NNGT Documentation

Release 0.5

Tanguy Fardet

February 26, 2016

CHAPTER 1

Introduction

1.1 Yet another graph library?

It is not ;)

This library is based on existing graph libraries (such as graph_tool, and possibly soon SNAP) and acts as a convenient interface to build various networks from efficient and verified algorithms.

Moreover, it also acts as an interface between those graph libraries and the NEST simulator.

1.2 Description

Neural networks are described by four container classes:

- *`Graph`*: container for simple topological graphs with no spatial structure, nor biological properties
- *`SpatialGraph`*: container for spatial graphs without biological properties
- *`Network`*: container for topological graphs with biological properties (to interact with NEST)
- *`SpatialNetwork`*: container with spatial and biological properties (to interact with NEST)

Using these objects, the user can access to a the `graph` attribute, which contains the topological structure of the network (including the connections' type – inhibitory or excitatory – and its weight which is always positive)

Warning: This object should never be directly modified through its methods but rather using those of the four containing classes. The only reason to access this object should be to perform graph-theoretical measurements on it which do not modify its structure; any other action will lead to undescribed behaviour.

Nodes/neurons are defined by a unique index which can be used to access their properties and those of the connections between them.

In addition to `graph`, the containers can have other attributes, such as:

- `shape` for *`SpatialGraph`*: and *`SpatialNetwork`*:, which describes the spatial delimitations of the neurons' environment (e.g. many *in vitro* culture are contained in circular dishes).
- `population` which contains informations on the various groups of neurons that exist in the network (for instance inhibitory and excitatory neurons can be grouped together)
- `connections` which stores the informations about the synaptic connections between the neurons

1.2.1 Graph-theoretical models

Several classical graphs are efficiently implemented and the generation procedures are detailed in the documentation.

Main module

For more details regarding the main classes, see:

Graph container classes

class `nngt.`**`SpatialGraph`**(*nodes=0*, *name='Graph'*, *weighted=True*, *directed=True*, *from_graph=None*, *shape=None*, *positions=None*, ***kwargs*)

The detailed class that inherits from *`Graph`* and implements additional properties to describe various biological functions and interact with the NEST simulator.

Variables

- **`shape`** – *`Shape`* Shape of the neurons environment.
- **`positions`** – `numpy.array` Positions of the neurons.
- **`graph`** – `GraphObject` Main attribute of the class instance.

classmethod **`make_spatial`**(*graph*, *shape=<nngt.core.graph_datastruct.Shape instance>*, *positions=None*)

`position`

`shape`

class `nngt.`**`Network`**(*name='Graph'*, *weighted=True*, *directed=True*, *from_graph=None*, *population=None*, ***kwargs*)

The detailed class that inherits from *`Graph`* and implements additional properties to describe various biological functions and interact with the NEST simulator.

Variables

- **`population`** – *`NeuralPop`* Object reparting the neurons into groups with specific properties.
- **`graph`** – *`GraphObject`* Main attribute of the class instance
- **`nest_gid`** – `numpy.array` Array containing the NEST gid associated to each neuron; it is `None` until a NEST network has been created.
- **`id_from_nest_gid`** – dict Dictionary mapping each NEST gid to the corresponding neuron index in the `nngt.~Network`

classmethod **`ei_network`**(*size*, *ei_ratio=0.2*, *en_model='aeif_cond_alpha'*, *en_param={}*, *es_model='static_synapse'*, *es_param={}*, *in_model='aeif_cond_alpha'*, *in_param={}*, *is_model='static_synapse'*, *is_param={}*)

Generate a network containing a population of two neural groups: inhibitory and excitatory neurons.

Parameters

- **size** (*int*) – Number of neurons in the network.
- **ei_ratio** (*double, optional (default: 0.2)*) – Ratio of inhibitory neurons: $\frac{N_i}{N_e+N_i}$.
- **en_model** (*string, optional (default: 'aeif_cond_alpha')*) – Nest model for the excitatory neuron.

- **en_param** (*dict, optional (default: {})*) – Dictionary of parameters for the the excitatory neuron.
- **es_model** (*string, optional (default: 'static_synapse')*) – NEST model for the excitatory synapse.
- **es_param** (*dict, optional (default: {})*) – Dictionary containing the excitatory synaptic parameters.
- **in_model** (*string, optional (default: 'aeif_cond_alpha')*) – Nest model for the inhibitory neuron.
- **in_param** (*dict, optional (default: {})*) – Dictionary of parameters for the the inhibitory neuron.
- **is_model** (*string, optional (default: 'static_synapse')*) – NEST model for the inhibitory synapse.
- **is_param** (*dict, optional (default: {})*) – Dictionary containing the inhibitory synaptic parameters.

Returns **net** (`Network` or subclass) – Network of disconnected excitatory and inhibitory neurons.

static **`make_network`**(*graph*, *neural_pop*)
Turn a `Graph` object into a `Network`, or a `SpatialGraph` into a `SpatialNetwork`.

Parameters

- **graph** (`Graph` or `SpatialGraph`) – Graph to convert
- **neural_pop** (`NeuralPop`) – Population to associate to the new `Network`

Notes

In-place operation that directly converts the original graph.

`nest_gid`

`neuron_properties`(*idx_neuron*)
Properties of a neuron in the graph.

Parameters **idx_neuron** (*int*) – Index of a neuron in the graph.

Returns *dict of the neuron properties.*

classmethod **`num_networks`**()
Returns the number of alive instances.

`population`
`NeuralPop` that divides the neurons into groups with specific properties.

classmethod **`uniform_network`**(*size*, *neuron_model='aeif_cond_alpha'*, *neuron_param={}*, *syn_model='static_synapse'*, *syn_param={}*)
Generate a network containing only one type of neurons.

Parameters

- **size** (*int*) – Number of neurons in the network.
- **neuron_model** (*string, optional (default: 'aief_cond_alpha')*) – Name of the NEST neural model to use when simulating the activity.

- **neuron_param** (*dict, optional (default: {})*) – Dictionary containing the neural parameters; the default value will make NEST use the default parameters of the model.
- **syn_model** (*string, optional (default: 'static_synapse')*) – NEST synaptic model to use when simulating the activity.
- **syn_param** (*dict, optional (default: {})*) – Dictionary containing the synaptic parameters; the default value will make NEST use the default parameters of the model.

Returns **net** (`Network` or subclass) – Uniform network of disconnected neurons.

class `nngt.`**`SpatialNetwork`**(*population*, *name='Graph'*, *weighted=True*, *directed=True*, *shape=None*, *from_graph=None*, *positions=None*, ***kwargs*)

Class that inherits from `Network` and `SpatialGraph` to provide a detailed description of a real neural network in space, i.e. with positions and biological properties to interact with NEST.

Variables

- **`shape`** – `nngt.core.Shape` Shape of the neurons environment.
- **`positions`** – `numpy.array` Positions of the neurons.
- **`population`** – `NeuralPop` Object reparting the neurons into groups with specific properties.
- **`graph`** – `GraphObject` Main attribute of the class instance.
- **`nest_gid`** – `numpy.array` Array containing the NEST gid associated to each neuron; it is `None` until a NEST network has been created.
- **`id_from_nest_gid`** – dict Dictionary mapping each NEST gid to the corresponding neuron index in the `nngt.~SpatialNetwork`

Side classes

class `nngt.`**`Shape`**(*parent=None*)

Class containing the shape of the area where neurons will be distributed to form a network.

..warning : so far, only a rectangle can be created.

`area`

double

Area of the shape in mm^2.

`com`

tuple of doubles

Position of the center of mass of the current shape.

`add_subshape: void`

@todo Add a `Shape` to a preexisting one.

`add_subshape`(*subshape*, *position*, *unit='mm'*)

Add a `Shape` to the current one.

