected matrix 1
    A2 : NxN np.ndarray
        undirected matrix 2

    Returns
    -------
    r : float
        Correlation coefficient describing edgewise similarity of a1 and a2
    '''
```

11.8 corr_flat_dir

table of contents

```
def corr_flat_dir(a1, a2):
    '''
    Returns the correlation coefficient between two flattened adjacency
    matrices. Similarity metric for weighted matrices.

    Parameters
    ----------
    A1 : NxN np.ndarray
        directed matrix 1
    A2 : NxN np.ndarray
        directed matrix 2

    Returns
    -------
    r : float
        Correlation coefficient describing edgewise similarity of a1 and a2
    '''
```