NNGT Documentation

Release 0.6

Tanguy Fardet

May 21, 2017

User Documentation

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.

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

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

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.

The docs

Installation

Dependencies

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

Basic dependencies

Regardless of your needs, the following libraries are required:

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

Additionnal dependencies

- matplotlib (optional but will limit the functionalities if not present)
- *peewee* for database features

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*)

Simple install

Linux

Install the requirements (through `aptitude` or `apt-get` on debian/ubuntu/mint, `pacman` and `yaourt` on arch-based distributions, or your *.rpm* manager on fedora. Otherwise you can also install the latest versions via *pip*:

```
sudo pip install numpy scipy matplotlib networkx
```

Under most linux distributions, the simplest way is to install pip and git, then simply type into a terminal:

```
sudo pip install git+https://github.com/Silmathoron/NNGT.git
```

Mac

I recommend using Macports with which you can install all required features to use *NEST* and *NNGT* with *graph-tool*. The following command lines are used with *python 2.7* since it is what people are used to but I recommend using version *3.5* or higher (replace all 27/2.7 by 35/3.5).

```
sudo port select gcc mp-gcc5 && sudo port install gsl +gcc5 && sudo port install␣
↪autoconf automake libtool && sudo port install python27 pip && sudo port select␣
↪python python27 && sudo port install py27-cython && sudo port select cython␣
↪cython27 && sudo port install py27-numpy py27-scipy py27-matplotlib py27-ipython &&␣
↪sudo port select ipython ipython-2.7 && sudo port install py-graph-tool gtk3
```

Windows

It's the same as linux for windows users once you've installed Python and *pip*, but NEST won't work anyway...

Note: *igraph* can be installed on windows if you need something faster than *networkx*.

Local install

If you want to modify the library more easily, you can also install it locally, then simply add it to your `PYTHONPATH` environment variable:

```
cd && mkdir .nngt-install
cd .nngt-install
git clone https://github.com/Silmathoron/NNGT.git .
git submodule init
git submodule update
nano .bash_profile
```

Then add:

```
export PYTHONPATH="/path/to/your/home/.nngt-install/src/:PYTHONPATH"
```

In order to update your local repository to keep it up to date, you will need to run the two following commands:

```
git pull origin master
git submodule update --remote --merge
```

Configuration

The configuration file is created in `~/.nngt/nngt.conf` after you first run `import nngt` in *python*. Here is the default file:

```
###########################
# NNGT configuration file #
###########################


## default graph library
# (choose among "graph-tool", "igraph", "networkx")
graph_library = graph-tool

## Matplotlib backend
# Uncomment and choose among your available backends (http://matplotlib.org/faq/usage_
↪faq.html#what-is-a-backend)
#backend = Qt5Agg

## settings for data logging
set_logging = False

# use a database (if False, results will be stored in CSV files)
to_file = False
#log_folder = ~/.nngt/database

# database url or temporary database used if use_database = False
# example of real database url: db_url = mysql://user:password@host:port/my_db
db_url = mysql:///nngt_db
```

It can be necessary to modify this file to use the desired graph library, but mostly to correct problems with GTK and matplotlib (if the *plot* module complains, try `Gtk3Agg` and `Qt4Agg`).

Using NEST

If you want to simulate activities on your complex networks, NNGT can directly interact with the NEST simulator to implement the network inside *PyNEST*. For this, you will need to install NEST with Python bindings, which requires:

- the python headers (*python-dev* package on debian-based distribs)
- *autoconf*
- *automake*
- *libtool*
- *libltdl*
- *libncurses*
- *readlines*
- *gsl* (the GNU Scientific Library) for many neuronal models

Introduction

Yet another graph library?

It is not ;)

This library is based on existing graph libraries (such as graph_tool, igraph, networkx, 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.

Examples are given in the following sections:

Graph generation

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.

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)

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` explicitly by using:

```
>>> nngt.Connections.weights(graph, elist=edges_to_weigh, distrib="distrib_of_
↪choice", ...)
```

Examples

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

Simple generation

```python
num_nodes = 1000
degree = 25

# random graphs
g1 = ng.erdos_renyi(num_nodes, avg_deg=degree)
g2 = ng.erdos_renyi(num_nodes, avg_deg=degree, directed=False)  # the same graph
                                                                # but undirected

# 2-step generation of a scale-free with Gaussian weight distribution

w = {
    "distribution": "gaussian",
    "distrib_prop": {"avg": 60., "std":5.}
}
g3 = nngt.Graph(num_nodes, weights=w)
ng.random_scale_free(2.2, 2.9, from_graph=g3)
```

```
# same in 1 step
g4 = ng.random_scale_free(2.2, 2.9, nodes=num_nodes, weights=w)
```

Networks composed of heterogeneous groups

```python
# ------------------ #
# Make the population #
# ------------------ #

# two groups of neurons
g1 = nngt.NeuralGroup(range(500))              # neurons 0 to 499
g2 = nngt.NeuralGroup(range(500, num_nodes))   # neurons 500 to 999

# make population (without NEST models)
pop = nngt.NeuralPop.from_groups(
    (g1, g1), ("left", "right"), with_models=False)

# create network from this population
net = nngt.Network(population=pop)


# ------------------ #
# Connect the groups #
# ------------------ #

# inter-groups (Erdos-Renyi)
prop_er1 = {"density": 0.035}
ng.connect_neural_groups(net, "left", "right", "erdos_renyi", prop_er1)

# intra-groups (Newmann-Watts)
prop_nw = {
    "coord_nb": 10,
    "proba_shortcut": 0.1
}
ng.connect_neural_groups(net, "left", "left", "erdos_renyi", prop_er1)
ng.connect_neural_groups(net, "right", "right", "erdos_renyi", prop_er1)
```

Use with NEST

Generating a network with excitatory and inhibitory neurons:

```python
# 800 excitatory neurons, 200 inhibitory
net = nngt.Network.ei_network(num_nodes, ei_ratio=0.2)

'''
Connect the populations.
'''
# exc -> inhib (Erdos-Renyi)
prop_er1 = {"density": 0.035}
ng.connect_neural_types(net, 1, -1, "erdos_renyi", prop_er1)

# exc -> exc (Newmann-Watts)
prop_nw = {
```

```
    "coord_nb": 10,
    "proba_shortcut": 0.1
}
ng.connect_neural_types(net, 1, 1, "newman_watts", prop_nw)

# inhib -> exc (Random scale-free)
prop_rsf = {
    "in_exp": 2.1,
    "out_exp": 2.6,
    "density": 0.2
}
ng.connect_neural_types(net, -1, 1, "random_scale_free", prop_rsf)

# inhib -> inhib (Erdos-Renyi)
prop_er2 = { "density": 0.04 }
ng.connect_neural_types(net, -1, -1, "erdos_renyi", prop_er2)
```

Send the network to NEST:

```python
import nest
from nngt.simulation import monitor_groups, plot_activity, set_poisson_input

'''
Prepare the network and devices.
'''
# send to NEST,
gids = net.to_nest()
# excite
set_poisson_input(gids, rate=100000.)
# record
groups = [key for key in net.population]
recorder, record = monitor_groups(groups, net)

'''
Simulate and plot.
'''
simtime = 100.
nest.Simulate(simtime)

plot_activity(
    recorder, record, network=net, show=True, hist=False,
    limits=(0,simtime))
```

Advanced examples

Receptor ports in NEST

Some models, such as multisynaptic neurons, or advanced models incorporating various neurotransmitters use an additional information, called `"port"` to identify the synapse that will be used by the `nest.Connect` method. These models can also be used with NNGT by telling the *NeuralGroup* which type of port the neuron should try to bind to.

NB: the port is specified in the **source** neuron and declares which synapse of the **target** neuron is concerned.

```
'''
Build a network with two populations:
```

```python
* excitatory (80%)
* inhibitory (20%)
'''
num_neurons = 50    # number of neurons
avg_degree  = 20    # average number of neighbours
std_degree  = 3     # deviation for the Gaussian graph

# parameters
neuron_model = "ht_neuron"      # hill-tononi model
exc_syn = {'receptor_type': 1}  # 1 is 'AMPA' in this model
inh_syn = {'receptor_type': 3}  # 3 is 'GABA_A' in this model

pop = nngt.NeuralPop.exc_and_inhib(
    num_neurons, en_model=neuron_model, in_model=neuron_model,
    es_param=exc_syn, is_param=inh_syn)

# create the network and send it to NEST
w_prop = {"distribution": "gaussian", "avg": 1., "std": .2}
net = nngt.generation.gaussian_degree(
    avg_degree, std_degree, population=pop, weights=w_prop)

'''
Send to NEST and set excitation and recorders
'''
import nest
from nngt.simulation import monitor_groups, plot_activity, set_noise

gids = net.to_nest()

# add noise to the excitatory neurons
excs = list(pop["excitatory"].nest_gids)
set_noise(excs, 10., 2.)

# record
groups = [key for key in net.population]
recorder, record = monitor_groups(groups, net)

'''
Simulate and plot.
'''
simtime = 2000.
nest.Simulate(simtime)

plot_activity(
    recorder, record, network=net, show=True, hist=False,
    limits=(0, simtime))
```

Properties of graph components

> **Warning:** This section is not up to date anymore!

