itng toolbox

Release 0.1.2

Abolfazl Ziaeemehr

Mar 09, 2020

CONTENTS

CHAPTER

ONE

OVERVIEW:

itng (iasbs theoretical neuroscience group toolbox) is a collection of tools for visualising and analysing the time series, measuring the spike synchronies and also graph analysis. This is a collaborative work with complex network and theoretical neuroscience group of Institute of advanced studied in basic science in Zanjan.

1.1 Requirements:

itng project depend on the following packages:

1. numpy
2. scipy
3. networkx
4. python-igraph
5. matplotlib
6. bctpy
7. cloud_sptheme (for documentation)

1.2 Contents:

1.2.1 itng toolbox tutorials

This is a short description about how to use itng toolbox.

Graph generator

networks.Generators.complete_graph()

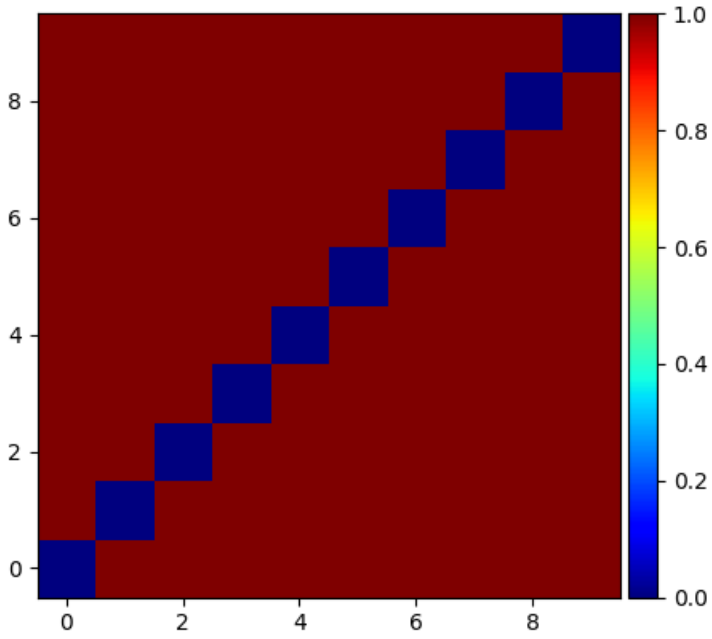
```
import itng
from itng import networks
import networkx as nx
from itng.drawing import Drawing

g = itng.networks.Generators()
G = g.complete_graph(10)
```

(continues on next page)

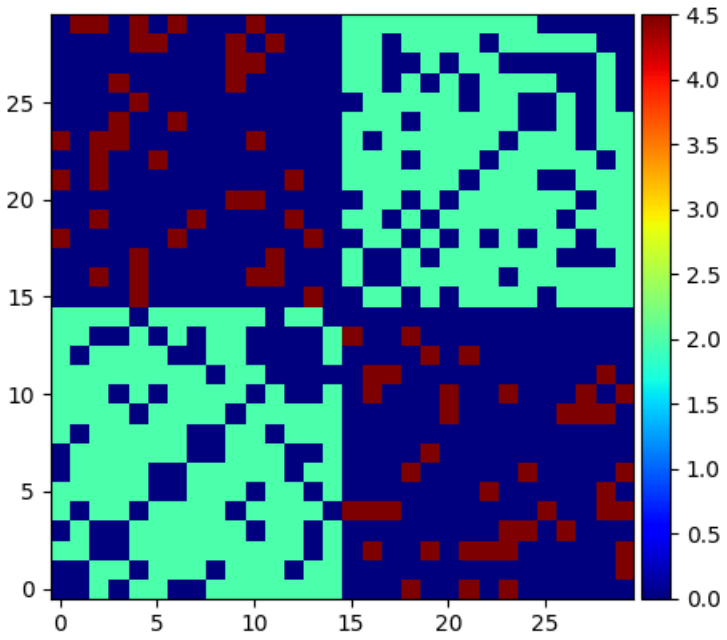
(continued from previous page)

```
Drawing.plot_adjacency_matrix(G,
                              fileName="complete_graph.png",
                              cmap="jet")
```



networks.Generators.modular_graph()