Parameters

- **subshape** (`Shape`) – Subshape to add.
- **position** (*tuple of doubles*) – Position of the subshape's center of gravity in space.
- **unit** (*string (default 'mm')*) – Unit in the metric system among 'um', 'mm', 'cm', 'dm', 'm'

Returns *None*

`area`
Area of the shape.

`com`
Center of mass of the shape.

classmethod `rectangle` (*parent*, *height*, *width*, *pos_com=(0.0, 0.0)*)
Generate a rectangle of given height, width and center of mass.

Parameters

- **parent** (`SpatialGraph` or subclass) – The parent container.
- **height** (*float*) – Height of the rectangle.
- **width** (*float*) – Width of the rectangle.
- **pos_com** (*tuple of floats, optional (default: (0., 0.))*) – Position of the rectangle's center of mass

Returns **shape** (`Shape`) – Rectangle shape.

`rnd_distrib` (*nodes=None*)

`set_parent` (*parent*)

`vertices`

class `nngt.`**`NeuralPop`** (*size=None*, *parent=None*, *with_models=True*, ***kwargs*)
The basic class that contains groups of neurons and their properties.

Variables ***`has_models`*** – `bool`, `True` if every group has a `model` attribute.

`add_to_group` (*group_name*, *id_list*)

classmethod `copy` (*pop*)
Copy an existing NeuralPop

classmethod `ei_population` (*size*, *iratio=0.2*, *parent=None*, *en_model='aeif_cond_alpha'*, *en_param={}*, *es_model='static_synapse'*, *es_param={}*, *in_model='aeif_cond_alpha'*, *in_param={}*, *is_model='static_synapse'*, *is_param={}*)
Make a NeuralPop with a given ratio of inhibitory and excitatory neurons.

`has_models`

`is_valid`

`new_group` (*name*, *id_list*, *ntype=1*, *neuron_model=None*, *neuron_param={}*, *syn_model='static_synapse'*, *syn_param={}*)

`next` ()

classmethod `pop_from_network` (*graph*, **args*)
Make a NeuralPop object from a network. The groups of neurons are determined using instructions from an arbitrary number of `GroupProperties`.

`set_model` (*model*, *group=None*)
Set the groups' models.

Parameters

- **model** (*dict*) – Dictionary containing the model type as key ("neuron" or "synapse") and the model name as value (e.g. {"neuron": "iaf_neuron"}).

- **group** (*list of strings, optional (default: None)*) – List of strings containing the names of the groups which models should be updated.
- **.. warning** (*:*) – No check is performed on the validity of the models, which means that errors will only be detected when building the graph in NEST.
- **.. note** (*:*) – By default, synapses are registered as "static_synapse"s in NEST; because of this, only the `neuron_model` attribute is checked by the `has_models` function: it will answer `True` if all groups have a 'non-None' `neuron_model` attribute.

`set_param`(*param*, *group=None*)
Set the groups' parameters.

Parameters

- **param** (*dict*) – Dictionary containing the model type as key ("neuron" or "synapse") and the model parameter as value (e.g. {"neuron": {"C_m": 125.}}).
- **group** (*list of strings, optional (default: None)*) – List of strings containing the names of the groups which models should be updated.
- **.. warning** (*:*) – No check is performed on the validity of the parameters, which means that errors will only be detected when building the graph in NEST.

`size`

classmethod `uniform_population`(*size*, *parent=None*, *neuron_model='aeif_cond_alpha'*, *neuron_param={}*, *syn_model='static_synapse'*, *syn_param={}*)
Make a NeuralPop of identical neurons

NNGT

Neural Networks Growth and Topology analyzing tool.

Provides algorithms for

1. growing networks
2. analyzing their activity
3. studying the graph theoretical properties of those networks

How to use the documentation Documentation is not yet really available. I will try to implement more extensive docstrings within the code. I recommend exploring the docstrings using IPython, an advanced Python shell with TAB-completion and introspection capabilities. See below for further instructions. The docstring examples assume that *numpy* has been imported as *np*:

```
>>> import numpy as np
```

Code snippets are indicated by three greater-than signs:

```
>>> x = 42
>>> x = x + 1
```

Use the built-in `help` function to view a function's docstring:

```
>>> help(nggt.GraphClass)
```

Available subpackages

core Contains the main network classes. These are loaded in nngt at import so specifying `nngt.core` is not necessary

generation Functions to generate specific networks

lib Basic functions used by several sub-packages.

io @todo: Tools for input/output operations

nest NEST integration tools

growth @todo: Growing networks tools

plot plot data or graphs (@todo) using matplotlib and graph_tool

Utilities

show_config @todo: Show build configuration

version NNGT version string

Units Functions related to spatial embedding of networks are using milimeters (mm) as default unit; other units from the metric system can also be provided:

- *um* for micrometers
- *cm* centimeters
- *dm* for decimeters
- *m* for meters

Main graph classes

class `nngt.`**`Graph`**(*nodes=0*, *name='Graph'*, *weighted=True*, *directed=True*, *from_graph=None*, ***kwargs*)

The basic class that contains a `graph_tool.Graph` and some of is properties or methods to easily access them.

Variables

- **`id`** – `int` unique id that identifies the instance.
- **`graph`** – `GraphObject` main attribute of the class instance.

class `nngt.`**`SpatialGraph`**(*nodes=0*, *name='Graph'*, *weighted=True*, *directed=True*, *from_graph=None*, *shape=None*, *positions=None*, ***kwargs*)

The detailed class that inherits from `Graph` and implements additional properties to describe various biological functions and interact with the NEST simulator.

Variables

- **`shape`** – `Shape` Shape of the neurons environment.
- **`positions`** – `numpy.array` Positions of the neurons.
- **`graph`** – `GraphObject` Main attribute of the class instance.

class `nngt.`**`Network`**(*name='Graph'*, *weighted=True*, *directed=True*, *from_graph=None*, *population=None*, ***kwargs*)

The detailed class that inherits from `Graph` and implements additional properties to describe various biological functions and interact with the NEST simulator.

Variables

- **population** – *NeuralPop* Object reparting the neurons into groups with specific properties.
- **graph** – *GraphObject* Main attribute of the class instance
- **nest_gid** – `numpy.array` Array containing the NEST gid associated to each neuron; it is `None` until a NEST network has been created.
- **id_from_nest_gid** – dict Dictionary mapping each NEST gid to the corresponding neuron index in the `nngt.~Network`

class `nngt.`**SpatialNetwork**(*population*, *name='Graph'*, *weighted=True*, *directed=True*, *shape=None*, *from_graph=None*, *positions=None*, ***kwargs*)

Class that inherits from *Network* and *SpatialGraph* to provide a detailed description of a real neural network in space, i.e. with positions and biological properties to interact with NEST.

Variables

- **shape** – `nngt.core.Shape` Shape of the neurons environment.
- **positions** – `numpy.array` Positions of the neurons.
- **population** – *NeuralPop* Object reparting the neurons into groups with specific properties.
- **graph** – *GraphObject* Main attribute of the class instance.
- **nest_gid** – `numpy.array` Array containing the NEST gid associated to each neuron; it is `None` until a NEST network has been created.
- **id_from_nest_gid** – dict Dictionary mapping each NEST gid to the corresponding neuron index in the `nngt.~SpatialNetwork`

Core module

Core classes and functions. Most of them are not visible in the module as they are directly loaded at *nngt* level.

Content

`nngt.core.`**GraphObject**

alias of *GtGraph*

class `nngt.core.`**IGraph**(*nodes=0*, *g=None*, *directed=True*, *parent=None*)

Bases: `abc.Mock`

Subclass of `igraph.Graph`.

__abstractmethods__ = frozenset([])

__init__(*nodes=0*, *g=None*, *directed=True*, *parent=None*)

betweenness_list(*use_weights=True*, *as_prop=False*, *norm=True*)

clear_all_edges()

Remove all connections in the graph.

degree_list(*node_list=None*, *deg_type='total'*, *use_weights=True*)

edge_nb()

new_edge(*source*, *target*, *weight=1.0*)

Adding a connection to the graph, with optional properties.

Parameters

- **source** (`int/node`) – Source node.
- **target** (`int/node`) – Target node.
- **weight** (`double`, optional (default: 1.)) – Weight of the connection (synaptic strength with NEST).

Returns *The new connection.*

`new_edges`(*edge_list*, *eprops=None*)
Adds a list of connections to the graph

`new_node`(*n=1*, *ntype=1*)
Adding a node to the graph, with optional properties.

Parameters

- **n** (*int, optional (default: 1)*) – Number of nodes to add.
- **ntype** (*int, optional (default: 1)*) – Type of neuron (1 for excitatory, -1 for inhibitory)

Returns *The node or an iterator over the nodes created.*

`node_nb`()

`remove_edge`(*edge*)

`remove_vertex`(*node*, *fast=False*)

class `nngt.core.`**`GtGraph`**(*nodes=0*, *g=None*, *directed=True*, *prune=False*, *vorder=None*)
Bases: `abc.Mock`

Subclass of `graph_tool.Graph` that (with `SnapGraph`) unifies the methods to work with either *graph_tool* or *SNAP*.

`__abstractmethods__` = frozenset([])

`__init__`(*nodes=0*, *g=None*, *directed=True*, *prune=False*, *vorder=None*)
@todo: document that see `graph_tool.Graph`'s constructor

`betweenness_list`(*use_weights=True*, *as_prop=False*, *norm=True*)

`clear_all_edges`()

`degree_list`(*node_list=None*, *deg_type='total'*, *use_weights=True*)

`edge_nb`()

`new_edge`(*source*, *target*, *weight=1.0*)
Adding an edge to the graph, with optional properties.

Parameters

- **source** (`int/node`) – Source node.
- **target** (`int/node`) – Target node.
- **weight** (`double`, optional (default: 1.)) – Weight of the connection (synaptic strength with NEST).

Returns *The new edge.*

`new_edges`(*edge_list*, *eprops=None*)
Adds a list of edges to the graph @todo: see how the eprops work

new_node (*n=1*, *ntype=1*)
Adding a node to the graph, with optional properties.

Parameters

- **n** (*int, optional (default: 1)*) – Number of nodes to add.
- **ntype** (*int, optional (default: 1)*) – Type of neuron (1 for excitatory, -1 for inhibitory)

Returns *The node or an iterator over the nodes created.*

node_nb ()

class `nngt.core.`**NxGraph** (*nodes=0*, *g=None*, *directed=True*)
Bases: `abc.Mock`

Subclass of networkx Graph

__abstractmethods__ = frozenset([])

__init__ (*nodes=0*, *g=None*, *directed=True*)

betweenness_list (*use_weights=True*, *as_prop=False*)

clear_all_edges ()
Remove all connections in the graph

degree_list (*node_list=None*, *deg_type='total'*, *use_weights=True*)

di_value = {'int': 0, 'double': 0.0, 'string': ''}

edge_nb ()

new_edge (*source*, *target*, *weight=1.0*)
Adding a connection to the graph, with optional properties.

Parameters

- **source** (`int/node`) – Source node.
- **target** (`int/node`) – Target node.
- **add_missing** (`bool`, optional (default: None)) – Add the nodes if they do not exist.
- **weight** (`double`, optional (default: 1.)) – Weight of the connection (synaptic strength with NEST).

Returns *The new connection.*

new_edge_attribute (*name*, *value_type*, *values=None*, *val=None*)

new_edges (*edge_list*, *eprops=None*)
Adds a list of connections to the graph

new_node (*n=1*, *ntype=1*)
Adding a node to the graph, with optional properties.

Parameters

- **n** (*int, optional (default: 1)*) – Number of nodes to add.
- **ntype** (*int, optional (default: 1)*) – Type of neuron (1 for excitatory, -1 for inhibitory)

Returns *The node or an iterator over the nodes created.*

new_node_attribute (*name*, *value_type*, *values=None*, *val=None*)

node_nb ()

set_node_property()

Generation module

Functions that generates the underlying connectivity of graphs, as well as the synaptic properties (weight/strength and delay).

Content

`nngt.generation.`**connect_neural_groups**(*network*, *source_groups*, *target_groups*, *graph_model*, *model_param*)

Function to connect excitatory and inhibitory population with a given graph model. .. todo

```
    make the modifications for only a set of edges
```

Parameters

- **network** (`Network` or `SpatialNetwork`) – The network to connect.
- **source_groups** (*tuple of strings*) – Names of the source groups (which contain the pre-synaptic neurons)
- **target_groups** (*tuple of strings*) – Names of the target groups (which contain the post-synaptic neurons)
- **graph_model** (*string*) – The name of the connectivity model (among "erdos_renyi", "random_scale_free", "price_scale_free", and "newman_watts").
- **model_param** (*dict*) – Dictionary containing the model parameters (the keys are the keywords of the associated generation function — see above).

`nngt.generation.`**connect_neural_types**(*network*, *source_type*, *target_type*, *graph_model*, *model_param*, *weighted=True*)

Function to connect excitatory and inhibitory population with a given graph model. .. todo

```
    make the modifications for only a set of edges
```

Parameters

- **network** (`Network` or `SpatialNetwork`) – The network to connect.
- **source_type** (*int*) – The type of source neurons (`1` for excitatory, ‘‘-1 for inhibitory neurons).
- **source_type** (*int*) – The type of target neurons.
- **graph_model** (*string*) – The name of the connectivity model (among "erdos_renyi", "random_scale_free", "price_scale_free", and "newman_watts").
- **model_param** (*dict*) – Dictionary containing the model parameters (the keys are the keywords of the associated generation function — see above).
- **weighted** (*bool, optional (default: True)*) – @todo Whether the graph edges have weights.

`nngt.generation.`**distance_rule**(*scale*, *rule='exp'*, *shape=None*, *neuron_density=1000.0*, *nodes=0*, *density=0.1*, *edges=-1*, *avg_deg=-1.0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='DR'*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Create a graph using a 2D distance rule to create the connection between neurons. Available rules are linear and exponential.

Parameters

- **scale** (*float*) – Characteristic scale for the distance rule. E.g for linear distance- rule, $P(i,j) \propto (1{-}d_{ij}/scale))$, whereas for the exponential distance-rule, $P(i,j) \propto e^{-d_{ij}/scale}$.
- **rule** (*string, optional (default: 'exp')*) – Rule that will be apply to draw the connections between neurons. Choose among "exp" (exponential), "lin" (linear, not implemented yet), "power" (power-law, not implemented yet).
- **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment. If not specified, a square will be created with the appropriate dimensions for the number of neurons and the neuron spatial density.
- **neuron_density** (*float, optional (default: 1000.)*) – Density of neurons in space ($neurons \cdot mm^{-2}$).
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **density** (*double, optional (default: 0.1)*) – Structural density given by *edges* / (*nodes* * *nodes*).
- **edges** (*int (optional)*) – The number of edges between the nodes
- **avg_deg** (*double, optional*) – Average degree of the neurons given by *edges* / *nodes*.
- **weighted** (*bool, optional (default: True)*) – @todo Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or not.
- **multigraph** (*bool, optional (default: False)*) – Whether the graph can contain multiple edges between two nodes.
- **name** (*string, optional (default: "DR")*) – Name of the created graph.
- **positions** (`numpy.ndarray`, optional (default: None)) – A 2D or 3D array containing the positions of the neurons in space.
- **population** (`NeuralPop`, optional (default: None)) – Population of neurons defining their biological properties (to create a `Network`).
- **from_graph** (`Graph` or subclass, optional (default: None)) – Initial graph whose nodes are to be connected.

`nngt.generation.`**`erdos_renyi`**(*nodes=0, density=0.1, edges=-1, avg_deg=-1.0, reciprocity=-1.0, weighted=True, directed=True, multigraph=False, name='ER', shape=None, positions=None, population=None, from_graph=None, **kwargs*)

Generate a random graph as defined by Erdos and Renyi but with a reciprocity that can be chosen.

Parameters

- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **density** (*double, optional (default: 0.1)*) – Structural density given by *edges* / *nodes*^{2}.
- **edges** (*int (optional)*) – The number of edges between the nodes
- **avg_deg** (*double, optional*) – Average degree of the neurons given by *edges* / *nodes*.
- **reciprocity** (*double, optional (default: -1 to let it free)*) – Fraction of edges that are bidirectional (only for directed graphs – undirected graphs have a reciprocity of 1 by definition)
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or not.

- **multigraph** (*bool, optional (default: False)*) – Whether the graph can contain multiple edges between two nodes.
- **name** (*string, optional (default: "ER")*) – Name of the created graph.
- **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment.
- **positions** (`numpy.ndarray`, optional (default: None)) – A 2D or 3D array containing the positions of the neurons in space.
- **population** (*`NeuralPop`*, optional (default: None)) – Population of neurons defining their biological properties (to create a *`Network`*).
- **from_graph** (`Graph` or subclass, optional (default: None)) – Initial graph whose nodes are to be connected.

Returns **graph_er** (*`Graph`*, or subclass) – A new generated graph or the modified *from_graph*.

Notes

nodes is required unless *from_graph* or *population* is provided. If an *from_graph* is provided, all preexistant edges in the object will be deleted before the new connectivity is implemented.

`nngt.generation.`**`fixed_degree`**(*degree*, *degree_type='in'*, *nodes=0*, *reciprocity=-1.0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='ER'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a random graph with constant in- or out-degree.

Parameters

- **degree** (*int*) – The value of the constant degree.
- **degree_type** (*str, optional (default: 'in')*) – The type of the fixed degree, among 'in', 'out' or 'total' (@todo: not implemented yet).
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **reciprocity** (*double, optional (default: -1 to let it free)*) – @todo: not implemented yet. Fraction of edges that are bidirectional (only for directed graphs – undirected graphs have a reciprocity of 1 by definition)
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – @todo: only for directed graphs for now. Whether the graph is directed or not.
- **multigraph** (*bool, optional (default: False)*) – Whether the graph can contain multiple edges between two nodes.
- **name** (*string, optional (default: "ER")*) – Name of the created graph.
- **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment.
- **positions** (`numpy.ndarray`, optional (default: None)) – A 2D or 3D array containing the positions of the neurons in space.
- **population** (*`NeuralPop`*, optional (default: None)) – Population of neurons defining their biological properties (to create a *`Network`*).
- **from_graph** (`Graph` or subclass, optional (default: None)) – Initial graph whose nodes are to be connected.

Returns **graph_er** (*`Graph`*, or subclass) – A new generated graph or the modified *from_graph*.

Notes

nodes is required unless *from_graph* or *population* is provided. If an *from_graph* is provided, all preexistant edges in the object will be deleted before the new connectivity is implemented.

`nngt.generation.`**`random_scale_free`**(*in_exp*, *out_exp*, *nodes=0*, *density=0.1*, *edges=-1*, *avg_deg=-1*, *reciprocity=0.0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='RandomSF'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a free-scale graph of given reciprocity and otherwise devoid of correlations.

in_exp [float] Absolute value of the in-degree exponent γ_i, such that :math:'p(k_i) propto k_i^{-gamma_i}

out_exp [float] Absolute value of the out-degree exponent γ_o, such that :math:'p(k_o) propto k_o^{-gamma_o}

nodes [int, optional (default: None)] The number of nodes in the graph.

density: double, optional (default: 0.1) Structural density given by *edges* / (*nodes*'*'*nodes*).

edges [int (optional)] The number of edges between the nodes

avg_deg [double, optional] Average degree of the neurons given by *edges* / *nodes*.

weighted [bool, optional (default: True)] @todo Whether the graph edges have weights.

directed [bool, optional (default: True)] Whether the graph is directed or not.

multigraph [bool, optional (default: False)] Whether the graph can contain multiple edges between two nodes. can contain multiple edges between two

name [string, optional (default: "ER")] Name of the created graph.

shape [`Shape`, optional (default: None)] Shape of the neurons' environment.

positions [`numpy.ndarray`, optional (default: None)] A 2D or 3D array containing the positions of the neurons in space.

population [*`NeuralPop`*, optional (default: None)] Population of neurons defining their biological properties (to create a *`Network`*)

from_graph [`Graph` or subclass, optional (default: None)] Initial graph whose nodes are to be connected.

graph_fs : *`Graph`*

As reciprocity increases, requested values of *in_exp* and *out_exp* will be less and less respected as the distribution will converge to a common exponent :math:'gamma =

rac{gamma_i + gamma_o}{2}'. Parameter *nodes* is required unless *from_graph* or *population* is provided.

`nngt.generation.`**`price_scale_free`**(*m*, *c=None*, *gamma=1*, *nodes=0*, *weighted=True*, *directed=True*, *seed_graph=None*, *multigraph=False*, *name='PriceSF'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a Price graph model (Barabasi-Albert if undirected).

Parameters

- **m** (*int*) – The number of edges each new node will make.

- **c** (*double*) – Constant added to the probability of a vertex receiving an edge.
- **gamma** (*double*) – Preferential attachment power.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **weighted** (*bool, optional (default: True)*) – @todo Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or not.
- **multigraph** (*bool, optional (default: False)*) – Whether the graph can contain multiple edges between two nodes.
- **name** (*string, optional (default: "ER")*) – Name of the created graph.
- **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment
- **positions** (`numpy.ndarray`, optional (default: None)) – A 2D or 3D array containing the positions of the neurons in space.
- **population** (`NeuralPop`, optional (default: None)) – Population of neurons defining their biological properties (to create a `Network`).
- **from_graph** (`Graph` or subclass, optional (default: None)) – Initial graph whose nodes are to be connected.

Returns **graph_price** (`Graph` or subclass.)

Notes

nodes is required unless *from_graph* or *population* is provided.

`nngt.generation.`**`newman_watts`**(*coord_nb*, *proba_shortcut*, *nodes=0*, *directed=True*, *multigraph=False*, *name='NW'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a small-world graph using the Newman-Watts algorithm. .. todo

```
    generate the edges of a circular graph to not replace the graph of the
    `from_graph` and implement chosen reciprocity.
```

Parameters

- **coord_nb** (*int*) – The number of neighbours for each node on the initial topological lattice.
- **proba_shortcut** (*double*) – Probability of adding a new random (shortcut) edge for each existing edge on the initial lattice.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **density** (*double, optional (default: 0.1)*) – Structural density given by *edges* / (*nodes*'*'*nodes*).
- **edges** (*int (optional)*) – The number of edges between the nodes
- **avg_deg** (*double, optional*) – Average degree of the neurons given by *edges* / *nodes*.
- **weighted** (*bool, optional (default: True)*) – @todo Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or not.
- **multigraph** (*bool, optional (default: False)*) – Whether the graph can contain multiple edges between two nodes.
- **name** (*string, optional (default: "ER")*) – Name of the created graph.

- **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment
- **positions** (`numpy.ndarray`, optional (default: None)) – A 2D or 3D array containing the positions of the neurons in space.
- **population** (*`NeuralPop`*, optional (default: None)) – Population of neurons defining their biological properties (to create a *`Network`*).
- **from_graph** (`Graph` or subclass, optional (default: None)) – Initial graph whose nodes are to be connected.

Returns **graph_nw** (*`Graph`* or subclass)

Notes

nodes is required unless *from_graph* or *population* is provided.

Simulation module

Module to interact easily with the NEST simulator. It allows to :

- build a NEST network from *`Network`* or *`SpatialNetwork`* objects,
- monitor the activity of the network (taking neural groups into account)
- plot the activity while separating the behaviours of predefined neural groups

Content

`nngt.simulation.`**`make_nest_network`**(*network*, *use_weights=True*)

Create a new subnetwork which will be filled with neurons and connector objects to reproduce the topology from the initial network.

Parameters

- **network** (*`nngt.Network`* or *`nngt.SpatialNetwork`*) – the network we want to reproduce in NEST.
- **use_weights** (*bool, optional (default: True)*) – Whether to use the network weights or default ones (value: 10.).

Returns

- **subnet** (*tuple (node in NEST)*) – GID of the new NEST subnetwork
- **gids** (*tuple (nodes in NEST)*) – GIDs of the neurons in *subnet*

`nngt.simulation.`**`get_nest_network`**(*nest_subnet*, *id_converter=None*)

Get the adjacency matrix describing a NEST subnetwork.

Parameters

- **nest_subnet** (*tuple*) – Subnetwork node in NEST.
- **id_converter** (*dict, optional (default: None)*) – A dictionary which maps NEST gids to the desired neurons ids.

Returns **mat_adj** (`lil_matrix`) – Adjacency matrix of the network.

`nngt.simulation.`**`set_noise`**(*gids*, *mean*, *std*)

Submit neurons to a current white noise. @todo: check how NEST handles the $\sqrt{t}$ in the standard dev.

Parameters

- **gids** (*tuple*) – NEST gids of the target neurons.
- **mean** (*float*) – Mean current value.
- **std** (*float*) – Standard deviation of the current

Returns **noise** (*tuple*) – The NEST gid of the noise_generator.

`nngt.simulation.`**`set_poisson_input`**(*gids*, *rate*)

Submit neurons to a Poissonian rate of spikes.

Parameters

- **gids** (*tuple*) – NEST gids of the target neurons.
- **rate** (*float*) – Rate of the spike train.

Returns **poisson_input** (*tuple*) – The NEST gid of the poisson_generator.

`nngt.simulation.`**`monitor_nodes`**(*gids, nest_recorder=['spike_detector'], params=[{}], network=None*)

Monitoring the activity of nodes in the network.

Parameters

- **gids** (*tuple of ints or list of tuples*) – GIDs of the neurons in the NEST subnetwork; either one list per recorder if they should monitor different neurons or a unique list which will be monitored by all devices.
- **nest_recorder** (*list of strings, optional (default: ["spike_detector"])*) – List of devices to monitor the network.
- **params** (*list of dict, optional (default: [{}])*) – List of dictionaries containing the parameters for each recorder (see NEST documentation for details).
- **network** (`Network` or subclass, optional (default: “”)) – Network which population will be used to differentiate inhibitory and excitatory spikes.

Returns **recorders** (*tuple*) – Tuple of the recorders’ gids

`nngt.simulation.`**`plot_activity`**(*gid_recorder*, *record*, *network=None*, *gids=None*, *show=True*, *limits=None*, *hist=True*)

Plot the monitored activity.

Parameters

- **gid_recorder** (*tuple or list*) – The gids of the recording devices.
- **record** (*tuple or list*) – List of the monitored variables for each device.
- **network** (`Network` or subclass, optional (default: None)) – Network which activity will be monitored.
- **gids** (*tuple, optional (default: None)*) – NEST gids of the neurons which should be monitored.
- **show** (*bool, optional (default: True)*) – Whether to show the plot right away or to wait for the next plt.show().
- **hist** (*bool, optional (default: True)*) – Whether to display the histogram when plotting spikes rasters.
- **limits** (*tuple, optional (default: None)*) – Time limits of the plot (if not specified, times of first and last spike for raster plots).

Returns **fignums** (*list*) – List of the figure numbers.

`nngt.simulation.`**`activity_types`**(*network*, *spike_detector*, *limits*, *phase_coeff=(0.5, 10.0)*, *mbis=0.5*, *mfb=0.2*, *mflb=0.05*, *simplify=False*, *fignums=[]*, *show=False*)

Analyze the spiking pattern of a neural network. @todo: think about inserting t= and t=simtime at the beginning and at the end of ``times``.

Parameters

- **network** (`Network`) – Neural network that was analyzed
- **spike_detector** (*NEST node(s), (tuple or list of tuples)*) – The recording device that monitored the network's spikes
- **limits** (*tuple of floats*) – Time limits of the simulation regrion which should be studied (in ms).
- **phase_coeff** (*tuple of floats, optional (default: (0.2, 5.))*) – A phase is considered *bursting' when the interspike between all spikes that compose it is smaller than ``phase_coeff[0]*avg_rate`* (where `avg_rate` is the average firing rate), *quiescent' when it is greater that ``phase_coeff[1]*avg_rate`*, 'mixed' otherwise.
- **mbis** (*float, optional (default: 0.5)*) – Maximum interspike interval allowed for two spikes to be considered in the same burst (in ms).
- **mfb** (*float, optional (default: 0.2)*) – Minimal fraction of the neurons that should participate for a burst to be validated (i.e. if the interspike is smaller that the limit BUT the number of participating neurons is too small, the phase will be considered as *localized*).
- **mflb** (*float, optional (default: 0.05)*) – Minimal fraction of the neurons that should participate for a local burst to be validated (i.e. if the interspike is smaller that the limit BUT the number of participating neurons is too small, the phase will be considered as *mixed*).
- **simplify** (*bool, optional (default: False)*) – If `True`, *mixed* phases that are contiguous to a burst are incorporated to it.
- **return_steps** (*bool, optional (default: False)*) – If `True`, a second dictionary, *phases_steps* will also be returned. @todo: not implemented yet
- **fignums** (*list, optional (default: [])*) – Indices of figures on which the periods can be drawn.
- **show** (*bool, optional (default: False)*) – Whether the figures should be displayed.

Returns

- **phases** (*dict*) – Dictionary containing the time intervals (in ms) for all four phases (*bursting', 'quiescent', 'mixed', and 'localized*) as lists. E.g: `phases["bursting"]` could give `[[123.5,334.2],[857.1,1000.6]]`.
- **phases_steps** (*dict, optional (not implemented yet)*) – Dictionary containing the timesteps in NEST.

Plot module

Functions for plotting graphs and graph properties.

note :: For now, graph plotting is only supported when using the graph_tool library.

Content

`nngt.plot.`**`degree_distribution`**(*network*, *deg_type='total'*, *node_list=None*, *num_bins=50*, *use_weights=True*, *logx=False*, *logy=False*, *fignum=None*, *show=True*)

Plotting the degree distribution of a graph.

Parameters

- **graph** (`Graph` or subclass) – the graph to analyze.
- **deg_type** (*string or tuple, optional (default: "total")*) – type of degree to consider ("in", "out", or "total")
- **node_list** (*list or numpy.array of ints, optional (default: None)*) – Restrict the distribution to a set of nodes (default: all nodes).
- **use_weights** (*bool, optional (default: True)*) – use weighted degrees (do not take the sign into account : all weights are positive).
- **logx** (*bool, optional (default: False)*) – use log-spaced bins.
- **logy** (*bool, optional (default: False)*) – use logscale for the degree count.
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.

`nngt.plot.`**`betweenness_distribution`**(*network*, *btype='both'*, *use_weights=True*, *logx=False*, *logy=False*, *fignum=None*, *show=True*)

Plotting the betweenness distribution of a graph.

Parameters

- **graph** (`Graph` or subclass) – the graph to analyze.
- **btype** (*string, optional (default: "both")*) – type of betweenness to display ("node", "edge" or "both")
- **use_weights** (*bool, optional (default: True)*) – use weighted degrees (do not take the sign into account : all weights are positive).
- **logx** (*bool, optional (default: False)*) – use log-spaced bins.
- **logy** (*bool, optional (default: False)*) – use logscale for the degree count.
- **fignum** (*int, optional (default: None)*) – Number of the Figure on which the plot should appear
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.

`nngt.plot.`**`spike_raster`**(*spike_data*, *limits=None*, *title='Spike raster'*, *hist=True*, *num_bins=1000*, *neural_groups=None*, *fignum=None*, *show=True*)

Plotting routine that constructs a raster plot along with an optional histogram.

Parameters

- **spike_data** (2D-array (`numpy.array` or list)) – An 2-column array containing the neuron ids in the first row and the spike times in the second.
- **limits** (*tuple, optional (default: None)*) – Time limits of the plot (if not specified, times of first and last spike).
- **title** (*string, optional (default: 'Spike raster')*) – Title of the raster plot.
- **hist** (*bool, optional (default: True)*) – Whether to plot the raster's histogram.
- **num_bins** (*int, optional (default: 1000)*) – Number of bins for the histogram.

- **neural_groups** (`NeuralPop` or list of neuron ids) – An object that defines the different neural groups to plot their spikes in different colors.
- **fignum** (*int, optional (default: None)*) – Id of another raster plot to which the new data should be added.
- **show** (*bool, optional (default: True)*) – Whether to show the plot right away or to wait for the next plt.show().

Returns **fig.number** (*int*) – Id of the `matplotlib.Figure` on which the raster is plotted.

`nngt.plot.`**`draw_network`**(*network*, *nsize='total-degree'*, *ncolor='group'*, *nshape='o'*, *nborder_color='k'*, *nborder_width=0.5*, *esize=1.0*, *ecolor='k'*, *spatial=True*, *size=(600, 600)*, *dpi=75*)

Draw a given graph/network.

Parameters

- **network** (`Graph` or subclass) – The graph/network to plot.
- **nsize** (*float, array of floats or string, optional (default: "total-degree")*) – Size of the nodes; if a number, percentage of the canvas length, otherwize a string that correlates the size to a node attribute among "in/out/total-degree", "betweenness".
- **ncolor** (*float, array of floats or string, optional (default: 0.5)*) – Color of the nodes; if a float in [0, 1], position of the color in the current palette, otherwise a string that correlates the color to a node attribute among "in/out/total-degree", "betweenness" or "group".
- **nshape** (*char or array of chars, optional (default: "o")*) – Shape of the nodes (see Matplotlib markers).
- **nborder_color** (*char, array of char, float or array of float, optional (default: "k")*) – Color of the node's border using predefined Matplotlib colors). or floats in [0, 1] defining the position in the palette.
- **nborder_width** (*float or array of floats, optional (default: 0.5)*) – Width of the border in percent of canvas size.
- **esize** (*float or array of floats, optional (default: 0.5)*) – Width of the edges in percent of canvas size.
- **ecolor** (*char, array of char, float or array of float, optional (default: "k")*) – Edge color.
- **spatial** (*bool, optional (default: True)*) – If True, use the neurons' positions to draw them.
- **size** (*tuple of ints, optional (default: (600,600))*) – (width, height) tuple for the canvas size (in px).
- **dpi** (*int, optional (default: 75)*) – Resolution (dot per inch).

Analysis module

Content

`nngt.analysis.`**`degree_distrib`**(*graph*, *deg_type='total'*, *node_list=None*, *use_weights=True*, *log=False*, *num_bins=30*)

Computing the degree distribution of a graphwork.

Parameters

- **graph** (`Graph` or subclass) – the graphwork to analyze.
- **deg_type** (*string, optional (default: "total")*) – type of degree to consider ("in", "out", or "total").

- **node_list** (*list or numpy.array of ints, optional (default: None)*) – Restrict the distribution to a set of nodes (default: all nodes).
- **use_weights** (*bool, optional (default: True)*) – use weighted degrees (do not take the sign into account: all weights are positive).
- **log** (*bool, optional (default: False)*) – use log-spaced bins.

Returns

- **counts** (`numpy.array`) – number of nodes in each bin
- **deg** (`numpy.array`) – bins

`nngt.analysis.`**`betweenness_distrib`**(*graph*, *use_weights=True*, *log=False*)

Computing the betweenness distribution of a graphwork

Parameters

- **graph** (`Graph` or subclass) – the graphwork to analyze.
- **use_weights** (*bool, optional (default: True)*) – use weighted degrees (do not take the sign into account : all weights are positive).
- **log** (*bool, optional (default: False)*) – use log-spaced bins.

Returns

- **ncounts** (`numpy.array`) – number of nodes in each bin
- **nbetw** (`numpy.array`) – bins for node betweenness
- **ecounts** (`numpy.array`) – number of edges in each bin
- **ebetw** (`numpy.array`) – bins for edge betweenness

`nngt.analysis.`**`assortativity`**(*graph*, *deg_type='total'*)

Assortativity of the graph. àtodo: check how the various libraries functions work.

Parameters

- **graph** (`Graph` or subclass) – Network to analyze.
- **deg_type** (*string, optional (default: 'total')*) – Type of degree to take into account (among 'in', 'out' or 'total').

Returns *a float describing the graphwork assortativity.*

`nngt.analysis.`**`reciprocity`**(*graph*)

Returns the graphwork reciprocity, defined as $E^{\leftrightarrow}/E$, where $E^{\leftrightarrow}$ and E are, respectively, the number of bidirectional edges and the total number of edges in the graphwork.

`nngt.analysis.`**`clustering`**(*graph*)

Returns the global clustering coefficient of the graph, defined as

$$c = 3imes$$

rac{ ext{number of triangles}} { ext{number of connected triples}}

`nngt.analysis.`**`num_iedges`**(*graph*)

Returns the number of inhibitory connections.

`nngt.analysis.`**`num_scc`**(*graph*, *listing=False*)

Returns the number of strongly connected components, i.e. ensembles where all nodes inside the ensemble can reach any other node in the ensemble using the directed edges.

See also:

`num_wcc()`

`nngt.analysis.`**`num_wcc`**`(`*graph*, *listing=False*`)`

Connected components if the directivity of the edges is ignored (i.e. all edges are considered as bidirectional).

See also:

`num_scc()`

`nngt.analysis.`**`diameter`**`(`*graph*`)`

Pseudo-diameter of the graph @todo: weighted diameter

`nngt.analysis.`**`spectral_radius`**`(`*graph*, *typed=True*, *weighted=True*`)`

Spectral radius of the graph, defined as the eigenvalue of greatest module.

Parameters

- **graph** (`Graph` or subclass) – Network to analyze.
- **typed** (*bool, optional (default: True)*) – Whether the excitatory/inhibitory type of the connnections should be considered.
- **weighted** (*bool, optional (default: True)*) – Whether the weights should be taken into account.

Returns *the spectral radius as a float.*

`nngt.analysis.`**`adjacency_matrix`**`(`*graph*, *types=True*, *weights=True*`)`

Adjacency matrix of the graph.

Parameters

- **graph** (`Graph` or subclass) – Network to analyze.
- **types** (*bool, optional (default: True)*) – Whether the excitatory/inhibitory type of the connnections should be considered (only if the weighing factor is the synaptic strength).
- **weights** (*bool or string, optional (default: True)*) – Whether weights should be taken into account; if True, then connections are weighed by their synaptic strength, if False, then a binary matrix is returned, if *weights* is a string, then the ponderation is the correponding value of the edge attribute (e.g. “distance” will return an adjacency matrix where each connection is multiplied by its length).

Returns a `csr_matrix`.

Lib module

Content

exception `nngt.lib.`**`InvalidArgument`**

Error raise when an argument is invalid.

`nngt.lib.`**`load_from_file`**`(`*filename*, *format='neighbour'*, *delimiter=' '*, *secondary=';'*, *attributes=[]*, *notifier='@'*, *ignore='#'*`)`

Import a saved graph as a (set of) `csr_matrix` from a file. @todo: implement population and shape loading, implement gml, dot, xml, gt

Parameters

- **filename** (*str*) – The path to the file.

- **format** (*str, optional (default: "neighbour")*) – The format used to save the graph. Supported formats are: "neighbour" (neighbour list, default if format cannot be deduced automatically), "ssp" (scipy.sparse), "edge_list" (list of all the edges in the graph, one edge per line, represented by a `source target`-pair), "gml" (gml format, default if *filename* ends with '.gml'), "graphml" (graphml format, default if *filename* ends with '.graphml' or '.xml'), "dot" (dot format, default if *filename* ends with '.dot'), "gt" (only when using *graph_tool'<http://graph-tool.skewed.de/>_ as library, detected if 'filename* ends with '.gt').
- **delimiter** (*str, optional (default " ")*) – Delimiter used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")
- **secondary** (*str, optional (default: ";")*) – Secondary delimiter used to separate attributes in the case of custom formats.
- **attributes** (*list, optional (default: [])*) – List of names for the attributes present in the file. If a *notifier* is present in the file, names will be deduced from it; otherwise the attributes will be numbered.
- **notifier** (*str, optional (default: "@")*) – Symbol specifying the following as meaningfull information. Relevant information are formatted `@info_name=info_value`, where `info_name` is in ("attributes", "directed", "name", "size") and associated `info_value``s are of type (``list`, `bool`, `str`, `int`). Additional notifiers are `@type=SpatialGraph/Network/SpatialNetwork`, which must be followed by the relevant notifiers among `@shape`, `@population`, and `@graph`.
- **ignore** (*str, optional (default: "#")*) – Ignore lines starting with the *ignore* string.

Returns

- **edges** (*list of 2-tuple*) – Edges of the graph.
- **di_attributes** (*dict*) – Dictionary containing the attribute name as key and its value as a list sorted in the same order as *edges*.
- **pop** (`NeuralPop`) – Population (`None` if not present in the file).
- **shape** (`Shape`) – Shape of the graph (`None` if not present in the file).

`nngt.lib.`**`save_to_file`**(*graph*, *filename*, *format='auto'*, *delimiter=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*)

Save a graph to file. @todo: implement population and shape saving, implement gml, dot, xml, gt

Parameters

- **graph** (`Graph` or subclass) – Graph to save.
- **filename** (*str*) – The path to the file.
- **format** (*str, optional (default: "auto")*) – The format used to save the graph. Supported formats are: "neighbour" (neighbour list, default if format cannot be deduced automatically), "ssp" (scipy.sparse), "edge_list" (list of all the edges in the graph, one edge per line, represented by a `source target`-pair), "gml" (gml format, default if *filename* ends with '.gml'), "graphml" (graphml format, default if *filename* ends with '.graphml' or '.xml'), "dot" (dot format, default if *filename* ends with '.dot'), "gt" (only when using *graph_tool'<http://graph-tool.skewed.de/>_ as library, detected if 'filename* ends with '.gt').
- **delimiter** (*str, optional (default " ")*) – Delimiter used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")
- **secondary** (*str, optional (default: ";")*) – Secondary delimiter used to separate attributes in the case of custom formats.

- **attributes** (list, optional (default: `None`)) – List of names for the edge attributes present in the graph that will be saved to disk; by default (`None`), all attributes will be saved.
- **notifier** (*str, optional (default: "@")*) – Symbol specifying the following as meaningfull information. Relevant information are formatted `@info_name=info_value`, with `info_name` in ("attributes", "attr_types", "directed", "name", "size"). Additional notifiers are `@type=SpatialGraph/Network/SpatialNetwork`, which are followed by the relevant notifiers among `@shape`, `@population`, and `@graph` to separate the sections.
- **warning** (*:*) – For now, all formats lead to dataloss if your graph is a subclass of *`SpatialGraph`* or *`Network`* (the *`Shape`* and *`NeuralPop`* attributes will not be saved).

Properties module

So far it contains only NeuralGroup... I'm thinking about moving it to core.

Content

CHAPTER 2

Installation

2.1 Simple install

Under most linux distributions, the simplest way is to install *pip <https://pip.pypa.io/en/stable/installing/>* and git, then simply type into a terminal: >>> sudo pip install git+https://github.com/Silmathoron/NNGT.git

There are ways for windows users, but NEST won't work anyway...

2.2 Dependencies

This package depends on several libraries (the number varies according to which modules you want to use).

2.2.1 Basic dependencies

Regardless of your needs, the following libraries are required:

- numpy
- scipy
- matplotlib
- graph_tool
- or igraph
- or networkx

Note: If they are not present on your computer, `pip` will directly try to install the three first libraries, however:

- lapack is necessary for scipy and pip cannot install it on its own
- you will have to install the graph library yourself (only networkx can be installed directly using `pip`)

2.2.2 Using NEST

If you want to simulate activities on your complex networks, NNGT can directly interact with the NEST simulator to implement

- Python headers (*python-dev* package on many linux distributions)
- autoconf
- automake
- libtool
- libltdl

CHAPTER 3

Graph generation

3.1 Principle

In order to keep the code as generic and easy to maintain as possible, the generation of graphs or networks is divided in several steps:

- **Structured connectivity:** a simple graph is generated as an assembly of nodes and edges, without any biological properties. This allows us to implement known graph-theoretical algorithms in a straightforward fashion.
- **Populations:** detailed properties can be implemented, such as inhibitory synapses and separation of the neurons into inhibitory and excitatory populations – these can be done while respecting user-defined constraints.
- **Synaptic properties:** eventually, synaptic properties such as weight/strength and delays can be added to the network.

3.2 Modularity

The library as been designed so that these various operations can be realized in any order!

Juste to get work on a topological graph/network:

1. Create graph class
2. Connect
3. Set connection weights (optional)
4. Spatialize (optional)
5. Set types (optional: to use with NEST)

To work on a really spatially embedded graph/network:

1. Create spatial graph/network
2. Connect (can depend on positions)
3. Set connection weights (optional, can depend on positions)
4. Set types (optional)

Or to model a complex neural network in NEST:

1. Create spatial network (with space and neuron types)
2. Connect (can depend on types and positions)

3. Set connection weights and types (optional, can depend on types and positions)

3.3 Setting weights

The weights can be either user-defined or generated by one of the available distributions (MAKE A REF). User-defined weights are generated via:

- a list of edges
- a list of weights

Pre-defined distributions require the following variables:

- a distribution name ("constant", "gaussian"...)
- a dictionary containing the distribution properties
- an optional attribute for distributions that are correlated to another (e.g. the distances between neurons)
- a optional value defining the variance of the Gaussian noise that should be applied on the weights

There are several ways of settings the weights of a graph which depend on the time at which you assign them.

At graph creation You can define the weights by entering a `weight_prop` argument to the constructor; this should be a dictionary containing at least the name of the weight distribution: `{"distrib": "distribution_name"}`. If entered, this will be stored as a graph property and used to assign the weights whenever new edges are created unless you specifically assign rules for those new edges' weights.

At any given time You can use the `Connections` class to set the weights of a `graph` explicitely by using:

```
>>> nngt.Connections.weights(graph, elist=edges_to_weigh, distrib="distrib_of_choice", ...)
```

3.4 Examples

```
import nngt
import nngt.generation as ng
```

Generating simple graphs:

```
# random graphs
g1 = ng.erdos_renyi(1000, avg_deg=25)
g2 = ng.erdos_renyi(1000, avg_deg=25, directed=False) # the same graph but undirected
# scale-free with Gaussian weight distribution
g3 = nngt.Graph(1000, weight_prop={"distrib":"gaussian", "distrib_prop":{"avg": 60., "std":5.}})
ng.random_scale_free(2.2, 2.9, from_graph=g3)
```

Generating a network with excitatory and inhibitory neurons:

```
# 800 excitatory neurons, 200 inhibitory
net = nngt.Network.ei_network(1000, ei_ratio=0.2)
# connect the populations
ng.connect_neural_types(net, 1, -1, "erdos_renyi", {"density": 0.035}) # exc -> inhib
ng.connect_neural_types(net, 1, 1, "newman_watts", {"coord_nb":10, "proba_shortcut": 0.1}) # exc -> e
ng.connect_neural_types(net, -1, 1, "random_scale_free", {"in_exp": 2.1, "out_exp": 2.6, "density": 0
ng.connect_neural_types(net, -1, -1, "erdos_renyi", {"density": 0.04}) # inhib -> inhib
```

CHAPTER 4

Properties of graph components

Warning: This section is not up to date anymore!

4.1 Components

In the graph libraries used by NNGT, the main components of a graph are *nodes* (also called *vertices* in graph theory), which correspond to *neurons* in neural networks, and *edges*, which link *nodes* and correspond to synaptic connections between neurons in biology.

The library supposes for now that nodes/neurons and edges/synapses are always added and never removed. Because of this, we can attribute indices to the nodes and the edges which will be directly related to the order in which they have been created (the first node will have index 0, .

4.2 Node properties

If you are just working with basic graphs (for instance looking at the influence of topology with purely excitatory networks), then your nodes do not need to have properties. This is the same if you consider only the average effect of inhibitory neurons by including inhibitory connections between the neurons but not a clear distinction between populations of purely excitatory and purely inhibitory neurons. To model more realistic networks, however, you might want to define these two types of populations and connect them in specific ways.

4.2.1 Two types of node properties

In the library, there is a difference between:

- spatial properties (the positions of the neurons), which are stored in a specific `numpy.array`,
- biological/group properties, which define assemblies of nodes sharing common properties, and are stored inside a `NeuralPop` object.

4.2.2 Biological/group properties

Note: All biological/group properties are stored in a `NeuralPop` object inside a *`Network`* instance (let us call it `graph` in this example); this attribute can be accessed using `graph.population`. `NeuralPop` objects can also be created from a *`Graph`* or *`SpatialGraph`* but they will not be stored inside the object.

The `NeuralPop` class allows you to define specific groups of neurons (described by a `NeuralGroup`). Once these populations are defined, you can constrain the connections between those populations. If the connectivity already exists, you can use the `GroupProperties` class to create a population with groups that respect specific constraints.

Warning: The implementation of this library has been optimized for generating an arbitrary number of neural populations where neurons share common properties; this implies that accessing the properties of one specific neuron will take O(N) operations, where N is the number of neurons. This might change in the future if such operations are judged useful enough.

4.3 Edge properties

In the library, there is a difference between the (synaptic) weights and types (excitatory or inhibitory) and the other biological properties (delays, synaptic models and synaptic parameters). This is because the weights and types are directly involved in many measurements in graph theory and are therefore directly stored inside the *`GraphObject`*.

CHAPTER 5

Detailed structure

Warning: This section is not up to date anymore!

Here is a small bottom-up approach of the library to justify its structure.

5.1 Rationale for the structure

5.1.1 The basis: a graph

The core object is *`nngt.core.GraphObject`* that inherits from either `graph_tool.Graph` or `snap.TNEANet` and `Shape` that encodes the spatial structure of the neurons' environment. The purpose of `GraphObject` is simple: implementing a library independant object with a unique set of functions to interact with graphs.

Warning: This object should never be directly modified through its methods but rather using those of the four containing classes. The only reason to access this object should be to perform graph-theoretical measurements on it which do not modify its structure; any other action will lead to undescribed behaviour.

5.1.2 Frontend

Detailed neural networks contain properties that the *`GraphObject`* does not know about; because of this, direct modification of the structure can lead to nodes or edges missing properties or to properties assigned to nonexistent nodes or edges.

The user can safely interact with the graph using one of the following classes:

- *`Graph`*: container for simple topological graphs with no spatial embedding, nor biological properties
- *`SpatialGraph`*: container for spatial graphs without biological properties
- *`Network`*: container for topological graphs with biological properties (to interact with NEST)
- *`SpatialNetwork`*: container with spatial and biological properties (to interact with NEST)

The reason behind those four objects is to ensure coherence in the properties: either nodes/edges all have a given property or they all don't. Namely:

- adding a node will always require a position parameter when working with a spatial graph,
- adding a node or a connection will always require biological parameters when working with a network.

Moreover, these classes contain the `GraphObject` in their `graph` attribute and do not inherit from it. The reason for this is to make it easy to maintain different addition/deletion functions for the topological and spatial container by keeping independant of the graph library. (otherwise overwriting one of these function would require the use of library-dependant features).

CHAPTER 6

Graph attributes

Warning: This section is not up to date anymore!

The *Graph* class and its subclasses contain several attributes regarding the properties of the edges and nodes. Edges attributes are contained in the graph dictionary; more complex properties about the biological details of the nodes/neurons are contained in the NeuralPop member of the *Graph*. These are briefly described in Properties of graph components; a more detailed description is provided here.

6.1 Attributes and graph libraries

Usual graph libraries can store node and edge properties; as an example, many graphs are weighted and these weights can then be used to compute other properties such as weighted centralities, which is why it is interesting to have those properties stored in the basic graph library class.

The *graph_object.py* file contains the `_GtEProperty` and `_GtNProperty` classes which allow a generic interactions with the various libraries ways of storing properties.

However, several problems occurs:

- for graph_tool, the edge properties are stored in a linear array that is not directly related to the adjacency matrix, thus difficult to handle; this could however be avoided by multiplying the adjacency matrix by the property of interest...
- but for igraph, there is not straightforward way to obtain a scipy adjacency matrix multiplied by an edge property...

To get rid of those problems, the (possibly temporary) solution adopted is to have the weights (synaptic strength) and types (inhibitory or excitatory) attributes stored both in the graph library object and in the *Graph* container.

The libraries indices the edges in the order they are created; because of this, weights must be added to the library using the edge list, which is stored inside the *Graph* container (access it through the `''edges''` key). The addition is performed in the following way: let `lil_matrix_attribute` contain the attribute of interest and `network` be the graph container to which we want to add the property, then the following code is used,

```
>>> sources, targets = network["edges"][:,0], network["edges"][:,1]
>>> list_ordered_weights = lil_matrix_attribute[sources,targets].data[0]
>>> network.graph.new_edge_attribute("weight", "double", values=list_ordered_weights)
```

6.2 Use of attributes in a graph object

This allows for fast graph filtering: we can keep only the edges or nodes we are interested in.

This property is invaluable if you want to study the graph properties of only the inhibitory network or look a the squeleton of the strongest synapses in the graph...

Note: This mixed format is not too good... I should either store everything in the container or in the library graph.

Library graph:

- difficult to manage
- but users can use the library on the graph
- if I cannot provide a fast conversion, it will be bad to interact with NEST

Container:

- easier to manage
- but need to convert for the analysis functions
- users cannot use the library as the graph misses its attributes
- optimized for NEST interactions

CHAPTER 7

Overview

The Neural Network Growth and Topology (NNGT) module provides tools to grow and study detailed biological networks by interfacing efficient graph libraries with highly distributed activity simulators.

7.1 Main classes

NNGT uses four main classes:

`Graph` provides a simple implementation over graphs objects from graph libraries (namely the addition of a name, management of detailed nodes and connection properties, and simple access to basic graph measurements).

`SpatialGraph` a Graph embedded in space (neurons have positions and connections are associated to a distance)

`Network` provides more detailed characteristics to emulate biological neural networks, such as classes of inhibitory and excitatory neurons, synaptic properties...

`SpatialNetwork` combines spatial embedding and biological properties

7.2 Generation of graphs

Structured connectivity: connectivity between the nodes can be chosen from various well-known graph models

Populations: populations of neurons are distributed afterwards on the structured connectivity, and can be set to respect various constraints (for instance a given fraction of inhibitory neurons and synapses)

Synaptic properties: synaptic weights and delays can be set from various distributions or correlated to edge properties

7.3 Interacting with NEST

The generated graphs can be used to easily create complex networks using the NEST simulator, on which you can then simulate their activity.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search