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, .

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.

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.

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.

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`.

Activity analysis

Principle

The interesting fact about having a link between the graph and the simulation is that you can easily analyze the activity be taking into account what you know from the graph structure.

Sorted rasters

Rater plots can be sorted depending on some specific node property, e.g. the degree or the betweenness:

```python
import nest

import nngt
from nngt.simulation import monitor_nodes, plot_activity

pop = nngt.NeuralPop.uniform(1000, neuron_model="aeif_psc_alpha")
net = nngt.generation.gaussian_degree(100, 20, population=pop)

nodes = net.to_nest()
recorders, recordables = monitor_nodes(nodes)
simtime = 1000.
nest.Simulate(simtime)

fignums = plot_activity(
    recorders, recordables, network=net, show=True, hist=False,
    limits=(0.,simtime), sort="in-degree")
```

Activity properties

NNGT can also be used to analyze the general properties of a raster.

Either from a .gdf file containing the raster data

```python
import nngt
from nngt.simulation import analyze_raster

a = analyze_raster("path/to/raster.gdf")
print(a.phases)
print(a.properties)
```

Or from a spike detector gid `sd`:

```python
a = analyze_raster(sd)
```

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.**activity_types**(*spike_detector*, *limits*, *network=None*, *phase_coeff=(0.5, 10.0)*,
mbis=0.5, *mfb=0.2*, *mflb=0.05*, *skip_bursts=0*, *simplify=False*,
fignums=[], *show=False*)

Analyze the spiking pattern of a neural network.

@todo: think about inserting t=0. and t=simtime at the beginning and at the end of `times`.

> **Parameters**
>
> - **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).
>
> - **network** (`Network`, optional (default: None)) – Neural network that was analyzed
>
> - **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 partici-
>   pate 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*).
>
> - **skip_bursts** (*int, optional (default: 0)*) – Skip the *skip_bursts* first bursts to consider only
>   the permanent regime.
>
> - **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.

Note: Effects of *skip_bursts* and *limits[0]* are cumulative: the *limits[0]* first milliseconds are ignored, then the
skip_bursts first bursts of the remaining activity are ignored.

> **Returns** **phases** (*dict*) – Dictionary containing the time intervals (in ms) for all four phases (*burst-
> ing', 'quiescent', 'mixed', and 'localized*) as lists. E.g: `phases["bursting"]` could give
> `[[123.5,334.2], [857.1,1000.6]]`.

nngt.simulation.**get_nest_network**(*id_converter=None*)

Get the adjacency matrix describing a NEST network.

> > **Parameters 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.**make_nest_network**(*network*, *use_weights=True*)

> Create a new network 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 gids** (*tuple (nodes in NEST)*) – GIDs of the neurons in the network.

nngt.simulation.**monitor_groups**(*group_names*, *network*, *nest_recorder=None*, *params=None*)

> Monitoring the activity of nodes in the network.
>
> > **Parameters**
> >
> > - **group_name** (*list of strings*) – Names of the groups that should be recorded.
> >
> > - **network** (`Network` or subclass) – Network which population will be used to differentiate groups.
> >
> > - **nest_recorder** (*strings or list, optional (default: "spike_detector"0)*) – Device(s) to monitor the network.
> >
> > - **params** (dict or list of, optional (default: {})) – Dictionarie(s) containing the parameters for each recorder (see NEST documentation for details).
> >
> > **Returns**
> >
> > - **recorders** (*tuple*) – Tuple of the recorders' gids
> >
> > - **recordables** (*tuple*) – Typle of the recordables' names.

nngt.simulation.**monitor_nodes**(*gids*, *nest_recorder=None*, *params=None*, *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** (*strings or list, optional (default: "spike_detector")*) – Device(s) to monitor the network.
> >
> > - **params** (dict or list of, optional (default: {})) – Dictionarie(s) containing the parameters for each recorder (see NEST documentation for details).
> >
> > - **network** (`Network` or subclass, optional (default: None)) – Network which population will be used to differentiate groups.
> >
> > **Returns**
> >
> > - **recorders** (*tuple*) – Tuple of the recorders' gids
> >
> > - **recordables** (*tuple*) – Typle of the recordables' names.

---

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.**set_step_currents**(*gids*, *times*, *currents*)

> Set step-current excitations

> > **Parameters**
> >
> > - **gids** (*tuple*) – NEST gids of the target neurons.
> >
> > - **times** (list or `numpy.ndarray`) – List of the times where the current will change (by default the current generator is initiated at I=0. for t=0.)
> >
> > - **currents** (list or `numpy.ndarray`) – List of the new current value after the associated time value in *times*.
> >
> > **Returns noise** (*tuple*) – The NEST gid of the noise_generator.

nngt.simulation.**plot_activity**(*gid_recorder=None*, *record=None*, *network=None*, *gids=None*, *show=True*, *limits=None*, *hist=True*, *title=None*, *fignum=None*, *label=None*, *sort=None*, *normalize=1.0*, *decimate=None*)

> Plot the monitored activity.

> > **Parameters**
> >
> > - **gid_recorder** (*tuple or list, optional (default: None)*) – The gids of the recording devices. If None, then all existing "spike_detector"s are used.
> >
> > - **record** (*tuple or list, optional (default: None)*) – List of the monitored variables for each device. If *gid_recorder* is None, record can also be None and only spikes are considered.
> >
> > - **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).

- **title** (*str, optional (default: None)*) – Title of the plot.

- **fignum** (*int, optional (default: None)*) – Plot the activity on an existing figure (from `figure.number`).

- **label** (*str, optional (default: None)*) – Add a label to the plot.

- **sort** (*str or list, optional (default: None)*) – Sort neurons using a topological property ("in-degree", "out-degree", "total-degree" or "betweenness"), an activity-related property ("firing_rate") or a user-defined list of sorted neuron ids. Sorting is performed by increasing value of the *sort* property from bottom to top inside each group.

- **normalize** (*float, optional (default: None)*) – Normalize the recorded results by a given float.

- **decimate** (*int or list of ints, optional (default: None)*) – Represent only a fraction of the spiking neurons; only one neuron in *decimate* will be represented (e.g. setting *decimate* to 5 will lead to only 20% of the neurons being represented). If a list is provided, it must have one entry per NeuralGroup in the population.

> **Warning:** Sorting with "firing_rate" only works if NEST gids form a continuous integer range.

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

`nngt.simulation.`**`raster_plot`**(*times*, *senders*, *limits=None*, *title='Spike raster'*, *hist=False*, *num_bins=1000*, *color='b'*, *decimate=None*, *fignum=None*, *label=None*, *show=True*, *sort=None*, *sort_attribute=None*)
    Plotting routine that constructs a raster plot along with an optional histogram.

> **Parameters**

- **times** (list or `numpy.ndarray`) – Spike times.

- **senders** (list or `numpy.ndarray`) – Index for the spiking neuron for each time in *times*.

- **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.

- **color** (*string or float, optional (default: 'b')*) – Color of the plot lines and markers.

- **decimate** (*int, optional (default: None)*) – Represent only a fraction of the spiking neurons; only one neuron in *decimate* will be represented (e.g. setting *decimate* to 10 will lead to only 10% of the neurons being represented).

- **fignum** (*int, optional (default: None)*) – Id of another raster plot to which the new data should be added.

- **label** (*str, optional (default: None)*) – Label the current data.

- **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.

Multithreading

Principle

The NNGT package provides the possibility to use multithreaded algorithms to generate networks. This feature means that the computation is distributed on several CPUs and can be useful for:

- machines with several cores but low frequency
- generation functions requiring large amounts of computation
- very large graphs

However, the multithreading part concerns only the generation of the edges; if a graph library such as `graph-tool`, `igraph`, or `networkx` is used, the building process of the graph object will be taken care of by this library. Since this process is not multithreaded, obtaining the graph object can be much longer than the actual generation process.

Use

Setting multithreading

Multithreading in NNGT can be set via

```
>>> nngt.set_config({"multithreading": True, "omp": num_omp_threads})
```

and you can then switch it off using

```
>>> nngt.set_config("multithreading", False)
```

This will automatically switch between the standard and multithreaded algorithms for graph generation.

Random seeds for multithreading

@Todo

Graph-tool caveat

The `graph-tool` library also provides some multithreading capabilities, using

```
>>> graph_tool.openmp_set_num_threads(num_omp_threads)
```

However, this sets the number of OpenMP threads session-wide, which means that **it will interfere with the ''NEST'' setup!** Hence, if you are working with both `NEST` and `graph-tool`, **you have to use the same number of OpenMP threads in both libraries**.

To prevent bad surprises as much as possible, NNGT will raise an error if a value of `"omp"` is provided, which differs from the current NEST configuration. Regardless of this precaution, keeping only one value for the number of threads and using it consistently throughout the code is strongly advised.