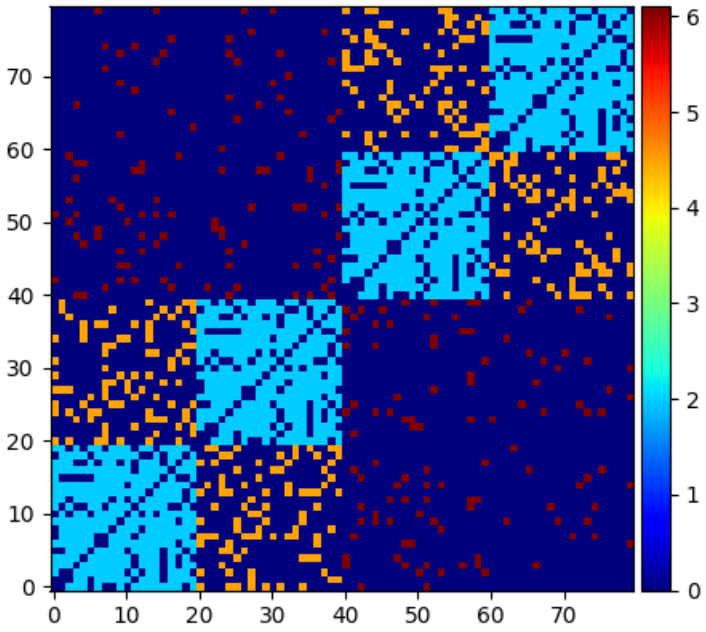
```
G = g.modular_graph(30, 0.7, 0.2, [15] * 2, 2.0, 4.5)
Drawing.plot_adjacency_matrix(G,fileName="modular.png", cmap="jet")
```



networks.Generators.hierarchical_modular_graph()

```
G = g.hierarchical_modular_graph(20, 3, 0.8, 0.25, 1.0, [2.0, 4.5, 6.1])
Drawing.plot_adjacency_matrix(G, fileName="hmn.png", cmap="jet")
print (nx.info(G))

# output
# Name:
# Type: Graph
# Number of nodes: 80
# Number of edges: 886
# Average degree:  22.1500
```



1.2.2 Auto Generated Documentation

class networks.**Generators**(*seed=None*)

generate graphs.

complete_graph(*N*, *create_using=<networkx.classes.graph.Graph object>*)

Return the complete graph with N nodes.

Parameters

- **N** – number of nodes
- **kwargs** – argument pass to ploting the adjacency matrix

Returns networkx graph

gnm_random_graph(*n*, *m*, *directed=False*)

Returns Erdos Renyi network, given size and number of edges

Parameters

- **n** – [int] number of nodes
- **m** – [int] number of edges
- **directed** – bool, optional (default=False) If True, this function returns a directed graph.

Returns networkx graph

See also of *gnp_random_grph()*.

gnp_random_graph(*n*, *p*, *directed=False*)

Returns Erdos Renyi graph, given size and probability

Parameters

- **n** – [int] number of nodes
- **p** – [float] probability of existing an edge
- **directed** – bool, optional (default=False) If True, this function returns a directed graph.

Returns networkx graph

See also of *gnm_random_grph()*.

hierarchical_modular_graph(*n0*, *level*, *prob0*, *prob*, *alpha*, *weights*)
return a connected hierarchical modular network

To construct a hierarchical and modular network (HMN) we started with a modular network of m modules, each having n0 nodes. In each module, nodes were connected with probability P1 (first level of hierarchy). Nodes in different modules were connected with the probability P2 (P2<P1) (second level of hierarchy. We add a copy of the previous network and connect the nodes belonging to these two different sets with a probability P3, where P3 < P2 (third level of hierarchy). Generally speaking, to construct a network with h levels of hierarchy, we repeated the above procedure h-1 times, with the hierarchical level-dependent probabilities, p_l=alpha q^(l-1) (l>1), where alpha is a constant, 0<q<1 and q<P1. The resulting network has 2^(h-1)m0 modules and 2^(h-1) m0 n0 nodes.

Parameters

- **n0** – size of module at level 1
- **level** – number of levels
- **prob0** – probability of nodes in level 1.
- **prob** – [float in [0,1]] this is q in : p_l=alpha q^(l-1)
- **alpha** – [float(default=1.0)] this is alpha in p_l=alpha q^(l-1)
- **delays** – [list of float or int] weights or any other attributes in each level

Returns weighted networkx graph

modular_graph(*N*, *pIn*, *pOut*, *sizes*, *wIn*, *wOut*, ***kwargs*)
returns a modular graphs

Parameters

- **N** – [int] number of nodes
- **pIn** – [int, float or list of numbers] probability of existing an edge inside clusters
- **pOut** – [int, float] probability of existing an edge between clusters
- **sizes** – [list of int] size of clusters in network
- **wIn** – weight of edges inside clusters
- **wOut** – weight of edges between clusters

Returns weighted networkx graph

modular_network_fixed_num_edges(*N*, *n_edges_in*, *n_edges_between*, *sizes*, *wIn*, *wOut*, *verbosity=True*)
Returns a modular graph with given number of edges

Parameters

- **N** – [int] number of nodes
- **n_edges_in** – [int] number of edges inside the clusters

- **n_edges_between** – [int] number of edges between clusters
- **sizes** – [list of int] size of each module
- **wIn** – [float] weight of edges inside the clusters
- **wOut** – [float] weight of edges between the clusters

Returns networkx graph

power_law_graph(*N*, *exp*, *dmin*, *expect_n_edges*, *num_trial=100*, *tol=100*)
generate power law [connected] graph by given number of nodes, edges and exponent.

Parameters

- **N** –
- **exp** – [float] negative exponent in power low graph, typical value is -2.0 to -3.0
- **dmin** – minimum degree of a node.
- **expect_n_edges** – expected number of edges in graph
- **num_trial** – number of trial to try find a connected graph
- **tol** – the tolerance number of edges versus the expected given number of edges.

Returns networkx graph

graphUtility.**binarize**(*adj*, *threshold*)
binarize the given 2d numpy array

Parameters

- **data** – [2d numpy array] given array.
- **threshold** – [float] threshold value.

Returns [2d int numpy array] binarized array.

graphUtility.**calculate_NMI**(*self*, *comm1*, *comm2*, *method='nmi'*)
Compares two community structures

Parameters

- **comm1** – the first community structure as a membership list or as a Clustering object.
- **comm2** – the second community structure as a membership list or as a Clustering object.
- **method** – [string] defaults to ["nmi"] the measure to use. "vi" or "meila" means the variation of information metric of Meila (2003), "nmi" or "danon" means the normalized mutual information as defined by Danon et al (2005), "split-join" means the split-join distance of van Dongen (2000), "rand" means the Rand index of Rand (1971), "adjusted_rand" means the adjusted Rand index of Hubert and Arabie (1985).

Returns [float] the calculated measure.

Reference:

- Meila M: Comparing clusterings by the variation of information. In: Scholkopf B, Warmuth MK (eds). Learning Theory and Kernel Machines: 16th Annual Conference on Computational Learning Theory and 7th Kernel Workship, COLT/Kernel 2003, Washington, DC, USA. Lecture Notes in Computer Science, vol. 2777, Springer, 2003. ISBN: 978-3-540-40720-1.
- Danon L, Diaz-Guilera A, Duch J, Arenas A: Comparing community structure identification. J Stat Mech P09008, 2005.

- van Dongen D: Performance criteria for graph clustering and Markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.
- Rand WM: Objective criteria for the evaluation of clustering methods. J Am Stat Assoc 66(336):846-850, 1971.
- Hubert L and Arabie P: Comparing partitions. Journal of Classification 2:193-218, 1985.

graphUtility.**check_edge_between_clusters**(*e*, *nodes*)
check if given edge is between clusters.

Parameters

- **e** – [int, int] given edge
- **nodes** – nested list (list of list of int) including the index of nodes in each cluster.

Returns [bool] True if edge be between the clusters

graphUtility.**clusters_info_modular**(*G*, *clusters*, *verbosity=True*)
Returns the properties of clusters in a modular network.

Parameters

- **G** – undirected Graph of modular network
- **clusters** – list of list of int inclusing the index of nodes in each cluster.
- **verbosity** – [bool] if True print the results on the screen.

Returns [int, int] total number of edges and number of edges between clusters.

graphUtility.**extract_attributes**(*G*, *attr=None*)
extract the matrix of given atributes from graph

Parameters

- **G** – [networkx graph object]
- **attr** – [string] given attribute

Returns [ndarray] matrix of given attribute

graphUtility.**find_indices_with_distance**(*DiGraph*, *distance*, *source*)
return index of nodes in given distance from source node.