Description

Neural networks are described by four graph classes which inherit from the main class of the chosen graph library (`graph_tool.Graph`, `igraph.Graph` or `networkx.DiGraph`):

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

Using these objects, the user can access to 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 the initial library's methods but always using those of the previously listed classes. If for some reason you should directly use the methods from the graph library on the graph, make sure they do not modify its structure; any modification performed from a method other than those of the *Graph* subclasses will lead to undefined 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, for *Network*, 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.

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 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.

  Initialize Graph instance

  Parameters

  - **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.
  - **name** (*string, optional (default: "Graph")*) – The name of this *Graph* instance.
  - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
  - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
  - **from_graph** (*GraphObject*, optional) – An optional *GraphObject* to serve as base.

- **kwargs** (*optional keywords arguments*) – Optional arguments that can be passed to the graph, e.g. a dict containing information on the synaptic weights (weights={"distribution": "constant", "value": 2.3} which is equivalent to weights=2.3), the synaptic *delays*, or a type information.

> **Returns** self (*Graph*)

attributes(*edge=None*, *name=None*)
>   Attributes of the graph's edges.

>>   **Parameters**

- **edge** (tuple, optional (default: None)) – Edge whose attribute should be displayed.

- **name** (str, optional (default: None)) – Name of the desired attribute.

>>   **Returns**

- *List containing the names of the graph's attributes (synaptic weights,*

- delays...) if *edge* is None, else a dict containing the

- attributes of the edge (or the value of attribute *name* if it is not

- None).

copy()
>   Returns a deepcopy of the current *Graph* instance

edge_attribute
>   Access edge attributes

static **from_file**(*filename*, *format='auto'*, *delimiter=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*, *ignore='#'*, *from_string=False*)
>   Import a saved graph 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 is 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.

> - **from_string** (*bool, optional (default: False)*) – Load from a string instead of a file.

> **Returns graph** ([`Graph`](#) or subclass) – Loaded graph.

classmethod **from_library** (*library_graph*, *weighted=True*, *directed=True*, ***kwargs*)

classmethod **from_matrix** (*matrix*, *weighted=True*, *directed=True*)
> Creates a [`Graph`](#) from a `scipy.sparse` matrix or a dense matrix.

> **Parameters**

> - **matrix** (`scipy.sparse` matrix or `numpy.array`) – Adjacency matrix.

> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.

> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.

> **Returns** [`Graph`](#)

get_attribute_type (*attribute_name*)
> Return the type of an attribute

get_betweenness (*btype='both'*, *use_weights=False*)
> Betweenness centrality sequence of all nodes and edges.

> **Parameters**

> - **btype** (str, optional (default: `"both"`)) – Type of betweenness to return (`"edge"`, `"node"`-betweenness, or `"both"`).

> - **use_weights** (*bool, optional (default: False)*) – Whether to use weighted (True) or simple degrees (False).

> **Returns**

> - **node_betweenness** (`numpy.array`) – Betweenness of the nodes (if *btype* is `"node"` or `"both"`).

> - **edge_betweenness** (`numpy.array`) – Betweenness of the edges (if *btype* is `"edge"` or `"both"`).

get_degrees (*deg_type='total'*, *node_list=None*, *use_weights=False*)
> Degree sequence of all the nodes.

> **Parameters**

> - **deg_type** (*string, optional (default: "total")*) – Degree type (among 'in', 'out' or 'total').

> - **node_list** (*list, optional (default: None)*) – List of the nodes which degree should be returned

> - **use_weights** (*bool, optional (default: False)*) – Whether to use weighted (True) or simple degrees (False).

> **Returns** `numpy.array` or None (if an invalid type is asked).

get_delays ()
> Returns the delay adjacency matrix as a `scipy.sparse.lil_matrix` if delays are present; else raises an error.

get_density()
    Density of the graph: $\frac{E}{N^2}$, where E is the number of edges and N the number of nodes.

get_edge_types()

get_graph_type()
    Return the type of the graph (see nngt.generation)

get_name()
    Get the name of the graph

get_weights()
    Returns the weighted adjacency matrix as a `scipy.sparse.lil_matrix`.

graph_id
    Unique `int` identifying the instance.

is_directed()
    Whether the graph is directed or not

is_network()
    Whether the graph is a subclass of *Network* (i.e. if it has a *NeuralPop* attribute).

is_spatial()
    Whether the graph is embedded in space (i.e. if it has a *Shape* attribute). Returns `True` is the graph is a subclass of *SpatialGraph*.

is_weighted()
    Whether the edges have weights

static **make_network**(*graph*, *neural_pop*, *copy=False*, ***kwargs*)
    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*
> - **copy** (bool, optional (default: `False`)) – Whether the operation should be made in-place on the object or if a new object should be returned.

> **Notes**
>
> In-place operation that directly converts the original graph if *copy* is `False`, else returns the copied *Graph* turned into a *Network*.

static **make_spatial**(*graph*, *shape*, *positions=None*, *copy=False*)
    Turn a *Graph* object into a *SpatialGraph*, or a *Network* into a *SpatialNetwork*.

> **Parameters**
>
> - **graph** (*Graph* or *SpatialGraph*) – Graph to convert.
> - **shape** (*Shape*, optional) – Shape to associate to the new *SpatialGraph*.
> - **positions** (*(N, 2) array*) – Positions, in a 2D space, of the N neurons.
> - **copy** (bool, optional (default: `False`)) – Whether the operation should be made in-place on the object or if a new object should be returned.

Notes

> In-place operation that directly converts the original graph if *copy* is `False`, else returns the copied *Graph* turned into a *SpatialGraph*.

name
> Name of the graph.

node_attributes
> Access node attributes

classmethod num_graphs()
> Returns the number of alive instances.

set_delays(*delay=None*, *elist=None*, *distribution=None*, *parameters=None*, *noise_scale=None*)
> Set the delay for spike propagation between neurons. ..todo :: take elist into account in Connections.delays

> **Parameters**

> - **delay** (float or class:*numpy.array*, optional (default: None)) – Value or list of delays (for user defined delays).

> - **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined delays).

> - **distribution** (class:*string*, optional (default: None)) – Type of distribution (choose among "constant", "uniform", "gaussian", "lognormal", "lin_corr", "log_corr").

> - **parameters** (*dict, optional (default: {})*) – Dictionary containing the properties of the delay distribution.

> - **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the delays.

set_edge_attribute(*attribute*, *values=None*, *val=None*, *value_type=None*, *edges=None*)
> Set attributes to the connections between neurons.

> > **Warning:** The special "type" attribute cannot be modified when using graphs that inherit from the *Network* class. This is because for biological networks, neurons make only one kind of synapse, which is determined by the `nngt.NeuralGroup` they belong to.

set_name(*name=''*)
> set graph name

set_types(*syn_type*, *nodes=None*, *fraction=None*)
> Set the synaptic/connection types.

> > **Warning:** The special "type" attribute cannot be modified when using graphs that inherit from the *Network* class. This is because for biological networks, neurons make only one kind of synapse, which is determined by the `nngt.NeuralGroup` they belong to.

> **Parameters**

> - **syn_type** (*int or string*) – Type of the connection among 'excitatory' (also *1*) or 'inhibitory' (also *-1*).

> - **nodes** (int, float or list, optional (default: *None*)) – If *nodes* is an int, number of nodes of the required type that will be created in the graph (all connections from inhibitory nodes

are inhibitory); if it is a float, ratio of *syn_type* nodes in the graph; if it is a list, ids of the *syn_type* nodes.

  • **fraction** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as *syn_type* (if *nodes* is not *None*, it is the fraction of the specified nodes' edges, otherwise it is the fraction of all edges in the graph).

Returns t_list (`numpy.ndarray`) – List of the types in an order that matches the *edges* attribute of the graph.

set_weights(*weight=None*, *elist=None*, *distribution=None*, *parameters=None*, *noise_scale=None*)
Set the synaptic weights.

..todo :: take elist into account in Connections.weights

Parameters

  • **weight** (float or class:*numpy.array*, optional (default: None)) – Value or list of the weights (for user defined weights).

  • **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined weights).

  • **distribution** (class:*string*, optional (default: None)) – Type of distribution (choose among "constant", "uniform", "gaussian", "lognormal", "lin_corr", "log_corr").

  • **parameters** (*dict, optional (default: {})*) – Dictionary containing the properties of the weight distribution.

  • **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

to_file(*filename*, *format='auto'*, *delimiter=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*)
Save graph to file; options detailed below.

See also:

`nngt.lib.save_to_file()` function for options.

type
Type of the graph.

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 spatial graphs (i.e. graph where the structure is embedded in space.

Initialize SpatialClass instance. .. todo:: see what we do with the from_graph argument

Parameters

  • **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.

  • **name** (*string, optional (default: "Graph")*) – The name of this `Graph` instance.

  • **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.

  • **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.

  • **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment (None leads to a square of side 1 cm)

  • **positions** (`numpy.array` (N, 2), optional (default: None)) – Positions of the neurons; if not specified and *nodes* is not 0, then neurons will be reparted at random inside the `Shape` object of the instance.

> **Returns** **self** (SpatialGraph)

get_positions(*neurons=None*)

> Returns the neurons' positions as a (N, 2) array.
>
> > **Parameters** **neurons** (*int or array-like, optional (default: all neurons)*) – List of the neurons for which the position should be returned.

shape

class nngt.**Network**(*name='Network'*, *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.

Initializes *Network* instance.

> **Parameters**
>
> - **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.
> - **name** (*string, optional (default: "Graph")*) – The name of this *Graph* instance.
> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
> - **from_graph** (*GraphObject*, optional (default: None)) – An optional *GraphObject* to serve as base.
> - **population** (*nngt.NeuralPop*, (default: None)) – An object containing the neural groups and their properties: model(s) to use in NEST to simulate the neurons as well as their parameters.
>
> **Returns** **self** (Network)

classmethod **ei_network**(*size*, *ei_ratio=0.2*, *en_model='aeif_cond_alpha'*, *en_param=None*, *es_model='static_synapse'*, *es_param=None*, *in_model='aeif_cond_alpha'*, *in_param=None*, *is_model='static_synapse'*, *is_param=None*)

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.

get_neuron_type(*neuron_ids*)

Return the type of the neurons (+1 for excitatory, -1 for inhibitory).

Parameters **neuron_ids** (*int or tuple*) – NEST gids.

Returns **ids** (*int or tuple*) – Ids in the network. Same type as the requested *gids* type.

id_from_nest_gid(*gids*)

Return the ids of the nodes in the [`nngt.Network`](#) instance from the corresponding NEST gids.

Parameters **gids** (*int or tuple*) – NEST gids.

Returns **ids** (*int or tuple*) – Ids in the network. Same type as the requested *gids* type.

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's properties.*

classmethod num_networks()

Returns the number of alive instances.

population

[`NeuralPop`](#) that divides the neurons into groups with specific properties.

set_types(*syn_type*, *nodes=None*, *fraction=None*)

to_nest(*use_weights=True*)

Send the network to NEST.

See also:

[`make_nest_network()`](#) for parameters

classmethod uniform_network(*size*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*)

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.

Initialize Graph instance

> **Parameters**
>
> - **name** (*string, optional (default: "Graph")*) – The name of this [`Graph`](#) instance.
> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
> - **shape** ([`Shape`](#), optional (default: None)) – Shape of the neurons' environment (None leads to a square of side 1 cm)
> - **positions** (`numpy.array`, optional (default: None)) – Positions of the neurons; if not specified and *nodes* != 0, then neurons will be reparted at random inside the [`Shape`](#) object of the instance.
> - **population** (class:~*nngt.NeuralPop*, optional (default: None))
>
> **Returns self** ([`SpatialNetwork`](#))

> **`set_types`** (*syn_type*, *nodes=None*, *fraction=None*)

Side classes

class `nngt.geometry.`**`Shape`** (*unit='um'*, *parent=None*)

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

..warning : With this backup shape, only a rectangle or a disk can be created.

`area`

> *double* – Area of the shape in mm^2.

`centroid`

> *tuple of doubles* – Position of the center of mass of the current shape.

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

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 'um')*) – Unit in the metric system among 'um', 'mm', 'cm', 'dm', 'm'

`area`

> Area of the shape.

`centroid`

> Centroid of the shape.

`coords`

classmethod `disk` (*radius*, *centroid=(0.0, 0.0)*, *unit='um'*, *parent=None*)

Generate a disk of given radius and center (*centroid*).

Parameters

- **height** (*float*) – Height of the rectangle.

- **width** (*float*) – Width of the rectangle.

- **centroid** (*tuple of floats, optional (default: (0., 0.)))* – Position of the rectangle's center of mass.

- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'.

- **parent** (`nngt.Graph` or subclass) – The graph which is associated to this Shape.

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

geom_type

parent

　　Return the parent of the `Shape`.

classmethod **rectangle** (*height*, *width*, *centroid=(0.0, 0.0)*, *unit='um'*, *parent=None*)

　　Generate a rectangle of given height, width and center of mass.

Parameters

- **height** (*float*) – Height of the rectangle.

- **width** (*float*) – Width of the rectangle.

- **centroid** (*tuple of floats, optional (default: (0., 0.)))* – Position of the rectangle's center of mass.

- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'.

- **parent** (`nngt.Graph` or subclass) – The graph which is associated to this Shape.

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

seed_neurons (*nodes=None*)

　　Return the positions of the neurons inside the `Shape`.

Parameters

- **unit** (*string (default: None)*) – Unit in which the positions of the neurons will be returned, among 'um', 'mm', 'cm', 'dm', 'm'.

- **neurons** (*int, optional (default: None)*) – Number of neurons to seed. This argument is considered only if the `Shape` has no *parent*, otherwise, a position is generated for each neuron in *parent*.

Returns **positions** (*array of double with shape (N, 2)*)

set_parent (*parent*)

unit

　　Return the unit for the `Shape` coordinates.

class nngt.**NeuralPop** (*size*, *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.

Initialize NeuralPop instance

Parameters

- **size** (*int*) – Number of neurons that the population will contain.

- **parent** (`Network`, optional (default: None)) – Network associated to this population.

- **with_models** (`bool`) – whether the population's groups contain models to use in NEST

- ****kwargs** (`dict`)

Returns pop (`NeuralPop` object.)

add_to_group(*group_name*, *id_list*)

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

create_group(*name*, *neurons*, *ntype=1*, *neuron_model=None*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*)
  Create a new groupe from given properties.

  Parameters

  - **name** (*str*) – Name of the group.

  - **neurons** (*array-like*) – List of the neurons indices.

  - **ntype** (*int, optional (default: 1)*) – Type of the neurons : 1 for excitatory, -1 for inhibitory.

  - **neuron_model** (*str, optional (default: None)*) – Name of a neuron model in NEST.

  - **neuron_param** (*dict, optional (default: None)*) – Parameters for *neuron_model* in the NEST simulator. If None, default parameters will be used.

  - **syn_model** (*str, optional (default: "static_synapse")*) – Name of a synapse model in NEST.

  - **syn_param** (*dict, optional (default: None)*) – Parameters for *syn_model* in the NEST simulator. If None, default parameters will be used.

classmethod **exc_and_inhib**(*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.

classmethod **from_groups**(*groups*, *names=None*, *parent=None*, *with_models=True*)
  Make a NeuralPop object from a (list of) `NeuralGroup` object(s).

classmethod **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`.

get_group(*neurons*, *numbers=False*)
  Return the group of the neurons.

  Parameters

  - **neurons** (*int or array-like*) – IDs of the neurons for which the group should be returned.

  - **numbers** (*bool, optional (default: False)*) – Whether the group identifier should be returned as a number; if `False`, the group names are returned.

get_param(*groups=None*, *neurons=None*, *element='neuron'*)
  Return the *element* (neuron or synapse) parameters for neurons or groups of neurons in the population.

  Parameters

- **groups** (str, int or array-like, optional (default: None)) – Names or numbers of the groups for which the neural properties should be returned.

- **neurons** (int or array-like, optional (default: None)) – IDs of the neurons for which parameters should be returned.

- **element** (list of str, optional (default: "neuron")) – Element for which the parameters should be returned (either "neuron" or "synapse").

Returns param (list) – List of all dictionaries with the elements' parameters.

has_models

is_valid

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.

>   **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.

>   **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.

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**(*size*, *parent=None*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*)
>   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

Available subpackages

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

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

get_config Show library _configuration

set_config Set library _configuration (graph library, multithreading...)

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 classes and functions

class `nngt.`**`Connections`**
    The basic class that computes the properties of the connections between neurons for graphs.

    static **`delays`**(*graph=None*, *dlist=None*, *elist=None*, *distribution='constant'*, *parameters=None*,
            noise_scale=None)
        Compute the delays of the neuronal connections.

            Parameters

                - **graph** (class:~*nngt.Graph* or subclass) – Graph the nodes belong to.

                - **dlist** (class:*numpy.array*, optional (default: None)) – List of user-defined delays).

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges which value should be updated.

- **distribution** (class:*string*, optional (default: "constant")) – Type of distribution (choose among "constant", "uniform", "lognormal", "gaussian", "user_def", "lin_corr", "log_corr").

- **parameters** (class:*dict*, optional (default: {})) – Dictionary containing the distribution parameters.

- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

Returns new_delays (class:*scipy.sparse.lil_matrix*) – A sparse matrix containing *ONLY* the newly-computed weights.

static **distances** (*graph*, *elist=None*, *pos=None*, *dlist=None*, *overwrite=False*)
 Compute the distances between connected nodes in the graph. Try to add only the new distances to the graph. If they overlap with previously computed distances, recomputes everything.

 Parameters

- **graph** (class:*~nngt.Graph* or subclass) – Graph the nodes belong to.

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges.

- **pos** (class:*numpy.array*, optional (default: None)) – Positions of the nodes; note that if *graph* has a "position" attribute, *pos* will not be taken into account.

- **dlist** (class:*numpy.array*, optional (default: None)) – List of distances (for user-defined distances)

Returns new_dist (class:*numpy.array*) – Array containing *ONLY* the newly-computed distances.

static **types** (*graph*, *inhib_nodes=None*, *inhib_frac=None*)
 @todo

 Define the type of a set of neurons. If no arguments are given, all edges will be set as excitatory.

 Parameters

- **graph** (`Graph` or subclass) – Graph on which edge types will be created.

- **inhib_nodes** (int, float or list, optional (default: *None*)) – If *inhib_nodes* is an int, number of inhibitory nodes in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of inhibitory nodes in the graph; if it is a list, ids of the inhibitory nodes.

- **inhib_frac** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as refractory (if *inhib_nodes* is not *None*, it is the fraction of the nodes' edges that will become inhibitory, otherwise it is the fraction of all the edges in the graph).

Returns t_list (`ndarray`) – List of the edges' types.

static **weights** (*graph=None*, *elist=None*, *wlist=None*, *distribution='constant'*, *parameters={}*, *noise_scale=None*)
 Compute the weights of the graph's edges. @todo: take elist into account

 Parameters

- **graph** (class:*~nngt.Graph* or subclass) – Graph the nodes belong to.

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined weights).

- **wlist** (class:*numpy.array*, optional (default: None)) – List of the weights (for user defined weights).

- **distribution** (class:*string*, optional (default: "constant")) – Type of distribution (choose among "constant", "uniform", "lognormal", "gaussian", "user_def", "lin_corr", "log_corr").

- **parameters** (class:*dict*, optional (default: {})) – Dictionary containing the distribution parameters.

- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

Returns new_weights (class:*scipy.sparse.lil_matrix*) – A sparse matrix containing *ONLY* the newly-computed weights.

nngt.**generate**(*di_instructions*, ***kwargs*)

Generate a [`Graph`](Graph) or one of its subclasses from a `dict` containing all the relevant informations.

Parameters

- **di_instructions** (`dict`) – Dictionary containing the instructions to generate the graph. It must have at least `"graph_type"` in its keys, with a value among `"distance_rule"`, `"erdos_renyi"`, `"fixed_degree"`, `"newman_watts"`, `"price_scale_free"`, `"random_scale_free"`. Depending on the type, *di_instructions* should also contain at least all non-optional arguments of the generator function.

- **.. seealso** – Generator functions are detailed in [`generation`](generation).

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.

Initialize Graph instance

Parameters

- **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.

- **name** (*string, optional (default: "Graph")*) – The name of this [`Graph`](Graph) instance.

- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.

- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.

- **from_graph** ([`GraphObject`](GraphObject), optional) – An optional [`GraphObject`](GraphObject) to serve as base.

- **kwargs** (*optional keywords arguments*) – Optional arguments that can be passed to the graph, e.g. a dict containing information on the synaptic weights (`weights={"distribution": "constant", "value": 2.3}` which is equivalent to `weights=2.3`), the synaptic *delays*, or a `type` information.

Returns self ([`Graph`](Graph))

attributes(*edge=None*, *name=None*)

Attributes of the graph's edges.

Parameters

- **edge** (tuple, optional (default: `None`)) – Edge whose attribute should be displayed.

- **name** (str, optional (default: `None`)) – Name of the desired attribute.

Returns

- *List containing the names of the graph's attributes (synaptic weights,*

- delays...) if *edge* is `None`, else a `dict` containing the

- attributes of the edge (or the value of attribute *name* if it is not

- `None`).

copy()
> Returns a deepcopy of the current [`Graph`](#) instance

edge_attribute
> Access edge attributes

static **from_file**(*filename*, *format='auto'*, *delimiter=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*, *ignore='#'*, *from_string=False*)
> Import a saved graph 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 is 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`.

> - **from_string** (*bool, optional (default: False)*) – Load from a string instead of a file.

> **Returns graph** ([`Graph`](#) or subclass) – Loaded graph.

classmethod **from_matrix**(*matrix*, *weighted=True*, *directed=True*)
> Creates a [`Graph`](#) from a `scipy.sparse` matrix or a dense matrix.

> **Parameters**

> - **matrix** (`scipy.sparse` matrix or `numpy.array`) – Adjacency matrix.

> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.

> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.

> **Returns** *Graph*

get_attribute_type(*attribute_name*)
> Return the type of an attribute

get_betweenness(*btype='both'*, *use_weights=False*)
> Betweenness centrality sequence of all nodes and edges.
>
> > **Parameters**
> >
> > - **btype** (str, optional (default: `"both"`)) – Type of betweenness to return (`"edge"`, `"node"`-betweenness, or `"both"`).
> >
> > - **use_weights** (*bool, optional (default: False)*) – Whether to use weighted (True) or simple degrees (False).
> >
> > **Returns**
> >
> > - **node_betweenness** (`numpy.array`) – Betweenness of the nodes (if *btype* is `"node"` or `"both"`).
> >
> > - **edge_betweenness** (`numpy.array`) – Betweenness of the edges (if *btype* is `"edge"` or `"both"`).

get_degrees(*deg_type='total'*, *node_list=None*, *use_weights=False*)
> Degree sequence of all the nodes.
>
> > **Parameters**
> >
> > - **deg_type** (*string, optional (default: "total")*) – Degree type (among 'in', 'out' or 'total').
> >
> > - **node_list** (*list, optional (default: None)*) – List of the nodes which degree should be returned
> >
> > - **use_weights** (*bool, optional (default: False)*) – Whether to use weighted (True) or simple degrees (False).
> >
> > **Returns** `numpy.array` or None (if an invalid type is asked).

get_delays()
> Returns the delay adjacency matrix as a `scipy.sparse.lil_matrix` if delays are present; else raises an error.

get_density()
> Density of the graph: $\frac{E}{N^2}$, where *E* is the number of edges and *N* the number of nodes.

get_graph_type()
> Return the type of the graph (see nngt.generation)

get_name()
> Get the name of the graph

get_weights()
> Returns the weighted adjacency matrix as a `scipy.sparse.lil_matrix`.

graph_id
> Unique *int* identifying the instance.

is_directed()
> Whether the graph is directed or not

is_network()
> Whether the graph is a subclass of *Network* (i.e. if it has a *NeuralPop* attribute).

is_spatial()
> Whether the graph is embedded in space (i.e. if it has a *Shape* attribute). Returns `True` is the graph is a subclass of *SpatialGraph*.

is_weighted()
> Whether the edges have weights

static **make_network**(*graph*, *neural_pop*, *copy=False*, ***kwargs*)
> 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*
> >
> > - **copy** (bool, optional (default: `False`)) – Whether the operation should be made in-place on the object or if a new object should be returned.
>
> > **Notes**
>
> > In-place operation that directly converts the original graph if *copy* is `False`, else returns the copied *Graph* turned into a *Network*.

static **make_spatial**(*graph*, *shape*, *positions=None*, *copy=False*)
> Turn a *Graph* object into a *SpatialGraph*, or a *Network* into a *SpatialNetwork*.
>
> > **Parameters**
> >
> > - **graph** (*Graph* or *SpatialGraph*) – Graph to convert.
> >
> > - **shape** (*Shape*, optional) – Shape to associate to the new *SpatialGraph*.
> >
> > - **positions** (*(N, 2) array*) – Positions, in a 2D space, of the N neurons.
> >
> > - **copy** (bool, optional (default: `False`)) – Whether the operation should be made in-place on the object or if a new object should be returned.
>
> > **Notes**
>
> > In-place operation that directly converts the original graph if *copy* is `False`, else returns the copied *Graph* turned into a *SpatialGraph*.

name
> Name of the graph.

node_attributes
> Access node attributes

classmethod **num_graphs**()
> Returns the number of alive instances.

set_delays(*delay=None*, *elist=None*, *distribution=None*, *parameters=None*, *noise_scale=None*)
> Set the delay for spike propagation between neurons. ..todo :: take elist into account in Connections.delays
>
> > **Parameters**
> >
> > - **delay** (float or class:*numpy.array*, optional (default: None)) – Value or list of delays (for user defined delays).

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined delays).

- **distribution** (class:*string*, optional (default: None)) – Type of distribution (choose among "constant", "uniform", "gaussian", "lognormal", "lin_corr", "log_corr").

- **parameters** (*dict, optional (default: {})*) – Dictionary containing the properties of the delay distribution.

- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the delays.

set_edge_attribute (*attribute*, *values=None*, *val=None*, *value_type=None*, *edges=None*)
    Set attributes to the connections between neurons.

> **Warning:** The special "type" attribute cannot be modified when using graphs that inherit from the `Network` class. This is because for biological networks, neurons make only one kind of synapse, which is determined by the `nngt.NeuralGroup` they belong to.

set_name (*name=''*)
    set graph name

set_types (*syn_type*, *nodes=None*, *fraction=None*)
    Set the synaptic/connection types.

> **Warning:** The special "type" attribute cannot be modified when using graphs that inherit from the `Network` class. This is because for biological networks, neurons make only one kind of synapse, which is determined by the `nngt.NeuralGroup` they belong to.

   Parameters

- **syn_type** (*int or string*) – Type of the connection among 'excitatory' (also *1*) or 'inhibitory' (also *-1*).

- **nodes** (int, float or list, optional (default: *None*)) – If *nodes* is an int, number of nodes of the required type that will be created in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of *syn_type* nodes in the graph; if it is a list, ids of the *syn_type* nodes.

- **fraction** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as *syn_type* (if *nodes* is not *None*, it is the fraction of the specified nodes' edges, otherwise it is the fraction of all edges in the graph).

   Returns t_list (`numpy.ndarray`) – List of the types in an order that matches the *edges* attribute of the graph.

set_weights (*weight=None*, *elist=None*, *distribution=None*, *parameters=None*, *noise_scale=None*)
    Set the synaptic weights.

   ..todo :: take elist into account in Connections.weights

   Parameters

- **weight** (float or class:*numpy.array*, optional (default: None)) – Value or list of the weights (for user defined weights).

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined weights).

- **distribution** (class:*string*, optional (default: None)) – Type of distribution (choose among "constant", "uniform", "gaussian", "lognormal", "lin_corr", "log_corr").

- **parameters** (*dict, optional (default: {})*) – Dictionary containing the properties of the weight distribution.

- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

to_file(*filename*, *format='auto'*, *delimiter=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*)
    Save graph to file; options detailed below.

    See also:

    nngt.lib.save_to_file() function for options.

type
    Type of the graph.

class nngt.**GroupProperty**(*size*,   *constraints={}*,   *neuron_model=None*,   *neuron_param={}*, *syn_model=None*, *syn_param={}*)
Class defining the properties needed to create groups of neurons from an existing GraphClass or one of its subclasses.

    Variables

- **size** – int Size of the group.

- **constraints** – dict, optional (default: {}) Constraints to respect when building the NeuralGroup .

- *neuron_model* – string, optional (default: None) name of the model to use when simulating the activity of this group.

- **neuron_param** – dict, optional (default: {}) the parameters to use (if they differ from the model's defaults)

Create a new instance of GroupProperties.

Notes

The constraints can be chosen among:

- "avg_deg", "min_deg", "max_deg" (int) to constrain the total degree of the nodes

- "avg/min/max_in_deg", "avg/min/max_out_deg", to work with the in/out-degrees

- "avg/min/max_betw" (double) to constrain the betweenness centrality

- "in_shape" (*nngt.geometry.Shape*) to chose neurons inside a given spatial region

Examples

```
>>> di_constrain = { "avg_deg": 10, "min_betw": 0.001 }
>>> group_prop = GroupProperties(200, constraints=di_constrain)
```

class nngt.**Network**(*name='Network'*, *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.

Initializes `Network` instance.

> **Parameters**
>
> - **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.
>
> - **name** (*string, optional (default: "Graph")*) – The name of this `Graph` instance.
>
> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
>
> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
>
> - **from_graph** (`GraphObject`, optional (default: None)) – An optional `GraphObject` to serve as base.
>
> - **population** (`nngt.NeuralPop`, (default: None)) – An object containing the neural groups and their properties: model(s) to use in NEST to simulate the neurons as well as their parameters.
>
> **Returns self** (`Network`)

classmethod ei_network(*size*, *ei_ratio=0.2*, *en_model='aeif_cond_alpha'*, *en_param=None*, *es_model='static_synapse'*, *es_param=None*, *in_model='aeif_cond_alpha'*, *in_param=None*, *is_model='static_synapse'*, *is_param=None*)

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.

get_neuron_type(*neuron_ids*)

> Return the type of the neurons (+1 for excitatory, -1 for inhibitory).
>
> > **Parameters neuron_ids** (*int or tuple*) – NEST gids.
> >
> > **Returns ids** (*int or tuple*) – Ids in the network. Same type as the requested *gids* type.

id_from_nest_gid(*gids*)

> Return the ids of the nodes in the `nngt.Network` instance from the corresponding NEST gids.
>
> > **Parameters gids** (*int or tuple*) – NEST gids.
> >
> > **Returns ids** (*int or tuple*) – Ids in the network. Same type as the requested *gids* type.

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's properties.*

classmethod num_networks()

> Returns the number of alive instances.

population

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

to_nest(*use_weights=True*)

> Send the network to NEST.
>
> **See also:**
>
> `make_nest_network()` for parameters

classmethod uniform_network(*size*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*)

> Generate a network containing only one type of neurons.
>
> > **Parameters**
> >
> > - **size** (*int*) – Number of neurons in the network.
> > - **neuron_model** (*string, optional (default: 'aeif_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.**NeuralPop**(*size*, *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.
>
> Initialize NeuralPop instance
>
> > **Parameters**
> >
> > - **size** (*int*) – Number of neurons that the population will contain.
> > - **parent** (`Network`, optional (default: None)) – Network associated to this population.

- **with_models** (`bool`) – whether the population's groups contain models to use in NEST

- ****kwargs** (`dict`)

Returns pop (`NeuralPop` object.)

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

create_group (*name*,   *neurons*,   *ntype=1*,   *neuron_model=None*,   *neuron_param=None*,
                syn_model='static_synapse', *syn_param=None*)
    Create a new groupe from given properties.

    Parameters

    - **name** (*str*) – Name of the group.

    - **neurons** (*array-like*) – List of the neurons indices.

    - **ntype** (*int, optional (default: 1)*) – Type of the neurons : 1 for excitatory, -1 for inhibitory.

    - **neuron_model** (*str, optional (default: None)*) – Name of a neuron model in NEST.

    - **neuron_param** (*dict, optional (default: None)*) – Parameters for *neuron_model* in the
      NEST simulator. If None, default parameters will be used.

    - **syn_model** (*str, optional (default: "static_synapse")*) – Name of a synapse model in
      NEST.

    - **syn_param** (*dict, optional (default: None)*) – Parameters for *syn_model* in the NEST
      simulator. If None, default parameters will be used.

classmethod **exc_and_inhib** (*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.

classmethod **from_groups** (*groups*, *names=None*, *parent=None*, *with_models=True*)
    Make a NeuralPop object from a (list of) `NeuralGroup` object(s).

classmethod **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`.

get_group (*neurons*, *numbers=False*)
    Return the group of the neurons.

    Parameters

    - **neurons** (*int or array-like*) – IDs of the neurons for which the group should be returned.

    - **numbers** (*bool, optional (default: False)*) – Whether the group identifier should be re-
      turned as a number; if `False`, the group names are returned.

get_param (*groups=None*, *neurons=None*, *element='neuron'*)
    Return the *element* (neuron or synapse) parameters for neurons or groups of neurons in the population.

    Parameters

    - **groups** (`str`, `int` or array-like, optional (default: `None`)) – Names or numbers of the
      groups for which the neural properties should be returned.

    - **neurons** (int or array-like, optional (default: `None`)) – IDs of the neurons for which pa-
      rameters should be returned.

- **element** (`list` of `str`, optional (default: `"neuron"`)) – Element for which the parameters should be returned (either `"neuron"` or `"synapse"`).

> **Returns param** (`list`) – List of all dictionaries with the elements' parameters.

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.

> **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.

> **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.

---

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.

classmethod uniform(*size*, *parent=None*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*)
> Make a NeuralPop of identical neurons

nngt.seed(*seed=None*)
> Seed the random generator used by NNGT (i.e. the numpy *RandomState*: for details, see `numpy.random.RandomState`).

> **Parameters seed** (*int or array_like, optional*) – Seed for *RandomState*. Must be convertible to 32 bit unsigned integers.

nngt.set_config(*config*, *value=None*)
> Set NNGT's configuration.

> **Parameters**

>> - **config** (*dict or str*) – Either a full configuration dictionary or one key to be set together with its associated value.

>> - **value** (*object, optional (default: None)*) – Value associated to *config* if *config* is a key.

Examples

```
>>> nngt.set_config({'multithreading': True, 'omp': 4})
>>> nngt.set_config('multithreading', False)
```

Note: See the config file *nngt/nngt.conf.default* or *~/.nngt/nngt.conf* for details about your configuration.

See also:

`get_config()`

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 spatial graphs (i.e. graph where the structure is embedded in space.

Initialize SpatialClass instance. .. todo:: see what we do with the from_graph argument

> **Parameters**
>
> - **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.
> - **name** (*string, optional (default: "Graph")*) – The name of this `Graph` instance.
> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
> - **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment (None leads to a square of side 1 cm)
> - **positions** (`numpy.array` (N, 2), optional (default: None)) – Positions of the neurons; if not specified and *nodes* is not 0, then neurons will be reparted at random inside the `Shape` object of the instance.
>
> **Returns** **self** (SpatialGraph)

get_positions (*neurons=None*)
Returns the neurons' positions as a (N, 2) array.

> **Parameters** **neurons** (*int or array-like, optional (default: all neurons)*) – List of the neurons for which the position should be returned.

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.

Initialize Graph instance

> **Parameters**
>
> - **name** (*string, optional (default: "Graph")*) – The name of this `Graph` instance.
> - **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
> - **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
> - **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment (None leads to a square of side 1 cm)

- **positions** (numpy.array, optional (default: None)) – Positions of the neurons; if not specified and *nodes* != 0, then neurons will be reparted at random inside the *Shape* object of the instance.

- **population** (class:~*nngt.NeuralPop*, optional (default: None))

   Returns self (*SpatialNetwork*)

nngt.**use_library**(*library*, *reloading=True*)

   Allows the user to switch to a specific graph library.

   Parameters

- **library** (*string*) – Name of a graph library among 'graph_tool', 'igraph', 'networkx'.

- **reload_moduleing** (*bool, optional (default: True)*) – Whether the graph objects should be reload_moduleed (this should always be set to True except when NNGT is first initiated!)

Core module

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

Content

class nngt.core.**Connections**

   The basic class that computes the properties of the connections between neurons for graphs.

   static **delays**(*graph=None*, *dlist=None*, *elist=None*, *distribution='constant'*, *parameters=None*, *noise_scale=None*)

   Compute the delays of the neuronal connections.

   Parameters

- **graph** (class:~*nngt.Graph* or subclass) – Graph the nodes belong to.

- **dlist** (class:*numpy.array*, optional (default: None)) – List of user-defined delays).

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges which value should be updated.

- **distribution** (class:*string*, optional (default: "constant")) – Type of distribution (choose among "constant", "uniform", "lognormal", "gaussian", "user_def", "lin_corr", "log_corr").

- **parameters** (class:*dict*, optional (default: {})) – Dictionary containing the distribution parameters.

- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

   Returns new_delays (class:*scipy.sparse.lil_matrix*) – A sparse matrix containing *ONLY* the newly-computed weights.

   static **distances**(*graph*, *elist=None*, *pos=None*, *dlist=None*, *overwrite=False*)

   Compute the distances between connected nodes in the graph. Try to add only the new distances to the graph. If they overlap with previously computed distances, recomputes everything.

   Parameters

- **graph** (class:~*nngt.Graph* or subclass) – Graph the nodes belong to.

- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges.

- **pos** (class:*numpy.array*, optional (default: None)) – Positions of the nodes; note that if *graph* has a "position" attribute, *pos* will not be taken into account.

- **dlist** (class:*numpy.array*, optional (default: None)) – List of distances (for user-defined distances)

   Returns new_dist (class:*numpy.array*) – Array containing *ONLY* the newly-computed distances.

static **types** (*graph*, *inhib_nodes=None*, *inhib_frac=None*)
   @todo

   Define the type of a set of neurons. If no arguments are given, all edges will be set as excitatory.

   Parameters

   - **graph** (*Graph* or subclass) – Graph on which edge types will be created.

   - **inhib_nodes** (int, float or list, optional (default: *None*)) – If *inhib_nodes* is an int, number of inhibitory nodes in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of inhibitory nodes in the graph; if it is a list, ids of the inhibitory nodes.

   - **inhib_frac** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as refractory (if *inhib_nodes* is not *None*, it is the fraction of the nodes' edges that will become inhibitory, otherwise it is the fraction of all the edges in the graph).

   Returns t_list (`ndarray`) – List of the edges' types.

static **weights** (*graph=None*, *elist=None*, *wlist=None*, *distribution='constant'*, *parameters={}*, *noise_scale=None*)
   Compute the weights of the graph's edges. @todo: take elist into account

   Parameters

   - **graph** (class:~*nngt.Graph* or subclass) – Graph the nodes belong to.

   - **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined weights).

   - **wlist** (class:*numpy.array*, optional (default: None)) – List of the weights (for user defined weights).

   - **distribution** (class:*string*, optional (default: "constant")) – Type of distribution (choose among "constant", "uniform", "lognormal", "gaussian", "user_def", "lin_corr", "log_corr").

   - **parameters** (class:*dict*, optional (default: {})) – Dictionary containing the distribution parameters.

   - **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

   Returns new_weights (class:*scipy.sparse.lil_matrix*) – A sparse matrix containing *ONLY* the newly-computed weights.

nngt.core.**GraphObject**
   Graph object (reference to one of the main libraries' wrapper

   alias of _GtGraph

class nngt.core.**NeuralGroup** (*nodes=None*, *ntype=1*, *model=None*, *neuron_param=None*, *syn_model=None*, *syn_param=None*)
   Class defining groups of neurons.

   Variables

- **id_list** – list of int the ids of the neurons in this group.

- **neuron_type** – int the default is 1 for excitatory neurons; −1 is for interneurons

- **model** – string, optional (default: None) the name of the model to use when simulating the activity of this group

- **neuron_param** – dict, optional (default: {}) the parameters to use (if they differ from the model's defaults)

Note: By default, synapses are registered as "static_synapse" in NEST; because of this, only the neuron_model attribute is checked by the has_model function.

> **Warning:** Equality between NeuralGroup`s only compares the neuronal and synaptic ``model` and param attributes, i.e. groups differing only by their id_list will register as equal.

__eq__(*other*)

__init__(*nodes=None*, *ntype=1*, *model=None*, *neuron_param=None*, *syn_model=None*, *syn_param=None*)

has_model

id_list

nest_gids

neuron_model

properties

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** (*str or tuple*) – Names of the source groups (which contain the pre-synaptic neurons)
>
> - **target_groups** (*str or tuple*) – 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*)

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).

`nngt.generation.`**`distance_rule`**(*scale*, *rule='exp'*, *shape=None*, *neuron_density=1000.0*, *nodes=0*, *density=0.1*, *edges=-1*, *avg_deg=-1.0*, *unit='um'*, *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), "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*.

- **unit** (*string (default: 'um')*) – Unit for the length *scale* among 'um' (μm), 'mm', 'cm', 'dm', 'm'.

- **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 (N, 2) or 3D (N, 3) shaped 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*.

Note: *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='FD', 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** *'total'* 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.

Note: *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.

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

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

Generate a random graph with constant in- or out-degree. @todo: adapt it for undirected graphs!

> **Parameters**
>
> - **avg** (*float*) – The value of the average degree.
>
> - **std** (*float*) – The standard deviation of the Gaussian distribution.
>
> - **degree_type** (*str, optional (default: 'in')*) – The type of the fixed degree, among 'in', 'out' or 'total' @todo: Implement 'total' degree

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

Note: *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.

Parameters

- **in_exp** (*float*) – Absolute value of the in-degree exponent γ_i, such that $p(k_i) \propto k_i^{-\gamma_i}$

- **out_exp** (*float*) – Absolute value of the out-degree exponent γ_o, such that $p(k_o) \propto k_o^{-\gamma_o}$

- **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.

Returns graph_fs (`Graph`)

Note: 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 $\gamma = (\gamma_i + \gamma_o)/2$. 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*)

@todo make the algorithm.

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.)

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

`nngt.generation.`**`newman_watts`**(*coord_nb*, *proba_shortcut*, *nodes=0*, *weighted=True*, *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)

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

Plot module

Functions for plotting graphs and graph properties.

Content

`nngt.plot.`**`Animation2d`**

`nngt.plot.`**`AnimationNetwork`**

`nngt.plot.`**`draw_network`**(*network*, *nsize='total-degree'*, *ncolor='group'*, *nshape='o'*, *nbor-der_color='k'*, *nborder_width=0.5*, *esize=1.0*, *ecolor='k'*, *max_nsize=5.0*, *max_esize=2.0*, *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 float or string, optional (default: "total-degree")*) – Size of the nodes as a percentage of the canvas length. Otherwise, it can be a string that correlates the size to a node attribute among "in/out/total-degree", or "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, float or array, 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, str, or array of floats, optional (default: 0.5)*) – Width of the edges in percent of canvas length. Available string values are "betweenness" and "weight".
> >
> > - **ecolor** (*char, float or array, optional (default: "k")*) – Edge color.
> >
> > - **max_esize** (*float, optional (default: 5.)*) – If a custom property is entered as *esize*, this normalizes the edge width between 0. and *max_esize*.
> >
> > - **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).

`nngt.plot.`**`degree_distribution`**(*network*, *deg_type='total'*, *nodes=None*, *num_bins=50*, *use_weights=False*, *logx=False*, *logy=False*, *fignum=None*, *axis_num=None*, *colors=None*, *norm=False*, *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")
> >
> > - **nodes** (*list or numpy.array of ints, optional (default: all nodes)*) – Restrict the distribution to a set of nodes.
> >
> > - **num_bins** (*int, optional (default: 50):*) – Number of bins used to sample the distribution.
> >
> > - **use_weights** (*bool, optional (default: False)*) – use weighted degrees (do not take the sign into account : only the magnitude of the weights is considered).
> >
> > - **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`)) – Index of the figure on which the plot should be drawn (default creates a new figure).

- **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*, *nodes=None*, *logx=False*, *logy=False*, *num_nbins=None*, *num_ebins=None*, *fignum=None*, *axis_num=None*, *colors=None*, *norm=False*, *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).

>> - **nodes** (*list or numpy.array of ints, optional (default: all nodes)*) – Restrict the distribution to a set of nodes (taken into account only for the node attribute).

>> - **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.**node_attributes_distribution**(*network*, *attributes*, *nodes=None*, *num_bins=50*, *show=True*)

> Return node *attributes* for a set of *nodes*.

> **Parameters**

>> - **network** ([*Graph*](#)) – The graph where the *nodes* belong.

>> - **attributes** (*str or list*) – Attributes which should be returned, among: * "betweenness" * "clustering" * "in-degree", "out-degree", "total-degree" * "subgraph_centrality" * "b2" (requires NEST) * "firing_rate" (requires NEST)

>> - **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.

>> - **num_bins** (*int or list, optional (default: 50)*) – Number of bins to plot the distributions. If only one int is provided, it is used for all attributes, otherwize a list containing one int per attribute in *attributes* is required.

nngt.plot.**compare_population_attributes**(*network*, *attributes*, *nodes=None*, *reference_nodes=None*, *num_bins=50*, *reference_color='gray'*, *title=None*, *show=True*)

> Compare node *attributes* between two sets of nodes. Since number of nodes can vary, normalized distributions are used.

> **Parameters**

>> - **network** ([*Graph*](#)) – The graph where the *nodes* belong.

- **attributes** (*str or list*) – Attributes which should be returned, among: * "betweenness" * "clustering" * "in-degree", "out-degree", "total-degree" * "subgraph_centrality" * "b2" (requires NEST) * "firing_rate" (requires NEST)

- **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.

- **reference_nodes** (*list, optional (default: all nodes)*) – Reference nodes for which the attributes should be returned in order to compare with *nodes*.

- **num_bins** (*int or list, optional (default: 50)*) – Number of bins to plot the distributions. If only one int is provided, it is used for all attributes, otherwize a list containing one int per attribute in *attributes* is required.

nngt.plot.**correlation_to_attribute**(*network*, *reference_attribute*, *other_attributes*, *nodes=None*, *title=None*, *show=True*)

    For each node plot the value of *reference_attributes* against each of the *other_attributes* to check for correlations.

        Parameters

- **network** (*Graph*) – The graph where the *nodes* belong.

- **reference_attribute** (*str or array-like*) – Attribute which should serve as reference, among:

  - "betweenness"

  - "clustering"

  - "in-degree", "out-degree", "total-degree"

  - "subgraph_centrality"

  - "b2" (requires NEST)

  - "firing_rate" (requires NEST)

  - a custom array of values, in which case one entry per node in *nodes* is required.

- **other_attributes** (*str or list*)

- **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.

Analysis module

Content

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`.

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

> Assortativity of the graph.

> **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 quantifying the graph assortativity.*

nngt.analysis.**betweenness_distrib**(*graph*, *use_weights=True*, *nodes=None*, *num_nbins=None*, *num_ebins=None*, *log=False*)

> Betweenness distribution of a graph.

> **Parameters**

> - **graph** (`Graph` or subclass) – the graph to analyze.
>
> - **use_weights** (*bool, optional (default: True)*) – use weighted degrees (do not take the sign into account : all weights are positive).
>
> - **nodes** (*list or numpy.array of ints, optional (default: all nodes)*) – Restrict the distribution to a set of nodes (only impacts the node attribute).
>
> - **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.**closeness**(*graph*, *nodes=None*, *use_weights=False*)

> Return the closeness centrality for each node in *nodes*.

> **Parameters**

> - **graph** (`Graph` object) – Graph to analyze.
>
> - **nodes** (*array-like container with node ids, optional (default = all nodes)*) – Nodes for which the local clustering coefficient should be computed.
>
> - **use_weights** (*bool, optional (default: False)*) – Whether weighted closeness should be used.

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

> Global clustering coefficient of the graph, defined as

$$c = 3 \times \frac{\text{triangles}}{\text{connected triples}}$$

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

> Degree distribution of a graph.

> **Parameters**

> - **graph** (`Graph` or subclass) – the graph 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: False)*) – 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.**diameter**(*graph*)

Pseudo-diameter of the graph @todo: weighted diameter

nngt.analysis.**local_clustering**(*graph*, *nodes=None*)

Local clustering coefficient of the nodes, defined as

$$c_i = 3 \times \frac{\text{triangles}}{\text{connected triples}}$$

Parameters

- **graph** ([`Graph`](#) object) – Graph to analyze.

- **nodes** (*array-like container with node ids, optional (default = all nodes)*) – Nodes for which the local clustering coefficient should be computed.

nngt.analysis.**node_attributes**(*network*, *attributes*, *nodes=None*)

Return node *attributes* for a set of *nodes*.

Parameters

- **network** ([`Graph`](#)) – The graph where the *nodes* belong.

- **attributes** (*str or list*) – Attributes which should be returned, among: * "betweenness" * "clustering" * "in-degree", "out-degree", "total-degree" * "subgraph_centrality"

- **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.

Returns **values** (*array-like or dict*) – Returns the attributes, either as an array if only one attribute is required (*attributes* is a [`str`](#)) or as a [`dict`](#) of arrays.

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.`**`reciprocity`**(*graph*)

> Graph 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 graph.
>
> > **Returns** *a float quantifying the reciprocity.*

`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.`**`subgraph_centrality`**(*graph*, *weights=True*, *normalize='max_centrality'*)

> Subgraph centrality, accordign to [Estrada2005], for each node in the graph.
>
> [Estrada2005]: Ernesto Estrada and Juan A. Rodríguez-Velázquez, Subgraph centrality in complex networks, PHYSICAL REVIEW E 71, 056103 (2005), available on ArXiv.
>
> > **Parameters**
> >
> > - **graph** (`Graph` or subclass) – Network to analyze.
> >
> > - **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).
> >
> > - **normalize** (*str, optional (default: "max_centrality")*) – Whether the centrality should be normalized. Accepted normalizations are "max_eigenvalue" and "max_centrality"; the first rescales the adjacency matrix by the its largest eigenvalue before taking the exponential, the second sets the maximum centrality to one.
> >
> > **Returns** **centralities** (`numpy.ndarray`) – The subgraph centrality of each node.

Lib module

Content

`nngt.lib.`**`as_string`**(*graph*, *format='neighbour'*, *delimiter=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*, *return_info=False*)

> Full string representation of the graph.
>
> > **Parameters**
> >
> > - **graph** (`Graph` or subclass) – Graph to save.
> >
> > - **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 us-
> ing *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 meaning-full 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.

> **Returns** **str_graph** (*string*) – The full graph representation as a string.

`nngt.lib.`**`delta_distrib`**(*graph=None*, *elist=None*, *num=None*, *value=1.0*, ***kwargs*)
> Delta distribution for edge attributes.

> **Parameters**

- **graph** (`Graph` or subclass) – Graph for which an edge attribute will be generated.

- **elist** (*@todo*)

- **value** (*float, optional (default: 1.)*) – Value of the delta distribution.

- **Returns** (`numpy.ndarray`) – Attribute value for each edge in *graph*.

`nngt.lib.`**`gaussian_distrib`**(*graph*, *elist=None*, *num=None*, *avg=1.0*, *std=0.2*, ***kwargs*)
> Gaussian distribution for edge attributes.

> **Parameters**

- **graph** (`Graph` or subclass) – Graph for which an edge attribute will be generated.

- **elist** (*@todo*)

- **avg** (*float, optional (default: 0.)*) – Average of the Gaussian distribution.

- **std** (*float, optional (default: 1.5)*) – Standard deviation of the Gaussian distribution.

- **Returns** (`numpy.ndarray`) – Attribute value for each edge in *graph*.

exception `nngt.lib.`**`InvalidArgument`**
> Error raise when an argument is invalid.

`nngt.lib.`**`lin_correlated_distrib`**(*graph*, *elist=None*, *correl_attribute='betweenness'*, *noise_scale=None*, *lower=0.0*, *upper=2.0*, ***kwargs*)

`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.**log_correlated_distrib**(*graph*, *elist=None*, *correl_attribute='betweenness'*, *noise_scale=None*, *lower=0.0*, *upper=2.0*, ***kwargs*)

nngt.lib.**lognormal_distrib**(*graph*, *elist=None*, *num=None*, *position=1.0*, *scale=0.2*, ***kwargs*)

nngt.lib.**nonstring_container**(*obj*)
    Returns true for any iterable which is not a string or byte sequence.

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 meaning-full 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).

nngt.lib.**uniform_distrib**(*graph*, *elist=None*, *num=None*, *lower=0.0*, *upper=1.5*, ***kwargs*)
   Uniform distribution for edge attributes.

   > **Parameters**

   - **graph** (*Graph* or subclass) – Graph for which an edge attribute will be generated.

   - **elist** (*@todo*)

   - **lower** (*float, optional (default: 0.)*) – Min value of the uniform distribution.

   - **upper** (*float, optional (default: 1.5)*) – Max value of the uniform distribution.

   - **Returns** (`numpy.ndarray`) – Attribute value for each edge in *graph*.

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.

Rationale for the structure

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.

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).

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.

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)
```

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

Indices and tables

- genindex
- modindex
- search

n

Index