Parameters

- **DiGraph** – networkx directed graph.
- **distance** – [int] given distance from source node.
- **source** – index of given nodes as source, distance of other nodes calculated from this node.

Returns index of nodes in given distance.

graphUtility.**find_leader_node**(*self*, *adj*, *directed=True*)
find the leader node in adj matrix of a directed acyclic graph leader node is the node with in degree of zero.

Parameters **adj** –

Directed [bool] if True consider a directed graph

Returns index of leader node

graphUtility.**lambdan_over_lambda2**(*G*)
calculate the fraction of lambda_n over lambda_2. lambda_i s are the eigen values of laplacian matrix

graphUtility.**multilevel**(*adj_matrix*, *directed=False*, *return_levels=False*)
find communities of given weighted adjacency network

Parameters

- **adjMatrix** – [2d array] adjacency matrix
- **directed** – [bool(default=False)] choose directed or undirected network
- **return_levels** – if True, the communities at each level are returned in a list. If False, only the community structure with the best modularity is returned.

Returns a list of VertexClustering objects, one corresponding to each level (if return_levels is True), or a VertexClustering corresponding to the best modularity.

graphUtility.**print_adj_matrix**(*G*, *file_name=None*, *fmt='%d'*, *binary_file=False*)
print the adjacency matrix of given graph

Parameters

- **G** – networkx graph
- **file_name** – optional(default=None), if given save to file
- **fmt** – optional, format of numbeers in weighted adjacency matrix, "%d" for integer and e.g. "%10.3f", "%g", ... for float numbers.
- **binary_file** – [bool] if True, save npz file in binary format.

graphUtility.**topological_sort**(*self*, *adj_matrix*)
return topologicaly sorted network of given directed acyclic graph.

A topological sort is a nonunique permutation of the nodes such that an edge from u to v implies that u appears before v in the topological sort order.

Parameters **adj_atrix** – [ndarray] given adjacency matrix of directed acyclic graph (DAG).

Returns [ndarray] sorted adjacency matrix

statistics.**cv**(*self*, *ts*, *gids*)
Compute the coefficient of variation.

!todo

statistics.**isi**(*self*, *spiketimes*, *gids*)
calculate interspike interval of given spike train

Parameters

- **spikeTimes** – time of spikes, 1 dimensional array, list or tuple
- **gids** – global ID of neurons

Returns inter spike interval 1d numpy array

statistics.**mean_firing_rate**(*self*, *spiketrain*, *t_start=None*, *t_stop=None*)
Return the firing rate of the spike train.

Parameters

- **spiketrain** – [np.ndarray] the spike times
- **t_start** – [float] The start time to use for the interval
- **t_stop** – [float] The end time to use for the interval

Returns The firing rate of the spiketrain

#!todo

synchrony.**spike_synchrony**(*self*, *ts*, *gids*, *threshold_num_spikes=10*)
calculate spike synchrony.

Parameters

- **ts** – time of spikes 1 dimensional array, list or tuple.
- **gids** – global ID of neurons.
- **threshold_num_spikes** – minimum number of spikes required for calculation of measure.

Returns [float] spike synchrony in [0, 1].

Reference

- Tiesinga, P. H. & Sejnowski, T. J. Rapid temporal modulation of synchrony by competition in cortical interneuron networks. Neural computation 16, 251–275 (2004).

synchrony.**voltage_synchrony**(*self*, *voltages*)
calculate voltage synchrony.

Parameters **voltages** – [ndarray, nested list or nested tuple (number of nodes by number of time steps)] votages of n nodes.

Returns [float] voltage synchrony in [0, 1]

Reference:

- Lim, W. & Kim, S. Y. Coupling-induced spiking coherence in coupled subthreshold neurons. Int. J. Mod. Phys. B 23, 2149–2157 (2009)

randomGenerators.**power_law**(*self*, *N*, *e*, *xmin*, *xmax*)
generate a power law distribution of integers from uniform distribution

Parameters

- **N** – [int] number of data in powerlaw distribution (pwd).
- **e** – [int, float] exponent of the pwd.
- **xmin** – [int] min value in pwd.
- **xmax** – [int] max value in pwd.

Returns [numpy array of int] the power law distribution

becuse the numbers will use as degree of nodes, the sum of degrees should be an even number.

Reference:

- http://mathworld.wolfram.com/RandomNumber.html

1.3 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX