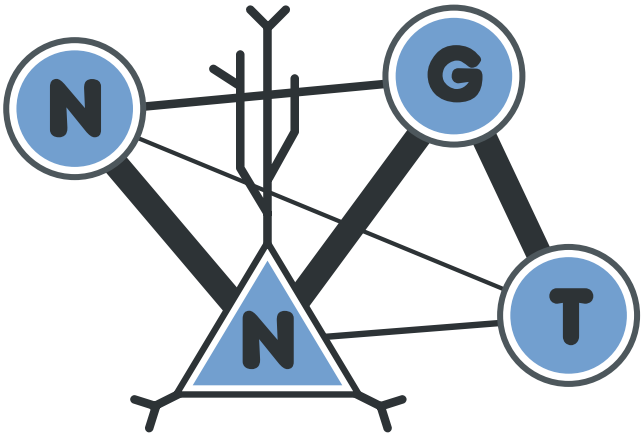


NNGT Documentation

Release 2.5.2

Tanguy Fardet

Nov 17, 2021

USER DOCUMENTATION

CHAPTER

ONE

OVERVIEW

The Neural Networks and Graphs' Topology (NNGT) module provides a unified interface to access, generate, and analyze networks via any of the well-known Python graph libraries: networkx, igraph, and graph-tool.

For people in neuroscience, the library also provides tools to grow and study detailed biological networks by interfacing efficient graph libraries with highly distributed activity simulators.

The library has two main targets:

- people looking for a unifying interface for these three graph libraries, allowing to run and share a single code on different platforms
- neuroscience people looking for an easy way to generate complex networks while keeping track of neuronal populations and their biological properties

1.1 Main classes

NNGT provides four main classes. The two first are aimed at the graph-theoretical community, the third and fourth are more for the neuroscience community. Additional details are provided on the *Main module (API)* page.

Graph provides a simple implementation to access and analyse topological graphs by wrapping any graph object from other graph libraries.

SpatialGraph a Graph embedded in space (nodes 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.

1.2 Generation of graphs

Structured graphs and connectivity: connectivity between the nodes can be chosen from various well-known graph models, specific groups and structures can be generated to simplify edge generation

Populations: populations of neurons can be used and be set to respect various constraints (for instance a given fraction of inhibitory neurons), they simplify network generation and make it highly efficient to interact with the NEST simulator

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

1.3 Interacting with NEST

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

CHAPTER

TWO

THE DOCS

2.1 Installation

2.1.1 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 (>= 1.11 required for full support)
- scipy

Though NNGT implements a default (limited) backend, installing one of the following libraries is highly recommended to do some proper network analysis:

- graph_tool (> 2.22)
- or igraph
- or networkx (>= 2.4)

Additionnal dependencies

- matplotlib (optional but will limit the functionalities if not present)
- shapely for complex spatial embedding
- peewee (> 3) for database features

Note: If they are not present on your computer, **`pip`** will directly try to install scipy and numpy. However, if you want advanced network analysis features, you will have to install the graph library yourself (only *networkx* can be installed directly using **`pip`**)

2.1.2 Simple install

Linux

Install the requirements (through **apt** on debian/ubuntu/mint, **pacman** and **trizen** on arch-based distributions, or **yum** on fedora/centos. Otherwise you can also install the latest versions via **pip**:

```
pip install --user numpy scipy matplotlib networkx
```

To install the last stable release, just use:

```
pip install --user nngt
```

Under most linux distributions, the simplest way to get the latest version of NNGT is to install to install both pip and git, then simply type into a terminal:

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

Mac

I recommend using Homebrew or 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 3.7* but you can use any python >= 3.5 (just replace all 37/3.7 by the desired version).

Homebrew

```
brew tap homebrew/core
brew tap brewsci/science

brew install gcc-8 cmake gsl autoconf automake libtool
brew install python
```

if you want nest, add

```
brew install nest --with-python
```

(note that setting `--with-python=3` might be necessary)

Macports

```
sudo port select gcc mp-gcc8 && sudo port install gsl +gcc8
sudo port install autoconf automake libtool
sudo port install python37 pip
sudo port select python python37
sudo port install py37-cython
sudo port select cython cython37
sudo port install py37-numpy py37-scipy py37-matplotlib py37-ipython
sudo port select ipython ipython-3.7
sudo port install py-graph-tool gtk3
```

Once the installation is done, you can just install:

```
export CC=gcc-8
export CXX=gcc-8
pip install --user nngt
```

Windows

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

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

Using the multithreaded algorithms

Install a compiler (the default **msvc** should already be present, otherwise you can install VisualStudio) before you make the installation.

In case of problems with **msvc**:

- install MinGW or MinGW-W64
- use it to install gcc with g++ support
- open a terminal, add the compiler to your *PATH* and set it as default: e.g.

```
set PATH=%PATH%;C:\MinGW\bin
set CC=C:\MinGW\bin\mingw32-gcc.exe
set CXX=C:\MinGW\bin\mingw32-g++.exe
```

- in that same terminal window, run `pip install --user nngt`

2.1.3 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
```

2.1.4 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 #
#-------------------------#

version = {version}

#-------------------------
## default backend       ------------------------------------------------------
#-------------------------

# library that will be used in the background to handle graph generation
# (choose among "graph-tool", "igraph", "networkx", or "nngt"). Note that only
# the 3 first options will allow full graph analysis features while only the
# last one allows for fully distributed memory on clusters.

backend = graph-tool


#----------------------
## Matplotlib backend ------------------------------------------------------
#----------------------

# Uncomment and choose among your available backends.
# See http://matplotlib.org/faq/usage_faq.html#what-is-a-backend for details

#mpl_backend = Qt5Agg

# use TeX rendering for axis labels
use_tex = False

# color library either matplotlib or seaborn
color_lib = matplotlib

# palette to use
palette_continuous = magma
palette_discrete = Set1


#----------------------------
## Settings for database    ------------------------------------------------
#----------------------------

use_database = False

# use a database (by default, will be stored in SQLite database)
db_to_file = False
db_folder  = ~/.nngt/database
db_name    = main
```

(continues on next page)

(continued from previous page)

```
# database url if you do not want to use a SQLite file
# example of real database url: db_url = mysql://user:password@host:port/my_db
# db_url = mysql:///nngt_db


#-----------------------------
## Settings for data logging ------------------------------------------------
#-----------------------------

# which messages are printed? (see logging module levels:
# https://docs.python.org/2/library/logging.html#levels)
# set to INFO or below to add the config messages on import
# set to WARNING or above to remove the messages on import
log_level = WARNING

# write log to file?
log_to_file = True
# if True, write to default folder '~/.nngt/log'
#log_folder = ~/.nngt/log


#-----------------------------
## Multithreaded/MPI algorithms ---------------------------------------------
#-----------------------------

# C++ algorithms using OpenMP are compiled and imported using Cython if True,
# otherwise regular numpy/scipy algorithms are used.
# Multithreaded algorithms should be prefered if available.

multithreading = True

# If using MPI, current MT or normal functions will be used except for the
# distance_rule algorithm, which will be overloaded by its MPI version.
# Note that the MPI version is not locally multithreaded.

mpi = False


#-----------------------------
## Third party libraries ----------------------------------------------------
#-----------------------------

# NNGT uses some 3rd party libraries for neuroscience and geospatial
# applications. Because these use compiled code, it can sometime lead to
# unexpected issues. You can prevent them from being loaded by setting their
# load option to False if you encounter such issues.

# try to load NEST on import

load_nest = True
```

(continues on next page)

(continued from previous page)

```
# try to load geospatial tools on import

load_gis = True
```

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`/`Qt5Agg`).

2.1.5 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

2.2 Intro & user manual

2.2.1 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. Most importantly, it provides a series of analysis functions that are guaranteed to provide the same results with all backends, enabling fully portable codes (see *Consistent tools for graph analysis*).

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

Documentation structure

For users that are in a hurry, you can go directly to the Tutorial section. For more specific and detailed examples, several topics are then detailed separately in the following pages:

Graph generation

This page gives example on how to generate increasingly complex network structures. The example files can be found at: docs/examples/simple_graphs.py, docs/examples/multi_groups_network.py, docs/examples/basic_nest_network.py, and docs/examples/nest_receptor_ports.py.

Content:

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 (*Attributes and distributions*). 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 `weights` 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 *`set_weights()`* function to set the weights of a `graph` explicitely by using:

```
graph.set_weights(elist=edges_to_weigh, distrib="distrib_of_choice", ...)
```

For more details on weights, other attributes, and available distributions, see *Properties of graph components*.

Examples

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

Simple generation

```
num_nodes  = 1000
avg_deg_er = 25
avg_deg_sf = 100

# random graphs
g1 = ng.erdos_renyi(nodes=num_nodes, avg_deg=avg_deg_er)

# the same graph but undirected
g2 = ng.erdos_renyi(nodes=num_nodes, avg_deg=avg_deg_er, directed=False)

# 2-step generation of a scale-free with Gaussian weight distribution
w = {
    "distribution": "gaussian",
    "avg": 60.,
    "std": 5.
}

g3 = nngt.Graph(num_nodes, weights=w)
ng.random_scale_free(2.2, 2.9, avg_deg=avg_deg_sf, from_graph=g3)

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

Networks composed of heterogeneous groups

```
'''
Make the population
'''

# two groups
g1 = nngt.Group(500)  # nodes 0 to 499
g2 = nngt.Group(500)  # nodes 500 to 999

# make structure
struct = nngt.Structure.from_groups((g1, g2), ("left", "right"))

# create network from this population
net = nngt.Graph(structure=struct)


'''
Connect the groups
'''

# inter-groups (Erdos-Renyi)
prop_er1 = {"density": 0.005}
ng.connect_groups(net, "left", "right", "erdos_renyi", **prop_er1)
```

(continues on next page)

(continued from previous page)

```
# intra-groups (Newman-Watts)
prop_nw = {
    "coord_nb": 20,
    "proba_shortcut": 0.1,
    "reciprocity_circular": 1.
}

ng.connect_groups(net, "left", "left", "newman_watts", **prop_nw)
ng.connect_groups(net, "right", "right", "newman_watts", **prop_nw)
```

Use with NEST

Generating a network with excitatory and inhibitory neurons:

```
'''
Build a network with two populations:
* excitatory (80%)
* inhibitory (20%)
'''
num_nodes = 1000

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

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

# exc -> exc (Newmann-Watts)
prop_nw = {
    "coord_nb": 10,
    "proba_shortcut": 0.1,
    "reciprocity_circular": 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)
```

Send the network to NEST:

```
if nngt.get_config('with_nest'):
    import nest
    import nngt.simulation as ns

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

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

    if nngt.get_config('with_plot'):
        ns.plot_activity(
            recorder, record, network=net, show=True, limits=(0,simtime))
```

You can check that connections from neurons that are marked as inhibitory are automatically assigned a negative sign in NEST:

```

    # sign of NNGT versus NEST inhibitory connections
    igroup = net.population["inhibitory"]

    # in NNGT
    iedges = net.get_edges(source_node=igroup.ids)
    w_nngt = set(net.get_weights(edges=iedges))

    # in NEST
    try:
        # nest 2
        iconn  = nest.GetConnections(
            source=list(net.population["inhibitory"].nest_gids),
```

Returns: `NNGT weights:  {1.0} versus NEST weights {-1.0}.`

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:
* 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

synapses = {
    (1, 1):   exc_syn,
    (1, -1):  exc_syn,
    (-1, 1):  inh_syn,
    (-1, -1): inh_syn,
}

pop = nngt.NeuralPop.exc_and_inhib(
    num_neurons, en_model=neuron_model, in_model=neuron_model,
    syn_spec=synapses)

# create the network and send it to NEST
w_prop = {"distribution": "gaussian", "avg": 0.2, "std": .05}
net = nngt.generation.gaussian_degree(
    avg_degree, std_degree, population=pop, weights=w_prop)

'''
Send to NEST and set excitation and recorders
'''
if nngt.get_config('with_nest'):
    import nest
    import nngt.simulation as ns

    nest.ResetKernel()

    gids = net.to_nest()

    # add noise to the excitatory neurons
    excs = list(pop["excitatory"].nest_gids)
```

(continues on next page)

(continued from previous page)

```
    inhs = list(pop["inhibitory"].nest_gids)
    ns.set_noise(excs, 10., 2.)
    ns.set_noise(inhs, 5., 1.)

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

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

    if nngt.get_config('with_plot'):
        ns.plot_activity(
            recorder, record, network=net, show=True, histogram=False,
            limits=(0, simtime))
```

Go to other tutorials:

- *Intro & user manual*
- *Properties of graph components*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Activity analysis*

Properties of graph components

This section details the different attributes and properties which can be associated to nodes/neurons and connections in graphs and networks.

Content:

- *Example*

Components of a graph

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, the second index 1, etc).

The source file for the examples given here can be found at doc/examples/attributes.py.

Node attributes

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 necessarily need to have attributes. 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. However, if you want to include additional information regarding the nodes, to account for specific differences in their properties, then node attributes are what you need. They are stored in *node_attributes*. Furthermore, to model more realistic neuronal networks, you might also want to define different groups and types of neurons, then connect them in specific ways. This specific feature will be provides by *NeuralGroup* objects.

Three types of node attributes

In the library, there is a difference between:

- standard attributes, which are stored in any type of *Graph* and can be created, modified, and accessed via the *new_node_attribute()*, *set_node_attribute()*, and *get_node_attributes()* functions.
- spatial properties (the positions of the neurons), which are stored in a specific `positions` `numpy.ndarray` and can be accessed using the *get_positions()* function,
- biological/group properties, which define assemblies of nodes sharing common properties, and are stored inside a *NeuralPop* object.

Standard attributes

Standard attributes can be any given label that might vary among the nodes in the network and will be attached to each node.

Users can define any attribute, through the *new_node_attribute()* function.

```
''' ---------------- #
# Generate a graph #
# ---------------- '''

num_nodes = 1000
avg_deg   = 25
```

(continues on next page)

(continued from previous page)

```
graph = ng.erdos_renyi(nodes=num_nodes, avg_deg=avg_deg)


''' ----------------- #
# Add node attributes #
# ----------------- '''

# Let's make a network of animals where nodes represent either cats or dogs.
# (no discrimination against cats or dogs was intended, no animals were harmed
# while writing or running this code)
animals  = ["cat" for _ in range(600)]  # 600 cats
animals += ["dog" for _ in range(400)]  # and 400 dogs
np.random.shuffle(animals)              # which we assign randomly to the nodes

graph.new_node_attribute("animal", value_type="string", values=animals)
```

Attributes can have different types:

- "double" for floating point numbers
- "int" for integers
- "string" for strings
- "object" for any other python object

Here we create a second node attribute of type "double":

```
# Nodes can have attributes of multiple types, let's add a size to our animals
catsizes = np.random.normal(50, 5, 600)   # cats around 50 cm
dogsizes = np.random.normal(80, 10, 400)  # dogs around 80 cm

# We first create the attribute without values (for "double", default to NaN)
graph.new_node_attribute("size", value_type="double")

# We now have to attributes: one containing strings, the other numbers (double)
print(graph.node_attributes)

# get the cats and set their sizes
cats = graph.get_nodes(attribute="animal", value="cat")
graph.set_node_attribute("size", values=catsizes, nodes=cats)

# We set 600 values so there are 400 NaNs left
assert np.sum(np.isnan(graph.get_node_attributes(name="size"))) == 400, \
    "There were not 400 NaNs as predicted."

# None of the NaN values belongs to a cat
assert not np.any(np.isnan(graph.get_node_attributes(cats, name="size"))), \
    "Got some cats with NaN size! :'("

# get the dogs and set their sizes
dogs = graph.get_nodes(attribute="animal", value="dog")
graph.set_node_attribute("size", values=dogsizes, nodes=dogs)
```

Biological/group properties

Note: All biological/group properties are stored in a *NeuralPop* object inside a *Network* instance; this attribute can be accessed through *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 *GroupProperty* class to create a population with groups that respect specific constraints.

For more details on biological properties, see *Groups, structures, and neuronal populations*.

Edge attributes

Like nodes, edges can also be attributed specific values to characterize them. However, where nodes are directly numbered and can be indexed and accessed easily, accessing edges is more complicated, especially since, usually, not all possible edges are present in a graph.

To easily access the desired edges, it is thus recommended to use the *get_edges()* function.

Edge attributes can then be created and recovered using similar functions as node attributes, namely *new_edge_attribute()*, *set_edge_attribute()*, and *get_edge_attributes()*.

Weights and delays

By default, graphs in NNGT are weighted: each edge is associated a "weight" value (this behavior can be changed by setting `weighted=False` upon creation).

Similarly, *Network* objects always have a "delay" associated to their connections.

Both attributes can either be set upon graph creation, through the `weights` and `delays` keyword arguments, or any any time using *set_weights()* and *set_delays()*.

Note: When working with NEST and using excitatory and inhibitory neurons via groups (see *Groups, structures, and neuronal populations*), the weight of all connections (including inhibitory connections) should be positive: the excitatory or inhibitory type of the synapses will be set automatically when the NEST network is created based on the type of the source neuron.

In general, it is also not a good idea to use negative weights directly since standard graph analysis methods cannot handle them. If you are not working with biologically realistic neurons and want to set some inhibitory connections that do not depend on a "neuronal type", use the *set_types()* method.

Let us see how the *get_edges()* function can be used to facilitate the creation of various weight patterns:

```
# Same as for node attributes, one can give attributes to the edges
# Let's give weights to the edges depending on how often the animals interact!
# cat's interact a lot among themselves, so we'll give them high weights
cat_edges = graph.get_edges(source_node=cats, target_node=cats)

# check that these are indeed only between cats
cat_set  = set(cats)
```

(continues on next page)

(continued from previous page)

```
node_set = set(np.unique(cat_edges))

assert cat_set == node_set, "Damned, something wrong happened to the cats!"

# uniform distribution of weights between 30 and 50
graph.set_weights(elist=cat_edges, distribution="uniform",
                  parameters={"lower": 30, "upper": 50})

# dogs have less occasions to interact except some which spend a lot of time
# together, so we use a lognormal distribution
dog_edges = graph.get_edges(source_node=dogs, target_node=dogs)
graph.set_weights(elist=dog_edges, distribution="lognormal",
                  parameters={"position": 2.2, "scale": 0.5})

# Cats do not like dogs, so we set their weights to -5
# Dogs like chasing cats but do not like them much either so we let the default
# value of 1
cd_edges = graph.get_edges(source_node=cats, target_node=dogs)
graph.set_weights(elist=cd_edges, distribution="constant",
                  parameters={"value": -5})

# Let's check the distribution (you should clearly see 4 separate shapes)
if nngt.get_config("with_plot"):
    nngt.plot.edge_attributes_distribution(graph, "weight")
```

Note that here, the weights were generated randomly from specific distributions; for more details on the available distributions and their parameters, see *Attributes and distributions*.

Custom edge attributes

Non-default edge attributes (besides "weights" or "delays") can also be created through smilar functions as node attributes:

```
class Human:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return "Human<{}>".format(self.name)

# let's create a class for humans and store it when two animals have interacted
# with the same human (the default will be an empty list if they did not)

# Alice interacted with all animals between 8 and 48
Alice   = Human("Alice")
animals = [i for i in range(8, 49)]
edges   = graph.get_edges(source_node=animals, target_node=animals)

graph.new_edge_attribute("common_interaction", value_type="object", val=[])
graph.set_edge_attribute("common_interaction", val=[Alice], edges=edges)

# Now suppose another human, Bob, interacted with all animals between 0 and 40
```

(continues on next page)

(continued from previous page)

```
Bob     = Human("Bob")
animals = [i for i in range(0, 41)]
edges2  = graph.get_edges(source_node=animals, target_node=animals)

# to update the values, we need to get them to add Bob to the list
ci = graph.get_edge_attributes(name="common_interaction", edges=edges2)

for interactions in ci:
    interactions.append(Bob)

graph.set_edge_attribute("common_interaction", values=ci, edges=edges2)

# now some of the initial `edges` should have had their attributes updated
new_ci = graph.get_edge_attributes(name="common_interaction", edges=edges)
print(np.sum([0 if len(interaction) < 2 else 1 for interaction in new_ci]),
      "interactions have been updated among the", len(edges), "from Alice.")
```

Attributes and distributions

Node and edge attributes can be generated based on the following distributions:

uniform

- a flat distribution with identical probability for all values,
- parameters: `"lower"` and `"upper"` values.

delta

- the Dirac delta "distribution", where a single value can be drawn,
- parameters: `"value"`.

Gaussian

- the normal distribution $P(x) = P_0 e^{(x-\mu)^2/(2\sigma^2)}$
- parameters: `"avg"` (μ) and `"std"` (σ).

lognormal

- $P(x) = P_0 e^{(\log(x)-\mu)^2/(2\sigma^2)}$
- parameters: `"position"` (μ) and `"scale"` (σ).

linearly correlated

- distribution name: `"lin_corr"`
- a distribution which evolves linearly between two values depending on the value of a reference variable
- parameters: `"correl_attribute"` (the reference variable, usually another attribute), `"lower"` and `"upper"`, the minimum and maximum values.

Example

Generating a graph with delays that are linearly correlated to the distance between nodes.

```
dmin = 1.
dmax = 8.

d = {
    "distribution": "lin_corr", "correl_attribute": "distance",
    "lower": dmin, "upper": dmax
}

g = nngt.generation.distance_rule(200., nodes=100, avg_deg=10, delays=d)
```

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Activity analysis*

Consistent tools for graph analysis

NNGT provides several functions for topological analysis that return consistent results for all backends (the results will always be the same regardless of which library is used under the hood). This section describes these functions and gives an overview of the currently supported methods.

Note: It is of course possible to use any function from the library on the *graph* attribute; however, not using one of the supported NNGT functions below will usually return results that are not consistent between libraries (and the code will obviously no longer be portable).

Supported functions

The following table details which functions are supported for directed and undirected networks, and whether they also work with weighted edges.

The test file where these functions are checked can be found here: testing/library_compatibility.py.

For each type of graph, the table tells which libraries are supported for the given function (graph-tool is *gt*, networkx is *nx* and igraph is *ig*). Custom implementation of a function is denoted by *nngt*, meaning that the function can be used even if no graph library is installed. A library marked between parentheses denotes partial support and additional explanation is usually given in the footnotes. A cross means that no consistent implementation is currently provided and the function will raise an error if one tries to use it on such graphs. Methods that are not defined for weighted or directed graphs are marked by NA.

Method	Unweighted undirected	Unweighted directed	Weighted undirected	Weighted directed
all_shortest_paths()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
average_path_length()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
assortativity()[1]	gt, nx, ig	gt, nx, ig	gt, ig	gt, ig
betweenness()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
betweenness_distrib()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
closeness()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
connected_components()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
degree_distrib()	gt, nx, ig, nngt	gt, nx, ig, nngt	gt, nx, ig, nngt	gt, nx, ig, nngt
diameter()[2]	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
global_clustering()	gt, nx, ig, nngt	nngt	nngt	nngt
local_clustering()[3]	gt, nx, ig, nngt	nngt	nngt	nngt
reciprocity()	gt, nx, ig, nngt	gt, nx, ig, nngt	NA	NA
shortest_distance()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
shortest_path()	gt, nx, ig	gt, nx, ig	gt, nx, ig	gt, nx, ig
spectral_radius()	nngt	nngt	nngt	nngt
subgraph_centrality()	nngt	nngt	nngt	nngt
transitivity()[4]	gt, nx, ig, nngt	nngt	nngt	nngt

Clustering in weighted and directed networks

For directed clustering, NNGT provides the total clustering proposed in [Fagiolo2007]

$$C_i^d = \frac{\frac{1}{2}(A + A^T)^3}{d_i^{tot}(d_i^{tot} - 1) - d_i^{\leftrightarrow}}$$

with $d_i^{\leftrightarrow} = A_{ii}^2$ is the reciprocal degree.

For undirected weighted clustering, NNGT provides the definition proposed in [Barrat2004], [Onnela2005], [Zhang2005] as well as a new continuous definition [Fardet2021].

$$C_{B,i}^u = \frac{(WA^2)_{ii}}{s_i(d_i - 1)}$$

$$C_{O,i}^u = \frac{(W^{[\frac{1}{3}]})_{ii}^3}{d_i(d_i - 1)}$$

$$C_{Z,i}^u = \frac{(W^3)_{ii}}{\sum_{j \neq k} w_{ij} w_{ik}}$$

$$C_{c,i}^u = \frac{\left(W^{[\frac{2}{3}]}\right)_{ii}^3}{\left(s_i^{[\frac{1}{2}]}\right)^2 - s_i}$$

with $s^{[\frac{1}{2}]}$ the generalized strength associated to the matrix $W^{[\frac{1}{2}]} = \{\sqrt{w_{ij}}\}$.

[1] networkx could be used via a workaround but an issue has been raised to find out how to best deal with this.
[2] the implementation of the diameter for graph-tool is approximmate so results may occasionaly be inexact with this backend.
[3] for directed and weighted networks, definitions and implementations differ between graph libraries, so generic implementations are provided in NNGT. See "*Clustering in weighted and directed networks*" for details.
[4] identical to `global_clustering`.

For directed weighted clustering, the generalization of Barrat from [Clemente2018] is provided, as well as a generalization of Onnela, Zhang–Horvath, and of the continuous clustering [Fardet2020], for all four directed modes (middleman, cycle, fan-in, and fan-out), as well as their sum, the total clustering:

$$C^d_{B,i} = \frac{\frac{1}{2}((W+W^T)(A+A^T)^2)_{ii}}{s_i(d^{tot}_i - 1) - s^{\leftrightarrow}_{c,i}}$$

with s the total strength and $s^{\leftrightarrow}_{c,i} = \frac{1}{2}(WA + AW)_{ii}$ the arithmetic reciprocal strength,

$$C^d_{O,i} = \frac{\frac{1}{2}(W^{[\frac{1}{3}]} + (W^{[\frac{1}{3}]})^T)^3_{ii}}{d^{tot}_i(d^{tot}_i - 1) - d^{\leftrightarrow}_i}$$

$$C^d_{Z,i} = \frac{(W+W^T)^3_{ii}}{\sum_{j\neq k}(w_{ij}+w_{ji})(w_{ik}+w_{ki})}$$

$$C^d_{c,i} = \frac{\frac{1}{2}\left(W^{[\frac{2}{3}]} + W^{[\frac{2}{3}],T}\right)^3_{ii}}{\left(s_i^{[\frac{1}{2}]}\right)^2 - 2s^{\leftrightarrow}_i - s_i}$$

with $s^{[\frac{1}{2}]}$ the total generalized strength and $s^{\leftrightarrow}_i = \left(W^{[\frac{1}{2}]}\right)^2$ the geometric reciprocal strength.

Global clusterings are defined as the sum of all numerators divided by the sum of all denominators for all definitions.

References

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Activity analysis*

Parallelism

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.

NNGT provides two types of parallelism:

- shared-memory parallelism, using OpenMP, which can be set using *nngt.set_config()* (`"multithreading", True`) or, setting the number of threads, with `nngt.set_config("omp", 8)` to use 8 threads.
- distributed-memory parallelism using MPI, which is set through `nngt.set_config("mpi", True)`. In that case, the python script must be run as `mpirun -n 8 python name_of_the_script.py` to be run in parallel.

These two ways of running code in parallel differ widely, both regarding the situations in which they can be useful, and in the way the user should interact with the resulting graph.

The easiest tool, because it does not significantly differ from the single-thread case on the user side, is OpenMP, which is why we will describe it first. Using MPI is a lot different and will require the user to adapt the code to use it and will depend on the backend used.

Parallelism and random numbers

When using parallel algorithms, additional care is necessary when dealing with random number generation. Here again, the situation differs between the OpenMP and MPI cases.

Warning: Never use the standard *random* module, only use *numpy.random*!

When using OpenMP, the parallel algorithms will use the random seeds defined by the user through `nngt.set_config("seeds", list_of_seeds)`. One seed per thread is necessary. These seeds are not used on the python level, so they are independent from whatever random generation could happen using *numpy* (e.g. to set node positions in space, or to generate attributes). To make a simulation fully reproducible, the user must set both the random seeds and the python level random number generators through the master seed. For instance, with 4 threads:

```
master_seed = 0
nngt.set_config({"msd": master_seed, "seeds": [1, 2, 3, 4]})
```

Note: If the seeds are not provided, then they are generated automatically, from the master seed for the first call to a graph-generation method (using $\{MSD + 1 + i\}_{i \in 0..N}$, with N the number of threads), then using a random number generated through numpy. This means that all previous calls to `numpy.random` will affect the random seeds used for the second or later calls to graph-generation methods unless new seeds are manually set by the user befor each new call (this does not mean that the code will not be reproducible, only that changes in the random calls in the code that occur before calls to graph-generation methods would affect the random structure of the generated graphs).

Warning: This is also how you should initialize random numbers when using MPI!

This may surprise experienced MPI users, but NNGT is implemented in such a way that shared properties are generated on all threads through the initial python master seed, then generation algorithms save the current common state, then re-initialize the RNGs for parallel generation, and finally restore the previous, common random state once the parallel generation is done. Of course the parallel initialization differs every time, but it is changed in a reproducible way through the master seed.

Using OpenMP (shared-memory parallelism)

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.

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.

Using MPI (distributed-memory parallelism)

Note: MPI algorithms are currently restricted to *gaussian_degree()* and *distance_rule()* only.

Handling MPI can be significantly more difficult than using OpenMP because it differs more strongly from the "standard" single-thread case.

NNGT provides two different ways of using MPI:

- When using one of the three graph libraries (graph-tool, igraph, or networkx), the connections are generated in parallel, but the final object is stored only on the master process. This means that in this case, the memory load will weigh only on this process, leading to a strong load imbalance. This feature is aimed at people who would require parallelism to speed up their graph generation but, for some reason, cannot use the OpenMP parallelism.

- For "real" memory distribution, e.g. for people working on clusters, who require a balanced memory-load, NNGT provides a custom backend, that can be set using `nngt.set_config('backend', 'nngt')`. In this case, each process stores only a fraction of all the edges. However, nodes and graph properties are fully available on all processes.

Warning: When using MPI with graph-tool, igraph, or networkx, all operations on the graph that has been generated must be limited to the root process. To that end, NNGT provides the *on_master_process()* function that returns *True* only on the root MPI process. Using the 'nngt' backend, the *edge_nb()* method, as well as all other edge-related methods will return information on the local edges only!

Fully distributed setup

The python file should include (before any graph generation):

```
import nngt

msd   = 0             # choose a master seed
seeds = [1, 2, 3, 4]  # choose initial seeds, one per MPI process

nngt.set_config({
    "mpi": True,
    "backend": "nngt",
    "msd": msd,
    "seeds": seeds,
})
```

The file should then be executed using:

```
>>> mpirun -n 4 python name_of_the_script.py
```

Note: Graph saving is available in parallel in the fully distributed setup through the *to_file()* and *save_to_file()* functions as in any other configuration.

Parallelized generation algorithms

Generation of some *directed* graphs are available with parallel implementations (see table below). No undirected graph generation mechanisms are currently implemented.

Function	OMP	MPI
all_to_all()	no	no
circular()	no	no
distance_rule()	yes	yes
erdos_renyi()	no	no
fixed_degree()	yes	yes
from_degree_list()	yes	yes
gaussian_degree()	yes	yes
newman_watts()	no	no
random_scale_free()	no	no

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Activity analysis*
- *Properties of graph components*

Groups, structures, and neuronal populations

One notable feature of NNGT is to enable users to group nodes (neurons) into groups sharing common properties in order to facilitate the generation of a network, the analysis of its properties, or complex simulations with NEST.

The complete example file containing the code discussed here, as well as additional information on how to access `NeuralGroup` and `NeuralPop` properties can be found there: docs/examples/introduction_to_groups.py.

Contents

Neuronal groups

Neuronal groups are entities containing neurons which share common properties. Inside a population, a single neuron belongs to a single `NeuralGroup` object. Conversely the union of all groups contains all neurons in the network once and only once.

When creating a group, it is therefore important to make sure that it forms a coherent set of neurons, as this will make network handling easier.

For more versatile grouping, where neurons can belong to multiple ensembles, see the section about meta-groups below: *Complex populations and metagroups*.

Creating simple groups

Groups can be created easily through calls to *Group* or *NeuralGroup*.

```
>>> group  = nngt.Group()
>>> ngroup = nngt.NeuralGroup()
```

create empty groups (nothing very interesting).

Minimally, any useful group requires at least neuron ids and, for a neuronal group, a type (excitatory or inhibitory) to be useful.

To create a useful group, one can therefore either just tell how many nodes/neurons it should contain:

```
group1 = Group(500)  # a group with 500 nodes
```

or directly pass it a list of ids (to avoid typing `nngt.` all the time, we do `from nngt import Group, NeuralGroup` at the beginning)

```
group2 = NeuralGroup(range(10, 20))  # neurons with ids from 10 to 19
```

Note that if you set ids directly you will be responsible for their consistency.

Creating a structured graph

To create a structured graph, the groups are gathered into a `Structure` which can then be used to create a graph and connect the nodes.

```
room1 = nngt.Group(25)
room2 = nngt.Group(50)
room3 = nngt.Group(40)
room4 = nngt.Group(35)

names = ["R1", "R2", "R3", "R4"]

struct = nngt.Structure.from_groups((room1, room2, room3, room4), names)

g = nngt.Graph(structure=struct)

for room in struct:
    nngt.generation.connect_groups(g, room, room, "all_to_all")

nngt.generation.connect_groups(g, (room1, room2), struct, "erdos_renyi",
                               avg_deg=10, ignore_invalid=True)

nngt.generation.connect_groups(g, room3, room1, "erdos_renyi", avg_deg=20)

nngt.generation.connect_groups(g, room4, room3, "erdos_renyi", avg_deg=10)
```

More realistic neuronal groups

When designing neuronal networks, one usually cares about their type (excitatory or inhibitory for instance), their properties, etc.

By default, neural groups are created excitatory and the following lines are therefore equivalent:

```
exc   = NeuralGroup(800, neuron_type=1)   # excitatory group
exc2  = NeuralGroup(800, neuron_type=1)   # also excitatory
```

To create an inhibitory group, the neural type must be set to -1:

```
inhib = NeuralGroup(200, neuron_type=-1)  # inhibitory group
```

Moving towards really realistic groups to run simulation on NEST afterwards, the last step is to associate a neuronal model and set the properties of these neurons (and optionally give them names):

```
pyr = NeuralGroup(800, neuron_type=1, neuron_model="iaf_psc_alpha",
                  neuron_param={"tau_m": 50.}, name="pyramidal_cells")

fsi = NeuralGroup(200, neuron_type=-1, neuron_model="iaf_psc_alpha",
                  neuron_param={"tau_m": 20.},
                  name="fast_spiking_interneurons")
```

Populations

Populations are ensembles of neuronal groups which describe all neurons in a corresponding network. They are usually created before the network and then used to generate connections, but the can also be generated after the network creation, then associated to it.

Simple populations

To create a population, you can start from scratch by creating an empty population, then adding groups to it:

```
# making populations from scratch
pop = nngt.NeuralPop(with_models=False)              # empty population
pop.create_group(200, "first_group")                 # create excitatory group
pop.create_group(5, "second_group", neuron_type=-1)  # create inhibitory group
```

NNGT also provides a two default routine to create simple populations:

- *uniform()*, to generate a single population where all neurons belong to the same group,
- *exc_and_inhib()*, to generate a mixed excitatory and inhibitory population.

As before, we do `from nngt import NeuralPop` to avoid typing `nngt.` all the time.

To create such populations, just use:

```
# the two default populations
unif_pop = NeuralPop.uniform(1000)                    # only excitatory
ei_pop   = NeuralPop.exc_and_inhib(1000, iratio=0.25)  # 25% inhibitory
```

Eventually, a population can be created from exiting groups using *from_groups()*:

```
ei_pop2 = NeuralPop.from_groups([exc, exc2, inhib], ["e1", "e2", "i"],
                                with_models=False)
```

Note that, here, we pass `with_models=False` to the population because these groups were created without the information necessary to create a network in NEST (a valid neuron model).

NEST-enabled populations

To create a NEST-enabled population, one can use one of the standard classmethods (*uniform()* and *exc_and_inhib()*) and pass it valid parameters for the neuronal models (optionally also a synaptic model and neuronal/synaptic parameters).

Otherwise, one can build the population from groups that already contain these properties, e.g. the previous `pyr` and `fsi` groups:

```
# optional synaptic properties
syn_spec = {
    'default': {"synaptic_model": "tsodyks2_synapse"},  # default connections
    ("pyramidal_cells", "pyramidal_cells"): {"U": 0.6}  # change a parameter
}

nest_pop = NeuralPop.from_groups([pyr, fsi], syn_spec=syn_spec)
```

Warning: *syn_spec* can contain any synaptic model and parameters associated to the NEST model; however, neither the synaptic weight nor the synaptic delay can be set there. For details on how to set synaptic weight and delays between groups, see *connect_groups()*.

To see how to use a population to create a *Network* and send it to NEST, see *Use with NEST*.

Complex populations and metagroups

When building complex neuronal networks, it may be useful to have neurons belong to multiple groups at the same time. Because standard groups can contain a neuron only once, meta-groups were introduced to provide this additional functionality.

Contrary to normal groups, a neuron can belong to any number of metagroups, which allow to make various sub- or super-groups. For instance, when modeling a part of cortex, neurons will belong to a layer, and to a given cell class whithin that layer. In that case, you may want to create specific groups for cell classes, like `L3Py`, `L5Py`, `L3I`, `L5I` for layer 4 and 5 pyramidal cells as well as interneurons, but you can then also group neurons in a same layer together, and same with pyramidal neurons or interneurons.

First create the normal groups:

```
nmod = "iaf_psc_exp"

idsL2gc = range(100)
idsL3py, idsL3i = range(100, 200), range(200, 300)
idsL4gc = range(300, 400)
idsL5py, idsL5i = range(400, 500), range(500, 600)
idsL6 = range(600, 700)
```

(continues on next page)

(continued from previous page)

```
L2GC = NeuralGroup(idsL2gc, neuron_model=nmod, name="L2GC", neuron_type=1)
L3Py = NeuralGroup(idsL3py, neuron_model=nmod, name="L3Py", neuron_type=1)
L3I  = NeuralGroup(idsL3i,  neuron_model=nmod, name="L3I",  neuron_type=-1)
L4GC = NeuralGroup(idsL4gc, neuron_model=nmod, name="L4GC", neuron_type=1)
L5Py = NeuralGroup(idsL5py, neuron_model=nmod, name="L5Py", neuron_type=1)
L5I  = NeuralGroup(idsL5i,  neuron_model=nmod, name="L5I",  neuron_type=-1)
L6c  = NeuralGroup(idsL6,   neuron_model=nmod, name="L6c",  neuron_type=1)
```

Then make the metagroups for the layers:

```
L2 = MetaGroup(idsL2gc, name="L2")
L3 = MetaNeuralGroup(L3Py.ids + L3I.ids, name="L3")
L4 = MetaGroup(idsL4gc, name="L4")
L5 = MetaNeuralGroup(L5Py.ids + L5I.ids, name="L5")
L6 = MetaGroup(idsL6, name="L6")
```

Note that I used `MetaNeuralGroup` for layers 3 and 5 because it enables to differenciate inhibitory and excitatory neurons using *`inhibitory`* and *`excitatory`*. Otherwise normal *`MetaGroup`* are equivalent and sufficient.

Create the population:

```
pop_column = NeuralPop.from_groups(
    [L2GC, L3Py, L3I, L4GC, L5Py, L5I, L6c], meta_groups=[L2, L3, L4, L5, L6])
```

Then add additional metagroups for cell types:

```
pyr = MetaGroup(L3Py.ids + L5Py.ids, name="pyramidal")
pop_column.add_meta_group(pyr)  # add from existing meta-group

pop_column.create_meta_group(L3I.ids + L5I.ids, "interneurons")  # single line

pop_column.create_meta_group(L2GC.ids + L4GC.ids, "granule")
```

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Parallelism*
- *Interacting with the NEST simulator*
- *Activity analysis*
- *Properties of graph components*

Interacting with the NEST simulator

This section details how to create detailed neuronal networks, then run simulations on them using the NEST simulator.

Readers are supposed to have a good grasp of the way NEST handles neurons and models, and how to create and setup NEST nodes. If this is not the case, please see the NEST user doc and the PyNEST tutorials first.

NNGT tools should work for NEST version 2 or 3; they can be separated into

- the structural tools (*Network*, *NeuralPop* ...) that are used to prepare the neuronal network and setup its properties and connectivity; these tools should be used **before**
- the *make_nest_network()* and the associated, *to_nest()* functions that are used to send the previously prepared network to NEST;
- then, **after** using one of the previous functions, all the other functions contained in the *nngt.simulation* module can be used to add stimulations to the neurons or monitor them.

Note: Calls to `nest.ResetKernel` will also reset all networks and populations, which means that after such a call, populations, parameters, etc, can again be changed until the next invocation of *make_nest_network()* or *to_nest()*.

Example files associated to the interactions between NEST and NNGT can be found here: docs/examples/nest_network.py / docs/examples/nest_receptor_ports.py.

Content:

Creating detailed neuronal networks

`NeuralPop` and `NeuralGroup`

These two classes are the basic blocks to design neuronal networks: a *NeuralGroup* is a set of neurons sharing common properties while the *NeuralPop* is the main container that represents the whole network as an ensemble of groups.

Depending on your perspective, you can either create the groups first, then build the population from them, or create the population first, then split it into various groups.

For more details on groups and populations, see *Groups, structures, and neuronal populations*.

Neuronal groups before the population

Neural groups can be created as follow:

```
# 100 inhibitory neurons
basic_group = nngt.NeuralGroup(100, neuron_type=-1)
# 10 excitatory (default) aeif neurons
```

(continues on next page)

(continued from previous page)

```
aeif_group  = nngt.NeuralGroup(10, neuron_model="aeif_psc_alpha")
# an unspecified number of aeif neurons with specific parameters
p = {"E_L": -58., "V_th": -54.}
aeif_g2 = nngt.NeuralGroup(neuron_model="aeif_psc_alpha", neuron_param=p)
```

In the case where the number of neurons is specified upon creation, NNGT can check that the number of neurons matches in the network and the associated population and raise a warning if they don't. However, it is just a security check and it does not prevent the network for being created if the numbers don't match.

Once the groups are created, you can simply generate the population using

```
pop = nngt.NeuralPop.from_groups([basic_group, aeif_group], ["b", "a"])
```

This created a population separated into "a" and "b" from the previously created groups.

Population before the groups

A population with excitatory and inhibitory neurons

```
pop = nngt.NeuralPop(1000)
pop.create_group(800, "first")
pop.create_group(200, "second", neuron_type=-1)
```

or, more compact

```
pop = nngt.NeuralPop.exc_and_inhib(1000, iratio=0.2)
```

The `Network` class

Besides connectivity, the main interest of the *NeuralGroup* is that you can pass it the biological properties that the neurons belonging to this group will share.

Since we are using NEST, these properties are:

- the model's name
- its non-default properties
- the synapses that the neurons have and their properties
- the type of the neurons (1 for excitatory or -1 for inhibitory)

```
''' Create groups with different parameters '''
# adaptive spiking neurons
base_params = {
    'E_L': -60., 'V_th': -58., 'b': 20., 'tau_w': 100.,
    'V_reset': -65., 't_ref': 2., 'g_L': 10., 'C_m': 250.
}
# oscillators
params1, params2 = base_params.copy(), base_params.copy()
params1.update(
    {'E_L': -65., 'b': 40., 'I_e': 200., 'tau_w': 400., "V_th": -57.})
# bursters
params2.update({'b': 25., 'V_reset': -55., 'tau_w': 300.})
```

(continues on next page)

(continued from previous page)

```
oscill = nngt.NeuralGroup(
    nodes=400, neuron_model='aeif_psc_alpha', neuron_type=1,
    neuron_param=params1)

burst = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', neuron_type=1,
    neuron_param=params2)

adapt = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', neuron_type=1,
    neuron_param=base_params)

synapses = {
    'default': {model: 'tsodyks2_synapse'},
    ('oscillators', 'bursters'): {model: 'tsodyks2_synapse', 'U': 0.6},
    ('oscillators', 'oscillators'): {model: 'tsodyks2_synapse', 'U': 0.7},
    ('oscillators', 'adaptive'): {model: 'tsodyks2_synapse', 'U': 0.5}
}

'''
Create the population that will represent the neuronal
network from these groups
'''
pop = nngt.NeuralPop.from_groups(
    [oscill, burst, adapt],
    names=['oscillators', 'bursters', 'adaptive'], syn_spec=synapses)

'''
Create the network from this population,
using a Gaussian in-degree
'''
net = ng.gaussian_degree(
    100., 15., population=pop, weights=155., delays=5.)
```

Once this network is created, it can simply be sent to nest through the command: `gids = net.to_nest()`, and the NEST gids are returned.

In order to access the gids from each group, you can do:

```
oscill_gids = net.nest_gids[oscill.ids]
```

or directly:

```
oscill_gids = oscill.nest_gids
```

As shown in "*Use with NEST*", synaptic strength from inhibitory neurons in NNGT are positive (for compatibility with graph analysis tools) but they are automatically converted to negative values when the network is created in NEST.

Changing the parameters of neurons

Before sending the network to NEST

Once the *NeuralPop* has been created, you can change the parameters of the neuron groups **before you send the network to NEST**.

To do this, you can use the `set_param()` function, to which you pass the parameter dict and the name of the *NeuralGroup* you want to modify.

If you are dealing directly with *NeuralGroup* objects, you can access and modify their `neuron_param` attribute as long as the network has not been sent to nest. Once sent, these parameters become unsettable and any wourkaround to circumvent this will not change the values inside NEST anyway.

After sending the network to NEST, randomizing

Once the network has been sent to NEST, neuronal parameters can still be changed, but only for randomization purposes. It is possible to randomize the neuronal parameters through the *randomize_neural_states()* function. This sets the parameters using a specified distribution and stores their values inside the network nodes' attributes.

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Activity analysis*
- *Properties of graph components*

Activity analysis

- *Principle*
- *Sorted rasters*
- *Activity properties*

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:

```
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

```
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`:

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

Additional information:

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.ActivityRecord(spike_data, ...)	Class to record the properties of the simulated activity.
nngt.simulation.activity_types(...[, ...])	Analyze the spiking pattern of a neural network.
nngt.simulation.analyze_raster([raster, ...])	Return the activity types for a given raster.
nngt.simulation.get_nest_adjacency([...])	Get the adjacency matrix describing a NEST network.
nngt.simulation.get_recording(network, record)	Return the evolution of some recorded values for each neuron.
nngt.simulation.make_nest_network(network[, ...])	Create a new network which will be filled with neurons and connector objects to reproduce the topology from the initial network.
nngt.simulation.monitor_groups(group_names, ...)	Monitoring the activity of nodes in the network.
nngt.simulation.monitor_nodes(gids[, ...])	Monitoring the activity of nodes in the network.
nngt.simulation.plot_activity([...])	Plot the monitored activity.
nngt.simulation.randomize_neural_states(...)	Randomize the neural states according to the instructions.
nngt.simulation.raster_plot(times, senders)	Plotting routine that constructs a raster plot along with an optional histogram.
nngt.simulation.reproducible_weights(...[, ...])	Find the values of the connection weights that will give PSP responses of *min_weight* and *max_weight* in mV.
nngt.simulation.save_spikes(filename[, ...])	Plot the monitored activity.
nngt.simulation.set_minis(network, ...[, ...])	Mimick spontaneous release of neurotransmitters, called miniature PSCs or "minis" that can occur at excitatory (mEPSCs) or inhibitory (mIPSCs) synapses.
nngt.simulation.set_noise(gids, mean, std)	Submit neurons to a current white noise.
nngt.simulation.set_poisson_input(gids, rate)	Submit neurons to a Poissonian rate of spikes.
nngt.simulation.set_step_currents(gids, ...)	Set step-current excitations

Details

class nngt.simulation.**ActivityRecord**(*spike_data*, *phases*, *properties*, *parameters=None*)

Class to record the properties of the simulated activity.

Initialize the instance using *spike_data* (store proxy to an optional *network*) and compute the properties of provided data.

Parameters

- **spike_data** (*2D array*) – Array of shape (num_spikes, 2), containing the senders on the 1st row and the times on the 2nd row.
- **phases** (*dict*) – Limits of the different phases in the simulated period.
- **properties** (*dict*) – Values of the different properties of the activity (e.g. "firing_rate", "IBI"...).
- **parameters** (*dict, optional (default: None)*) – Parameters used to compute the phases.

Note: The firing rate is computed as num_spikes / total simulation time, the period is the sum of an IBI and a bursting period.

nngt.simulation.**activity_types**(*spike_recorder*, *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_recorder** (*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 region 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 participate for a local burst to be validated (i.e. if the interspike is smaller that the limit BUT the number of participating neurons is too small, the phase will be considered as 'mixed').
- **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 (*bursting', `quiescent', `mixed', and `localized*) as lists. E.g: `phases["bursting"]` could give `[[123.5, 334.2], [857.1,1000.6]]`.

nngt.simulation.**analyze_raster**(*raster=None*, *limits=None*, *network=None*, *phase_coeff=(0.5, 10.0)*, *mbis=0.5*, *mfb=0.2*, *mflb=0.05*, *skip_bursts=0*, *skip_ms=0.0*, *simplify=False*, *fignums=[]*, *show=False*)

Return the activity types for a given raster.

Parameters

- **raster** (*array-like (N, 2) or str*) – Either an array containing the ids of the spiking neurons on the first column, then the corresponding times on the second column, or the path to a NEST .gdf recording.
- **limits** (*tuple of floats*) – Time limits of the simulation regrion which should be studied (in ms).
- **network** (*Network*, optional (default: None)) – Network on which the recorded activity was simulated.
- **phase_coeff** (*tuple of floats, optional (default: (0.2, 5.))*) – A phase is considered 'bursting' when the interspike between all spikes that compose it is smaller than `phase_coeff[0] / avg_rate` (where `avg_rate` is the average firing rate), 'quiescent' when it is greater that `phase_coeff[1] / avg_rate`, 'mixed' otherwise.
- **mbis** (*float, optional (default: 0.5)*) – Maximum interspike interval allowed for two spikes to be considered in the same burst (in ms).
- **mfb** (*float, optional (default: 0.2)*) – Minimal fraction of the neurons that should participate for a burst to be validated (i.e. if the interspike is smaller that the limit BUT the number of participating neurons is too small, the phase will be considered as 'localized').
- **mflb** (*float, optional (default: 0.05)*) – Minimal fraction of the neurons that should participate for a local burst to be validated (i.e. if the interspike is smaller that the limit BUT the number of participating neurons is too small, the phase will be considered as 'mixed').
- **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.
- **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 **activity** (*ActivityRecord*) – Object containing the phases and the properties of the activity from these phases.

nngt.simulation.**get_nest_adjacency**(*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.**get_recording**(*network*, *record*, *recorder=None*, *nodes=None*)

Return the evolution of some recorded values for each neuron.

Parameters

- **network** (*nngt.Network*) – Network for which the activity was simulated.
- **record** (*str or list*) – Name of the record(s) to obtain.
- **recorder** (*tuple of ints, optional (default: all multimeters)*) – GID of the spike recorder objects recording the network activity.

- **nodes** (*array-like, optional (default: all nodes)*) – NNGT ids of the nodes for which the recording should be returned.

Returns **values** (*dict of dict of arrays*) – Dictionary containing, for each *record*, an M array with the recorded values for n-th neuron is stored under entry *n* (integer). A *times* entry is also added; it should be the same size for all records, otherwise an error will be raised.

Examples

After the creation of a *Network* called `net`, use the following code:

```
import nest

rec, _ = monitor_nodes(
    net.nest_gids, "multimeter", {"record_from": ["V_m"]}, net)
nest.Simulate(100.)
recording = nngt.simulation.get_recording(net, "V_m")

# access the membrane potential of first neuron + the times
V_m   = recording["V_m"][0]
times = recording["times"]
```

nngt.simulation.**make_nest_network**(*network*, *send_only=None*, *weights=True*)

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

Changed in version 0.8: Added *send_only* parameter.

Parameters

- **network** (*nngt.Network* or *nngt.SpatialNetwork*) – the network we want to reproduce in NEST.
- **send_only** (*int, str, or list of str, optional (default: None)*) – Restrict the nodes that are created in NEST to either inhibitory or excitatory neurons *send_only* $\in \{1, -1\}$ to a group or a list of groups.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.

Returns **gids** (*tuple or NodeCollection (nodes in NEST)*) – GIDs of the neurons in the NEST 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 recorder)*) – 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** (*list or NodeCollection of the recorders' gids*)
- **recordables** (*list 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 recorder)*) – 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** (*list or NodeCollection containing the recorders' gids*)
- **recordables** (*list of the recordables' names.*)

nngt.simulation.**plot_activity**(*gid_recorder=None*, *record=None*, *network=None*, *gids=None*, *axis=None*, *show=False*, *limits=None*, *histogram=False*, *title=None*, *fignum=None*, *label=None*, *sort=None*, *average=False*, *normalize=1.0*, *decimate=None*, *transparent=True*, *kernel_center=0.0*, *kernel_std=None*, *resolution=None*, *cut_gaussian=5.0*, ***kwargs*)

Plot the monitored activity.

Changed in version 1.2: Switched *hist* to *histogram* and default value to False.

Changed in version 1.0.1: Added *axis* parameter, restored missing *fignum* parameter.

Parameters

- **gid_recorder** (*tuple or list of tuples, optional (default: None)*) – The gids of the recording devices. If None, then all existing spike_recs 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.
- **axis** (*matplotlib axis object, optional (default: new one)*) – Axis that should be use to plot the activity. This takes precedence over *fignum*.
- **show** (*bool, optional (default: False)*) – Whether to show the plot right away or to wait for the next plt.show().
- **histogram** (*bool, optional (default: False)*) – 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, or dict, optional (default: None)*) – Plot the activity on an existing figure (from `figure.number`). This parameter is ignored if *axis* is provided.
- **label** (*str or list, optional (default: None)*) – Add labels to the plot (one per recorder).
- **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 neuronal property) 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 or list, optional (default: None)*) – Normalize the recorded results by a given float. If a list is provided, there should be one entry per voltmeter or multimeter in the recorders. If the recording was done through *monitor_groups*, the population can be passed to normalize the data by the nuber of nodes in each group.
- **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.
- **kernel_center** (*float, optional (default: 0.)*) – Temporal shift of the Gaussian kernel, in ms (for the histogram).
- **kernel_std** (*float, optional (default: 0.5% of simulation time)*) – Characteristic width of the Gaussian kernel (standard deviation) in ms (for the histogram).
- **resolution** (float or array, optional (default: *0.1*kernel_std*)) – The resolution at which the firing rate values will be computed. Choosing a value smaller than *kernel_std* is strongly advised. If resolution is an array, it will be considered as the times were the firing rate should be computed (for the histogram).
- **cut_gaussian** (*float, optional (default: 5.)*) – Range over which the Gaussian will be computed (for the histogram). By default, we consider the 5-sigma range. Decreasing this value will increase speed at the cost of lower fidelity; increasing it with increase the fidelity at the cost of speed.
- ****kwargs** (*dict*) – "color" and "alpha" values can be overriden here.

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

Returns **lines** (list of lists of `matplotlib.lines.Line2D`) – Lines containing the data that was plotted, grouped by figure.

nngt.simulation.**randomize_neural_states**(*network*, *instructions*, *groups=None*, *nodes=None*, *make_nest=False*)

Randomize the neural states according to the instructions.

Changed in version 0.8: Changed *ids* to *nodes* argument.

Parameters

- **network** (*Network* subclass instance) – Network that will be simulated.
- **instructions** (*dict*) – Variables to initialize. Allowed keys are "V_m" and "w". Values are 3-tuples of type `("distrib_name", double, double)`.
- **groups** (list of *NeuralGroup*, optional (default: None)) – If provided, only the neurons belonging to these groups will have their properties randomized.

- **nodes** (*array-like, optional (default: all neurons)*) – NNGT ids of the neurons that will have their status randomized.
- **make_nest** (*bool, optional (default: False)*) – If `True` and network has not been converted to NEST, automatically generate the network, else raises an exception.

Example

```
instructions = {
    "V_m": ("uniform", -80., -60.),
    "w": ("normal", 50., 5.)
}
```

nngt.simulation.**raster_plot**(*times*, *senders*, *limits=None*, *title='Spike raster'*, *histogram=False*, *num_bins=1000*, *color='b'*, *decimate=None*, *axis=None*, *fignum=None*, *label=None*, *show=True*, *sort=None*, *sort_attribute=None*, *network=None*, *transparent=True*, *kernel_center=0.0*, *kernel_std=30.0*, *resolution=None*, *cut_gaussian=5.0*, ***kwargs*)

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

Changed in version 1.2: Switched *hist* to *histogram*.

Changed in version 1.0.1: Added *axis* parameter.

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.
- **histogram** (*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).
- **axis** (*matplotlib axis object, optional (default: new one)*) – Axis that should be use to plot the activity.
- **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().
- **kernel_center** (*float, optional (default: 0.)*) – Temporal shift of the Gaussian kernel, in ms.
- **kernel_std** (*float, optional (default: 30.)*) – Characteristic width of the Gaussian kernel (standard deviation) in ms.

- **resolution** (float or array, optional (default: *0.1*kernel_std*)) – The resolution at which the firing rate values will be computed. Choosing a value smaller than *kernel_std* is strongly advised. If resolution is an array, it will be considered as the times were the firing rate should be computed.
- **cut_gaussian** (*float, optional (default: 5.)*) – Range over which the Gaussian will be computed (for the histogram). By default, we consider the 5-sigma range. Decreasing this value will increase speed at the cost of lower fidelity; increasing it with increase the fidelity at the cost of speed.

Returns **lines** (list of `matplotlib.lines.Line2D`) – Lines containing the data that was plotted.

nngt.simulation.**reproducible_weights**(*weights*, *neuron_model*, *di_param={}*, *timestep=0.05*, *simtime=50.0*, *num_bins=1000*, *log=False*)

Find the values of the connection weights that will give PSP responses of *min_weight* and *max_weight* in mV.

Parameters

- **weights** (*list of floats*) – Exact desired synaptic weights.
- **neuron_model** (*string*) – Name of the model used.
- **di_param** (*dict, optional (default: {})*) – Parameters of the model, default parameters if not supplied.
- **timestep** (*float, optional (default: 0.01)*) – Timestep of the simulation in ms.
- **simtime** (*float, optional (default: 10.)*) – Simulation time in ms (default: 10).
- **num_bins** (*int, optional (default: 10000)*) – Number of bins used to discretize the exact synaptic weights.
- **log** (*bool, optional (default: False)*) – Whether bins should use a logarithmic scale.

Note: If the parameters used are not the default ones, they MUST be provided, otherwise the resulting weights will likely be WRONG.

---

nngt.simulation.**save_spikes**(*filename*, *recorder=None*, *network=None*, *save_positions=True*, ***kwargs*)

Plot the monitored activity.

New in version 0.7.

Parameters

- **filename** (*str*) – Path to the file where the activity should be saved.
- **recorder** (*tuple or list of tuples, optional (default: None)*) – The NEST gids of the recording devices. If None, then all existing spike recorders are used.
- **network** (*Network* or subclass, optional (default: None)) – Network which activity will be monitored.
- **save_positions** (*bool, optional (default: True)*) – Whether to include the position of the neurons in the file; this requires *network* to be provided.
- ****kwargs** (see `numpy.savetxt()`)

nngt.simulation.**set_minis**(*network*, *base_rate*, *weight*, *syn_type=1*, *nodes=None*, *gids=None*)

Mimick spontaneous release of neurotransmitters, called miniature PSCs or "minis" that can occur at excitatory (mEPSCs) or inhibitory (mIPSCs) synapses. These minis consists in only a fraction of the usual strength of a spike- triggered PSC and can be modeled by a Poisson process. This Poisson process occurs independently at

every synapse of a neuron, so a neuron receiving k inputs will be subjected to these events with a rate $k * \lambda$, where λ is the base rate.

Parameters

- **network** (*Network* object) – Network on which the minis should be simulated.
- **base_rate** (*float*) – Rate for the Poisson process on one synapse (λ), in Hz.
- **weight** (*float or array of size N*) – Amplitude of a minitature post-synaptic event.
- **syn_type** (*int, optional (default: 1)*) – Synaptic type of the noisy connections. By default, mEPSCs are generated, by taking into account only the excitatory degrees and synaptic weights. To setup mIPSCs, used *syn_type=-1*.
- **nodes** (*array-like (size N), optional (default: all nodes)*) – NNGT ids of the neurons that should be subjected to minis.
- **gids** (*array-like (size N), optional (default: all neurons)*) – NEST gids of the neurons that should be subjected to minis.

Note: *nodes* and *gids* are not compatible, only one one the two arguments can be used in any given call to *set_minis*.

---

nngt.simulation.**set_noise**(*gids*, *mean*, *std*)
Submit neurons to a current white noise.

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*, *syn_spec=None*, ***kwargs*)
Submit neurons to a Poissonian rate of spikes.

Changed in version 2.0: Added *kwargs*.

Parameters

- **gids** (*tuple*) – NEST gids of the target neurons.
- **rate** (*float*) – Rate of the spike train (in Hz).
- **syn_spec** (*dict, optional (default: static synapse with weight 1)*) – Properties of the connection between the `poisson_generator` object and the target neurons.
- ****kwargs** (*dict*) – Other optional parameters for the *poisson_generator*.

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.

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Properties of graph components*

Note: This library provides many tools which will (or not) be loaded on startup depending on the python packages available on your computer. The default behaviour of those tools is set in the *~/.nngt/nngt.conf* file (see Configuration). Moreover, to see all potential messages related to the import of those tools, you can use the logging function of NNGT, either by setting the *log_level* value to *INFO*, or by setting *log_to_file* to True, and having a look at the log file in *~/.nngt/log/*.

2.2.2 Description

The graph objects

Neural networks are described by four graph classes which contain a graph object from the chosen graph library (e.g. `gt.Graph`, `igraph.Graph`, or `networkx.Graph`):

- *`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 (for neuroscience, this includes the connections' type – inhibitory or excitatory – and its synaptic weight, which is always positive)

Additional properties

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

The graph objects can have other attributes, such as:

- *`shape`*, for *`SpatialGraph`* and *`SpatialNetwork`*, describes the spatial delimitations of the nodes' environment (e.g. many *in vitro* culture of neurons are contained in circular dishes),
- *`structure`* divides the graph into groups and can facilitate graph generation and analysis,
- *`population`*, for *`Network`*, contains informations on the various groups of neurons that exist in the network (for instance inhibitory and excitatory neurons can be grouped together), and is the updated version of `structure` for neuroscientific projects.

Graph-theoretical models

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

Main module (API)

Overview

For more details regarding the main classes, see:

Graph classes

NNGT provides four main graph classes that provide specific features to work as conveniently as possible with different object types: topological versus space-embedded graphs or neuronal networks.

nngt.Graph(*args, **kwargs)	The basic graph class, which inherits from a library class such as `graph_tool.Graph`, `networkx.DiGraph`, or `igraph.Graph`.
nngt.SpatialGraph(*args, **kwargs)	The detailed class that inherits from *Graph* and implements additional properties to describe spatial graphs (i.e.
nngt.Network(*args, **kwargs)	The detailed class that inherits from *Graph* and implements additional properties to describe various biological functions and interact with the NEST simulator.
nngt.SpatialNetwork(*args, **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.

A summary of the methods provided by these classes as well as more detailed descriptions are provided below. Unless specified, child classes can use all methods from the parent class (the only exception is *set_types()* which is not available to the *Network* subclasses).

- *Details*

Summary of the class members and methods

Graph

The main class for topological graphs.

nngt.Graph(*args, **kwargs)	The basic graph class, which inherits from a library class such as `graph_tool.Graph`, `networkx.DiGraph`, or `igraph.Graph`.
nngt.Graph.adjacency_matrix([types, ...])	Return the graph adjacency matrix.
nngt.Graph.clear_all_edges()	Remove all edges from the graph
nngt.Graph.copy()	Returns a deepcopy of the current *Graph* instance
nngt.Graph.delete_edges(edges)	Remove a list of edges
nngt.Graph.delete_nodes(nodes)	Remove nodes (and associated edges) from the graph.
nngt.Graph.edge_attributes	Access edge attributes.
nngt.Graph.edge_id(edge)	Return the ID a given edge or a list of edges in the graph.
nngt.Graph.edge_nb()	Number of edges in the graph
nngt.Graph.edges_array	Edges of the graph, sorted by order of creation, as an array of 2-tuple.
nngt.Graph.from_file(filename[, fmt, ...])	Import a saved graph from a file.
nngt.Graph.from_library(library_graph[, ...])	Create a *Graph* by wrapping a graph object from one of the supported libraries.
nngt.Graph.from_matrix(matrix[, weighted, ...])	Creates a *Graph* from a `scipy.sparse` matrix or a dense matrix.
nngt.Graph.get_attribute_type(attribute_name)	Return the type of an attribute (e.g.
nngt.Graph.get_betweenness([btype, weights])	Returns the normalized betweenness centrality of the nodes and edges.
nngt.Graph.get_degrees([mode, nodes, ...])	Degree sequence of all the nodes.
nngt.Graph.get_delays([edges])	Returns the delays of all or a subset of the edges.
nngt.Graph.get_density()	Density of the graph: $\frac{E}{N^2}$, where E is the number of edges and N the number of nodes.
nngt.Graph.get_edge_attributes([edges, name])	Attributes of the graph's edges.
nngt.Graph.get_edge_types([edges])	Return the type of all or a subset of the edges.
nngt.Graph.get_edges([attribute, value, ...])	Return the edges in the network fulfilling a given condition.
nngt.Graph.get_node_attributes([nodes, name])	Attributes of the graph's edges.
nngt.Graph.get_nodes([attribute, value])	Return the nodes in the network fulfilling a given condition.
nngt.Graph.get_structure_graph()	Return a coarse-grained version of the graph containing one node per *nngt.Group*.
nngt.Graph.get_weights([edges])	Returns the weights of all or a subset of the edges.
nngt.Graph.graph	Returns the underlying library object.
nngt.Graph.graph_id	Unique `int` identifying the instance.
nngt.Graph.has_edge(edge)	Whether *edge* is present in the graph.
nngt.Graph.is_connected([mode])	Return whether the graph is connected.
nngt.Graph.is_directed()	Whether the graph is directed or not
nngt.Graph.is_network()	Whether the graph is a subclass of *Network* (i.e.

continues on next page

Table 3 – continued from previous page

nngt.Graph.is_spatial()	Whether the graph is embedded in space (i.e.
nngt.Graph.is_weighted()	Whether the edges have weights
nngt.Graph.make_network(graph, neural_pop[, ...])	Turn a *Graph* object into a *Network*, or a *SpatialGraph* into a *SpatialNetwork*.
nngt.Graph.make_spatial(graph[, shape, ...])	Turn a *Graph* object into a *SpatialGraph*, or a *Network* into a *SpatialNetwork*.
nngt.Graph.name	Name of the graph.
nngt.Graph.neighbours(node[, mode])	Return the neighbours of *node*.
nngt.Graph.new_edge(source, target[, ...])	Adding a connection to the graph, with optional properties.
nngt.Graph.new_edge_attribute(name, value_type)	Create a new attribute for the edges.
nngt.Graph.new_edges(edge_list[, ...])	Add a list of edges to the graph.
nngt.Graph.new_node([n, neuron_type, ...])	Adding a node to the graph, with optional properties.
nngt.Graph.new_node_attribute(name, value_type)	Create a new attribute for the nodes.
nngt.Graph.node_attributes	Access node attributes.
nngt.Graph.node_nb()	Number of nodes in the graph
nngt.Graph.num_graphs()	Returns the number of alive instances.
nngt.Graph.set_delays([delay, elist, ...])	Set the delay for spike propagation between neurons.
nngt.Graph.set_edge_attribute(attribute[, ...])	Set attributes to the connections between neurons.
nngt.Graph.set_name([name])	Set graph name
nngt.Graph.set_node_attribute(attribute[, ...])	Set attributes to the connections between neurons.
nngt.Graph.set_types(edge_type[, nodes, ...])	Set the synaptic/connection types.
nngt.Graph.set_weights([weight, elist, ...])	Set the synaptic weights.
nngt.Graph.structure	Object structuring the graph into specific groups.
nngt.Graph.to_file(filename[, fmt, ...])	Save graph to file; options detailed below.
nngt.Graph.to_undirected([combine_numeric_eattr])	Convert the graph to its undirected variant.
nngt.Graph.type	Type of the graph.

SpatialGraph

Subclass of *Graph* providing additional tools to work with spatial graphs. It works together with the *Shape* object from the *geometry* module.

nngt.SpatialGraph(*args, **kwargs)	The detailed class that inherits from *Graph* and implements additional properties to describe spatial graphs (i.e.
nngt.SpatialGraph.get_positions([nodes])	Returns a copy of the nodes' positions as a (N, 2) array.
nngt.SpatialGraph.set_positions(positions[, ...])	Set the nodes' positions as a (N, 2) array.
nngt.SpatialGraph.shape	The environment's spatial structure.

Network

Subclass of *Graph* providing additional tools to work with neuronal networks. It works together with the *NeuralPop* object.

nngt.Network(*args, **kwargs)	The detailed class that inherits from *Graph* and implements additional properties to describe various biological functions and interact with the NEST simulator.
nngt.Network.exc_and_inhib(size[, iratio, ...])	Generate a network containing a population of two neural groups: inhibitory and excitatory neurons.
nngt.Network.from_gids(gids[, ...])	Generate a network from gids.
nngt.Network.get_neuron_type(neuron_ids)	Return the type of the neurons (+1 for excitatory, -1 for inhibitory).
nngt.Network.id_from_nest_gid(gids)	Return the ids of the nodes in the *nngt.Network* instance from the corresponding NEST gids.
nngt.Network.nest_gids	
nngt.Network.neuron_properties(idx_neuron)	Properties of a neuron in the graph.
nngt.Network.num_networks()	Returns the number of alive instances.
nngt.Network.population	*NeuralPop* that divides the neurons into groups with specific properties.
nngt.Network.to_nest([send_only, weights])	Send the network to NEST.
nngt.Network.uniform(size[, neuron_model, ...])	Generate a network containing only one type of neurons.

SpatialNetwork

Subclass of *Graph* providing additional tools to work with spatial neuronal networks. It works together with both *NeuralPop* and the *Shape* object from the *geometry* module.

nngt.SpatialNetwork(*args, **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.
nngt.SpatialNetwork.exc_and_inhib(size[, ...])	Generate a network containing a population of two neural groups: inhibitory and excitatory neurons.
nngt.SpatialNetwork.from_gids(gids[, ...])	Generate a network from gids.
nngt.SpatialNetwork.get_neuron_type(neuron_ids)	Return the type of the neurons (+1 for excitatory, -1 for inhibitory).
nngt.SpatialNetwork.get_positions([nodes])	Returns a copy of the nodes' positions as a (N, 2) array.
nngt.SpatialNetwork.id_from_nest_gid(gids)	Return the ids of the nodes in the *nngt.Network* instance from the corresponding NEST gids.
nngt.SpatialNetwork.nest_gids	
nngt.SpatialNetwork.neuron_properties(idx_neuron)	Properties of a neuron in the graph.
nngt.SpatialNetwork.num_networks()	Returns the number of alive instances.
nngt.SpatialNetwork.population	*NeuralPop* that divides the neurons into groups with specific properties.
nngt.SpatialNetwork.set_positions(positions)	Set the nodes' positions as a (N, 2) array.
nngt.SpatialNetwork.shape	The environment's spatial structure.

continues on next page

Table 6 – continued from previous page

nngt.SpatialNetwork.to_nest([send_only, weights])	Send the network to NEST.
nngt.SpatialNetwork.uniform(size[, ...])	Generate a network containing only one type of neurons.

Details

class nngt.**Graph**(**args*, ***kwargs*)

The basic graph class, which inherits from a library class such as graph_tool.Graph, networkx.DiGraph, or igraph.Graph.

The objects provides several functions to easily access some basic properties.

Initialize Graph instance

Changed in version 2.0: Renamed *from_graph* to *copy_graph*.

Changed in version 2.2: Added *structure* 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.
- **copy_graph** (*Graph*, optional) – An optional *Graph* that will be copied.
- **structure** (*Structure*, optional (default: None)) – A structure dividing the graph into specific groups, which can be used to generate specific connectivities and visualise the connections in a more coarse-grained manner.
- **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.

Note: When using *copy_graph*, only the topological properties are copied (nodes, edges, and attributes), spatial and biological properties are ignored. To copy a graph exactly, use *copy()*.

Returns **self** (*Graph*)

adjacency_matrix(*types=False*, *weights=False*, *mformat='csr'*)

Return the graph adjacency matrix.

Note: Source nodes are represented by the rows, targets by the corresponding columns.

Parameters

- **types** (*bool, optional (default: False)*) – Wether the edge types should be taken into account (negative values for inhibitory connections).

- **weights** (*bool or string, optional (default: False)*) – Whether the adjacecy matrix should be weighted. If True, all connections are multiply bythe associated synaptic strength; if weight is a string, the connections are scaled bythe corresponding edge attribute.
- **mformat** (*str, optional (default: "csr")*) – Type of `scipy.sparse` matrix that will be returned, by default `scipy.sparse.csr_matrix`.

Returns **mat** (`scipy.sparse` matrix) – The adjacency matrix of the graph.

`clear_all_edges()`
Remove all edges from the graph

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

`delete_edges(edges)`
Remove a list of edges

`delete_nodes(nodes)`
Remove nodes (and associated edges) from the graph.

`property edge_attributes`
Access edge attributes.

See also:

node_attributes, *get_edge_attributes*, *new_edge_attribute*, *set_edge_attribute*

`edge_id(edge)`
Return the ID a given edge or a list of edges in the graph. Raises an error if the edge is not in the graph or if one of the vertices in the edge is nonexistent.

Parameters **edge** (*2-tuple or array of edges*) – Edge descriptor (source, target).

Returns **index** (*int or array of ints*) – Index of the given *edge*.

`edge_nb()`
Number of edges in the graph

`property edges_array`
Edges of the graph, sorted by order of creation, as an array of 2-tuple.

`static from_file(filename, fmt='auto', separator=' ', secondary=';', attributes=None, attributes_types=None, notifier='@', ignore='#', from_string=False, name=None, directed=True, cleanup=False)`
Import a saved graph from a file.

Changed in version 2.0: Added optional *attributes_types* and *cleanup* arguments.

Parameters

- **filename** (*str*) – The path to the file.
- **fmt** (*str, optional (default: deduced from filename)*) – The format used to save the graph. Supported formats are: "neighbour" (neighbour list), "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 as library, detected if *filename* ends with '.gt').
- **separator** (*str, optional (default " ")*) – separator used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")

- **secondary** (*str, optional (default: ";")*) – Secondary separator 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. For "edge_list", attributes may also be present as additional columns after the source and the target.
- **attributes_types** (*dict, optional (default: str)*) – Backup information if the type of the attributes is not specified in the file. Values must be callables (types or functions) that will take the argument value as a string input and convert it to the proper type.
- **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` 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.
- **ignore** (*str, optional (default: "#")*) – Ignore lines starting with the *ignore* string.
- **name** (*str, optional (default: from file information or 'LoadedGraph')*) – The name of the graph.
- **directed** (*bool, optional (default: from file information or True)*) – Whether the graph is directed or not.
- **cleanup** (*bool, optional (default: False)*) – If true, removes nodes before the first one that appears in the edges and after the last one and renumber the nodes from 0.

Returns **graph** (*Graph* or subclass) – Loaded graph.

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

Create a *Graph* by wrapping a graph object from one of the supported libraries.

Parameters

- **library_graph** (*object*) – Graph object from one of the supported libraries (graph-tool, igraph, networkx).
- **name** (*str, optional (default: "ImportedGraph")*)
- ****kwargs** – Other standard arguments (see `__init__()`)

classmethod from_matrix(*matrix*, *weighted=True*, *directed=True*, *population=None*, *shape=None*, *positions=None*, *name=None*, ***kwargs*)

Creates a *Graph* from a `scipy.sparse` matrix or a dense matrix.

Parameters

- **matrix** (`scipy.sparse` matrix or `numpy.ndarray`) – 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.
- **population** (*NeuralPop*) – Population to associate to the new *Network*.
- **shape** (*Shape*, optional (default: None)) – Shape to associate to the new *SpatialGraph*.
- **positions** (*(N, 2) array*) – Positions, in a 2D space, of the N neurons.

- **name** (*str, optional*) – Graph name.

Returns *Graph*

`get_attribute_type`(*attribute_name*, *attribute_class=None*)
Return the type of an attribute (e.g. string, double, int).

Parameters

- **attribute_name** (*str*) – Name of the attribute.
- **attribute_class** (*str, optional (default: both)*) – Whether *attribute_name* is a "node" or an "edge" attribute.

Returns **type** (*str*) – Type of the attribute.

`get_betweenness`(*btype='both'*, *weights=None*)
Returns the normalized betweenness centrality of the nodes and edges.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **btype** (*str, optional (default 'both')*) – The centrality that should be returned (either 'node', 'edge', or 'both'). By default, both betweenness centralities are computed.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.

Returns

- **nb** (`numpy.ndarray`) – The nodes' betweenness if *btype* is 'node' or 'both'
- **eb** (`numpy.ndarray`) – The edges' betweenness if *btype* is 'edge' or 'both'

See also:

betweenness()

`get_degrees`(*mode='total'*, *nodes=None*, *weights=None*, *edge_type='all'*)
Degree sequence of all the nodes.

Changed in version 2.0: Changed *deg_type* to *mode*, *node_list* to *nodes*, *use_weights* to *weights*, and *edge_type* to *edge_type*.

Parameters

- **mode** (*string, optional (default: "total")*) – Degree type (among 'in', 'out' or 'total').
- **nodes** (*list, optional (default: None)*) – List of the nodes which degree should be returned
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **edge_type** (*int or str, optional (default: all)*) – Restrict to a given synaptic type ("excitatory", 1, or "inhibitory", -1), using either the "type" edge attribute for non-*Network* or the *inhibitory* nodes.

Returns

- **degrees** (`numpy.array`)
- *.. warning ::* – When using MPI with "nngt" (distributed) backend, returns only the degrees associated to local edges. "Complete" degrees are obtained by taking the sum of the results on all MPI processes.

`get_delays`(*edges=None*)
Returns the delays of all or a subset of the edges.

Changed in version 1.0.1: Added the possibility to ask for a subset of edges.

Parameters **edges** (*(E, 2) array, optional (default: all edges)*) – Edges for which the type should be returned.

Returns *the list of delays*

`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_attributes`(*edges=None*, *name=None*)
Attributes of the graph's edges.

Parameters

- **edges** (tuple or list of tuples, optional (default: `None`)) – Edge whose attribute should be displayed.
- **name** (str, optional (default: `None`)) – Name of the desired attribute.

Returns

- *Dict containing all graph's attributes (synaptic weights, delays...)*
- by default. If *edge* is specified, returns only the values for these
- edges. If *name* is specified, returns value of the attribute for each
- *edge.*

Note: The attributes values are ordered as the edges in *edges_array()* if *edges* is None.

See also:

get_node_attributes(), *new_edge_attribute()*, *set_edge_attribute()*, *new_node_attribute()*, *set_node_attribute()*

`get_edge_types`(*edges=None*)
Return the type of all or a subset of the edges.

Parameters **edges** (*(E, 2) array, optional (default: all edges)*) – Edges for which the type should be returned.

Returns *the list of types (1 for excitatory, -1 for inhibitory)*

`get_edges`(*attribute=None*, *value=None*, *source_node=None*, *target_node=None*)
Return the edges in the network fulfilling a given condition.

For undirected graphs, edges are always returned in the order (u, v) where $u <= v$.

Warning: Contrary to *edges_array()* that returns edges ordered by creation time (i.e. corresponding to the order of the edge attribute array), this function does not enforce any specific edge order. This also means that, if order does not matter, it may be faster to call `get_edges` that to call `edges_array`.

Parameters

- **attribute** (*str, optional (default: all nodes)*) – Whether the *attribute* of the returned edges should have a specific value.

- **value** (*object, optional (default : None)*) – If an *attribute* name is passed, then only edges with *attribute* being equal to *value* will be returned.
- **source_node** (*int or list of ints, optional (default: all nodes)*) – Retrict the edges to those stemming from *source_node*.
- **target_node** (*int or list of ints, optional (default: all nodes)*) – Retrict the edges to those arriving at *target_node*.

Returns *A list of edges (2-tuples).*

See also:

get_nodes(), *edge_attributes*, *edges_array()*

get_node_attributes(*nodes=None*, *name=None*)
Attributes of the graph's edges.

Changed in version 1.0.1: Corrected default behavior and made it the same as *get_edge_attributes()*.

New in version 0.9.

Parameters

- **nodes** (list of ints, optional (default: `None`)) – Nodes whose attribute should be displayed.
- **name** (str, optional (default: `None`)) – Name of the desired attribute.

Returns

- Dict containing all nodes attributes by default. If *nodes* is
- specified, returns a `dict` containing only the attributes of these
- nodes. If *name* is specified, returns a list containing the values of
- *the specific attribute for the required nodes (or all nodes if*
- *unspecified).*

See also:

get_edge_attributes(), *new_node_attribute()*, *set_node_attribute()*, new_edge_attributes(), *set_edge_attribute()*

get_nodes(*attribute=None*, *value=None*)
Return the nodes in the network fulfilling a given condition.

Parameters

- **attribute** (*str, optional (default: all nodes)*) – Whether the *attribute* of the returned nodes should have a specific value.
- **value** (*object, optional (default : None)*) – If an *attribute* name is passed, then only nodes with *attribute* being equal to *value* will be returned.

See also:

get_edges(), *node_attributes*

get_structure_graph()
Return a coarse-grained version of the graph containing one node per *nngt.Group*. Connections between groups are associated to the sum of all connection weights. If no structure is present, returns an empty Graph.

get_weights(*edges=None*)
Returns the weights of all or a subset of the edges.

Changed in version 1.0.1: Added the possibility to ask for a subset of edges.

Parameters **edges** (*(E, 2) array, optional (default: all edges)*) – Edges for which the type should be returned.

Returns *the list of weights*

property graph
Returns the underlying library object.

> **Warning:** Do not add or remove edges directly through this object.

See also:

Underlying graph objects and libraries, *Consistent tools for graph analysis*

property graph_id
Unique `int` identifying the instance.

has_edge(*edge*)
Whether *edge* is present in the graph.

New in version 2.0.

is_connected(*mode='strong'*)
Return whether the graph is connected.

Parameters **mode** (*str, optional (default: "strong")*) – Whether to test connectedness with directed ("strong") or undirected ("weak") connections.

References

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. 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=None*, *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 (default: None)) – 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*. The *shape* argument can be skipped if *positions* are given; in that case, the neurons will be embedded in a rectangle that contains them all.

property name
Name of the graph.

neighbours(*node*, *mode='all'*)
Return the neighbours of *node*.

Parameters

- **node** (*int*) – Index of the node of interest.
- **mode** (*string, optional (default: "all")*) – Type of neighbours that will be returned: "all" returns all the neighbours regardless of directionality, "in" returns the in-neighbours (also called predecessors) and "out" retruns the out-neighbours (or successors).

Returns **neighbours** (*set*) – The neighbours of *node*.

new_edge(*source*, *target*, *attributes=None*, *ignore=False*, *self_loop=False*)
Adding a connection to the graph, with optional properties.

Changed in version 2.0: Added *self_loop* argument to enable adding self-loops.

Parameters

- **source** (`int/node`) – Source node.
- **target** (`int/node`) – Target node.
- **attributes** (`dict`, optional (default: `{}`)) – Dictionary containing optional edge properties. If the graph is weighted, defaults to `{"weight": 1.}`, the unit weight for the connection (synaptic strength in NEST).
- **ignore** (*bool, optional (default: False)*) – If set to True, ignore attempts to add an existing edge and accept self-loops; otherwise an error is raised.
- **self_loop** (*bool, optional (default: False)*) – Whether to allow self-loops or not.

Returns *The new connection or None if nothing was added.*

`new_edge_attribute`(*name*, *value_type*, *values=None*, *val=None*)
Create a new attribute for the edges.

Parameters

- **name** (*str*) – The name of the new attribute.
- **value_type** (*str*) – Type of the attribute, among 'int', 'double', 'string', or 'object'
- **values** (*array, optional (default: None)*) – Values with which the edge attribute should be initialized. (must have one entry per node in the graph)
- **val** (*int, float or str , optional (default: None)*) – Identical value for all edges.

`new_edges`(*edge_list*, *attributes=None*, *check_duplicates=False*, *check_self_loops=True*, *check_existing=True*, *ignore_invalid=False*)
Add a list of edges to the graph.

Changed in version 2.0: Can perform all possible checks before adding new edges via the `check_duplicates` `check_self_loops`, and `check_existing` arguments.

Parameters

- **edge_list** (*list of 2-tuples or np.array of shape (edge_nb, 2)*) – List of the edges that should be added as tuples (source, target)
- **attributes** (`dict`, optional (default: `{}`)) – Dictionary containing optional edge properties. If the graph is weighted, defaults to `{"weight": ones}`, where `ones` is an array the same length as the *edge_list* containing a unit weight for each connection (synaptic strength in NEST).
- **check_duplicates** (*bool, optional (default: False)*) – Check for duplicate edges within *edge_list*.
- **check_self_loops** (*bool, optional (default: True)*) – Check for self-loops.
- **check_existing** (*bool, optional (default: True)*) – Check whether some of the edges in *edge_list* already exist in the graph or exist multiple times in *edge_list* (also performs *check_duplicates*).
- **ignore_invalid** (*bool, optional (default: False)*) – Ignore invalid edges: they are not added to the graph and are silently dropped. Unless this is set to true, an error is raised whenever one of the three checks fails.
- **.. warning::** – Setting *check_existing* to False will lead to undefined behavior if existing edges are provided! Only use it (for speedup) if you are sure that you are indeed only adding new edges.

Returns *Returns new edges only.*

`new_node`(*n=1*, *neuron_type=1*, *attributes=None*, *value_types=None*, *positions=None*, *groups=None*)
Adding a node to the graph, with optional properties.

Parameters

- **n** (*int, optional (default: 1)*) – Number of nodes to add.
- **neuron_type** (*int, optional (default: 1)*) – Type of neuron (1 for excitatory, -1 for inhibitory)
- **attributes** (*dict, optional (default: None)*) – Dictionary containing the attributes of the nodes.
- **value_types** (*dict, optional (default: None)*) – Dict of the *attributes* types, necessary only if the *attributes* do not exist yet.

- **positions** (*array of shape (n, 2), optional (default: None)*) – Positions of the neurons. Valid only for *SpatialGraph* or *SpatialNetwork*.
- **groups** (*str, int, or list, optional (default: None)*) – `NeuralGroup` to which the neurons belong. Valid only for *Network* or *SpatialNetwork*.

Returns *The node or a list of the nodes created.*

new_node_attribute(*name*, *value_type*, *values=None*, *val=None*)
Create a new attribute for the nodes.

Parameters

- **name** (*str*) – The name of the new attribute.
- **value_type** (*str*) – Type of the attribute, among 'int', 'double', 'string', or 'object'
- **values** (*array, optional (default: None)*) – Values with which the node attribute should be initialized. (must have one entry per node in the graph)
- **val** (*int, float or str , optional (default: None)*) – Identical value for all nodes.

See also:

new_edge_attribute(), *set_node_attribute()*, *get_node_attributes()*, *set_edge_attribute()*, *get_edge_attributes()*

property node_attributes
Access node attributes.

See also:

edge_attributes, *get_node_attributes*, *new_node_attribute*, *set_node_attribute*

node_nb()
Number of nodes in the graph

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.

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.

Parameters

- **attribute** (*str*) – The name of the attribute.
- **value_type** (*str*) – Type of the attribute, among 'int', 'double', 'string'
- **values** (*array, optional (default: None)*) – Values with which the edge attribute should be initialized. (must have one entry per node in the graph)
- **val** (*int, float or str , optional (default: None)*) – Identical value for all edges.
- **value_type** (*str, optional (default: None)*) – Type of the attribute, among 'int', 'double', 'string'. Only used if the attribute does not exist and must be created.
- **edges** (*list of edges or array of shape (E, 2), optional (default: all)*) – Edges whose attributes should be set. Others will remain unchanged.

See also:

set_node_attribute(), *get_edge_attributes()*, *new_edge_attribute()*, *new_node_attribute()*, *get_node_attributes()*

set_name(*name=None*)
Set graph name

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

Parameters

- **attribute** (*str*) – The name of the attribute.
- **value_type** (*str*) – Type of the attribute, among 'int', 'double', 'string'
- **values** (*array, optional (default: None)*) – Values with which the edge attribute should be initialized. (must have one entry per node in the graph)
- **val** (*int, float or str , optional (default: None)*) – Identical value for all edges.
- **value_type** (*str, optional (default: None)*) – Type of the attribute, among 'int', 'double', 'string'. Only used if the attribute does not exist and must be created.
- **nodes** (*list of nodes, optional (default: all)*) – Nodes whose attributes should be set. Others will remain unchanged.

See also:

set_edge_attribute(), *new_node_attribute()*, *get_node_attributes()*, *new_edge_attribute()*, *get_edge_attributes()*

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

Changed in version 2.0: Changed *syn_type* to *edge_type*.

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

- **edge_type** (*int, string, or array of ints*) – 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 *edge_type* nodes in the graph; if it is a list, ids of the *edge_type* nodes.
- **fraction** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as *edge_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.

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. Properties are as follow for the distributions
  - 'constant': 'value'
  - 'uniform': 'lower', 'upper'
  - 'gaussian': 'avg', 'std'
  - 'lognormal': 'position', 'scale'
- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

Note: If *distribution* and *parameters* are provided and the weights are set for the whole graph (*elist* is None), then the distribution properties will be kept as the new default for subsequent edges. That is, if new edges are created without specifying their weights, then these new weights will automatically be drawn from this previous distribution.

property structure

Object structuring the graph into specific groups.

Note: Points to *population* if the graph is a *Network*.

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

See also:

nngt.lib.save_to_file()

to_undirected(*combine_numeric_eattr='sum'*)
Convert the graph to its undirected variant.

Note: All non-numeric edge attributes will be discarded from the returned undirected graph.

Parameters **combine_numeric_eattr** (*str, optional (default: "sum")*) – How to combine numeric attributes from reciprocal edges. Can be either:

- "sum" (attributes are summed)
- "min" (smallest value is kept)
- "max" (largest value is kept)
- "mean" (the average of both attributes is taken)

In addition, *combine_numeric_eattr* can be a dictionary with one entry for each edge attribute.

property type
Type of the graph.

class nngt.**SpatialGraph**(**args*, ***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.

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.
- ****kwargs** (keyword arguments for *Graph* or) – *Shape* if no shape was given.

Returns **self** (SpatialGraph)

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

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

`set_positions`(*positions*, *nodes=None*)
Set the nodes' positions as a (N, 2) array.

Parameters

- **positions** (*array-like*) – List of positions, of shape (N, 2).
- **nodes** (*int or array-like, optional (default: all nodes)*) – List of the nodes for which the position should be set.

`property shape`
The environment's spatial structure.

`class nngt.Network`(**args*, ***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.
- **copy_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.
- **inh_weight_factor** (*float, optional (default: 1.)*) – Factor to apply to inhibitory synapses, to compensate for example the strength difference due to timescales between excitatory and inhibitory synapses.

Returns **self** (`Network`)

`classmethod exc_and_inhib`(*size*, *iratio=0.2*, *en_model='aeif_cond_alpha'*, *en_param=None*, *in_model='aeif_cond_alpha'*, *in_param=None*, *syn_spec=None*, ***kwargs*)
Generate a network containing a population of two neural groups: inhibitory and excitatory neurons.

Parameters

- **size** (*int*) – Number of neurons in the network.
- **i_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.
- **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.

- **syn_spec** (*dict, optional (default: static synapse)*) – Dictionary containg a directed edge between groups as key and the associated synaptic parameters for the post-synaptic neurons (i.e. those of the second group) as value. If provided, all connections between groups will be set according to the values contained in *syn_spec*. Valid keys are:
  - *('excitatory', 'excitatory')*
  - *('excitatory', 'inhibitory')*
  - *('inhibitory', 'excitatory')*
  - *('inhibitory', 'inhibitory')*

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

See also:

exc_and_inhib()

classmethod from_gids(*gids*, *get_connections=True*, *get_params=False*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*, ***kwargs*)

Generate a network from gids.

> **Warning:** Unless *get_connections* and *get_params* is True, or if your population is homogeneous and you provide the required information, the information contained by the network and its *population* attribute will be erroneous! To prevent conflicts the *to_nest()* function is not available. If you know what you are doing, you should be able to find a workaround...

Parameters

- **gids** (*array-like*) – Ids of the neurons in NEST or simply user specified ids.
- **get_params** (*bool, optional (default: True)*) – Whether the parameters should be obtained from NEST (can be very slow).
- **neuron_model** (*string, optional (default: None)*) – 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.

get_edge_types(*edges=None*)

Return the type of all or a subset of the edges. For all edges, the types are ordered according to the edges ids, i.e. in the same order as **:property:`~nngt.Graph.edges_array`**.

Changed in version 2.4: Updated it to make it compatible with the default *Graph* function, including the *edges* argument.

Parameters **edges** (*(E, 2) array, optional (default: all edges)*) – Edges for which the type should be returned.

Returns *the list of types (1 for excitatory, -1 for inhibitory)*

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.

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

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

> **Warning:** This function is not available for *Network* subclasses.

to_nest(*send_only=None*, *weights=True*)
Send the network to NEST.

See also:

make_nest_network() for parameters

classmethod uniform(*size*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*, ***kwargs*)
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**(**args*, ***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 SpatialNetwork 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)) – Population from which the network will be built.

Returns **self** (*SpatialNetwork*)

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

> **Warning:** This function is not available for *Network* subclasses.

Main functions

nngt.generate(di_instructions, **kwargs)	Generate a *Graph* or one of its subclasses from a `dict` containing all the relevant informations.
nngt.get_config([key, detailed])	Get the NNGT configuration as a dictionary.
nngt.load_from_file(filename[, fmt, ...])	Load a Graph from a file.
nngt.num_mpi_processes()	Returns the number of MPI processes (1 if MPI is not used)
nngt.on_master_process()	Check whether the current code is executing on the master process (rank 0) if MPI is used.
nngt.save_to_file(graph, filename[, fmt, ...])	Save a graph to file.
nngt.seed([msd, seeds])	Seed the random generator used by NNGT (i.e.
nngt.set_config(config[, value, silent])	Set NNGT's configuration.
nngt.use_backend(backend[, reloading, silent])	Allows the user to switch to a specific graph library as backend.

Details

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

Generate a *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.

See also:

generation

nngt.**get_config**(*key=None*, *detailed=False*)
Get the NNGT configuration as a dictionary.

Note: This function has no MPI barrier on it.

---

nngt.**load_from_file**(*filename*, *fmt='auto'*, *separator=' '*, *secondary=';'*, *attributes=None*, *attributes_types=None*, *notifier='@'*, *ignore='#'*, *name='LoadedGraph'*, *directed=True*, *cleanup=False*)
Load a Graph from a file.

Changed in version 2.0: Added optional *attributes_types* and *cleanup* arguments.

Warning: Support for GraphML and DOT formats are currently limited and require one of the non-default backends (DOT requires graph-tool).

Parameters

- **filename** (*str*) – The path to the file.
- **fmt** (*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').
- **separator** (*str, optional (default " ")*) – separator used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")
- **secondary** (*str, optional (default: ";")*) – Secondary separator 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. For "edge_list", attributes may also be present as additional columns after the source and the target.
- **attributes_types** (*dict, optional (default: str)*) – Backup information if the type of the attributes is not specified in the file. Values must be callables (types or functions) that will take the argument value as a string input and convert it to the proper type.
- **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` are of type (`list`, `bool`, `str`, `int`). Additional notifiers are `@type=SpatialGraph/Network/SpatialNetwork`, which must be followed by the relevant notifiers among `@shape`, `@structure`, and `@graph`.
- **ignore** (*str, optional (default: "#")*) – Ignore lines starting with the *ignore* string.
- **name** (*str, optional (default: from file information or 'LoadedGraph')*) – The name of the graph.

- **directed** (*bool, optional (default: from file information or True)*) – Whether the graph is directed or not.
- **cleanup** (*bool, optional (default: False)*) – If true, removes nodes before the first one that appears in the edges and after the last one and renumber the nodes from 0.

Returns **graph** (*Graph* or subclass) – Loaded graph.

nngt.**num_mpi_processes**()
Returns the number of MPI processes (1 if MPI is not used)

nngt.**on_master_process**()
Check whether the current code is executing on the master process (rank 0) if MPI is used.

Returns

- *True if rank is 0, if mpi4py is not present or if MPI is not used,*
- *otherwise False.*

nngt.**save_to_file**(*graph*, *filename*, *fmt='auto'*, *separator=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*)
Save a graph to file.

@todo: implement dot, xml/graphml, and gt formats

Parameters

- **graph** (*Graph* or subclass) – Graph to save.
- **filename** (*str*) – The path to the file.
- **fmt** (*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 as library, detected if *filename* ends with '.gt').
- **separator** (*str, optional (default " ")*) – separator used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")
- **secondary** (*str, optional (default: ";")*) – Secondary separator used to separate attributes in the case of custom formats.
- **attributes** (list, optional (default: `None`)) – List of names for the edge attributes present in the graph that will be saved to disk; by default (`None`), all attributes will be saved.
- **notifier** (*str, optional (default: "@")*) – Symbol specifying the following as meaningfull information. Relevant information are formatted `@info_name=info_value`, with `info_name` in ("attributes", "attr_types", "directed", "name", "size"). Additional notifiers are `@type=SpatialGraph/Network/SpatialNetwork`, which are followed by the relevant notifiers among `@shape`, `@structure`, and `@graph` to separate the sections.

Note: Positions are saved as bytes by `numpy.nparray.tostring()`

---

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

Parameters

- **msd** (*int, optional*) – Master seed for numpy *RandomState*. Must be convertible to 32-bit unsigned integers.
- **seeds** (*list of ints, optional*) – Seeds for *RandomState* (when using MPI). Must be convertible to 32-bit unsigned integers, one entry per MPI process.

nngt.**set_config**(*config*, *value=None*, *silent=False*)
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)
```

Notes

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

This function has an MPI barrier on it, so it must always be called on all processes.

See also:

get_config()

nngt.**use_backend**(*backend*, *reloading=True*, *silent=False*)
Allows the user to switch to a specific graph library as backend.

Warning: If *Graph* objects have already been created, they will no longer be compatible with NNGT methods.

Parameters

- **backend** (*string*) – Name of a graph library among 'graph_tool', 'igraph', 'networkx', or 'nngt'.
- **reloading** (*bool, optional (default: True)*) – Whether the graph objects should be reloaded through *reload* (this should always be set to True except when NNGT is first initiated!)
- **silent** (*bool, optional (default: False)*) – Whether the changes made to the configuration should be logged at the DEBUG (True) or INFO (False) level.

Side classes

The following side classes are used to structure graphs into groups that can then be used to generate specific connectivity patterns via the *connect_groups()* function or to assign specific properties to neuronal assemblies to use them in simulations with NEST.

nngt.Group([nodes, properties, name])	Class defining groups of nodes.
nngt.MetaGroup([nodes, properties, name])	Class defining a meta-group of nodes.
nngt.MetaNeuralGroup([nodes, neuron_type, ...])	Class defining a meta-group of neurons.
nngt.NeuralGroup([nodes, neuron_type, ...])	Class defining groups of neurons.
nngt.NeuralPop([size, parent, meta_groups, ...])	The basic class that contains groups of neurons and their properties.
nngt.Structure([size, parent, meta_groups])	The basic class that contains groups of nodes and their properties.

Summary of the classes

A summary of the methods provided by these classes as well as more detailed descriptions are provided below. Unless specified, child classes can use all methods from the parent class (*MetaGroup*, *NeuralGroup*, and *MetaNeuralGroup* inherit from *Group* while *NeuralPop* inherits from *Structure*).

- *Group*
- *NeuralGroup*
- *Structure*
- *NeuralPop*

Group

nngt.Group([nodes, properties, name])	Class defining groups of nodes.
nngt.Group.add_nodes(nodes)	Add nodes to the group.
nngt.Group.copy()	Return a deep copy of the group.
nngt.Group.ids	Ids of the nodes belonging to the group.
nngt.Group.is_metagroup	Whether the group is a meta-group.
nngt.Group.is_valid	i.e.
nngt.Group.name	The name of the group.
nngt.Group.parent	Return the parent *Structure* of the group
nngt.Group.properties	Properties associated to the nodes in the group.
nngt.Group.size	The (desired) number of nodes in the group.

NeuralGroup

nngt.NeuralGroup([nodes, neuron_type, ...])	Class defining groups of neurons.
nngt.NeuralGroup.has_model	Whether this group have been given a model for the simulation.
nngt.NeuralGroup.nest_gids	Global ids associated to the neurons in the NEST simulator.
nngt.NeuralGroup.neuron_model	Model that will be used to simulate the neurons of this group.
nngt.NeuralGroup.neuron_param	Parameters associated to the group's neurons.
nngt.NeuralGroup.neuron_type	Type of the neurons in the group (excitatory or inhibitory).

Structure

nngt.Structure([size, parent, meta_groups])	The basic class that contains groups of nodes and their properties.
nngt.Structure.add_meta_group(group[, name, ...])	Add an existing meta group to the structure.
nngt.Structure.add_to_group(group_name, ids)	Add nodes to a specific group.
nngt.Structure.copy()	Return a deep copy of the structure.
nngt.Structure.create_group(nodes, name[, ...])	Create a new group in the structure.
nngt.Structure.create_meta_group(nodes, name)	Create a new meta group and add it to the structure.
nngt.Structure.from_groups(groups[, names, ...])	Make a *Structure* object from a (list of) *Group* object(s).
nngt.Structure.get_group(nodes[, numbers])	Return the group of the nodes.
nngt.Structure.get_properties([key, groups, ...])	Return the properties of nodes or groups of nodes in the structure.
nngt.Structure.ids	Return all the ids of the nodes inside the structure.
nngt.Structure.is_valid	Whether the structure is consistent with the associated network.
nngt.Structure.meta_groups	
nngt.Structure.parent	Parent *Network*, if it exists, otherwise `None`.
nngt.Structure.set_properties(props[, ...])	Set the parameters of specific nodes or of a whole group.
nngt.Structure.size	Number of nodes in this structure.

NeuralPop

nngt.NeuralPop([size, parent, meta_groups, ...])	The basic class that contains groups of neurons and their properties.
nngt.NeuralPop.exc_and_inhib(size[, iratio, ...])	Make a NeuralPop with a given ratio of inhibitory and excitatory neurons.
nngt.NeuralPop.excitatory	Return the ids of all excitatory nodes inside the population.
nngt.NeuralPop.from_network(graph, *args)	Make a NeuralPop object from a network.

continues on next page

Table 12 – continued from previous page

nngt.NeuralPop.get_param([groups, neurons, ...])	Return the *element* (neuron or synapse) parameters for neurons or groups of neurons in the population.
nngt.NeuralPop.has_models	Whether all groups have been assigned a neuronal model.
nngt.NeuralPop.inhibitory	Return the ids of all inhibitory nodes inside the population.
nngt.NeuralPop.nest_gids	Return the NEST gids of the nodes inside the population.
nngt.NeuralPop.set_model(model[, group])	Set the groups' models.
nngt.NeuralPop.set_neuron_param(params[, ...])	Set the parameters of specific neurons or of a whole group.
nngt.NeuralPop.syn_spec	The properties of the synaptic connections between groups.
nngt.NeuralPop.uniform(size[, neuron_type, ...])	Make a NeuralPop of identical neurons belonging to a single "default" group.

Details

class nngt.**Group**(*nodes=None*, *properties=None*, *name=None*, ***kwargs*)

Class defining groups of nodes.

Its main variables are:

Variables

- **ids** – list of int the ids of the nodes in this group.
- **properties** – dict, optional (default: {}) properties associated to the nodes
- **is_metagroup** – bool whether the group is a meta-group or not.

Note: A *Group* contains a set of nodes that are unique; the size of the group is the number of unique nodes contained in the group. Passing non-unique nodes will automatically convert them to a unique set.

Warning: Equality between `Group`s only compares the size and ``properties`` attributes. This means that groups differing only by their `ids` will register as equal.

Calling the class creates a group of nodes. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the nodes in an existing graph.
- **properties** (*dict, optional (default: {})*) – Dictionary containing the properties associated to the nodes.

Returns A new *Group* instance.

add_nodes(*nodes*)

Add nodes to the group.

Parameters **nodes** (*list of ids*)

copy()
Return a deep copy of the group.

property ids
Ids of the nodes belonging to the group.

property is_metagroup
Whether the group is a meta-group.

property is_valid
i.e. if it has either a size or some ids associated to it.

Type Whether the group can be used in a structure

property name
The name of the group.

property parent
Return the parent *Structure* of the group

property properties
Properties associated to the nodes in the group.

property size
The (desired) number of nodes in the group.

class nngt.**MetaGroup**(*nodes=None*, *properties=None*, *name=None*, ***kwargs*)
Class defining a meta-group of nodes.

Its main variables are:

Variables **ids** – list of int the ids of the nodes in this group.

Calling the class creates a group of nodes. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the nodes in an existing graph.
- **name** (*str, optional (default: "Group N")*) – Name of the meta-group.

Returns A new *MetaGroup* object.

class nngt.**MetaNeuralGroup**(*nodes=None*, *neuron_type='undefined'*, *neuron_model=None*, *neuron_param=None*, *name=None*, ***kwargs*)
Class defining a meta-group of neurons.

Its main variables are:

Variables

- **ids** – list of int the ids of the neurons in this group.
- **is_metagroup** – bool whether the group is a meta-group or not (*neuron_type* is None for meta-groups)

Calling the class creates a group of neurons. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the neurons in an existing graph.

- **name** (*str, optional (default: "Group N")*) – Name of the meta-group.

Returns A new *MetaNeuralGroup* object.

property excitatory
Return the ids of all excitatory nodes inside the meta-group.

property inhibitory
Return the ids of all inhibitory nodes inside the meta-group.

property properties
Properties associated to the nodes in the group.

class nngt.**NeuralGroup**(*nodes=None*, *neuron_type='undefined'*, *neuron_model=None*, *neuron_param=None*, *name=None*, ***kwargs*)
Class defining groups of neurons.

Its main variables are:

Variables

- **ids** – list of int the ids of the neurons in this group.
- **neuron_type** – int the default is 1 for excitatory neurons; -1 is for inhibitory neurons; meta-groups must have *neuron_type* set to None
- **neuron_model** – str, 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)
- **is_metagroup** – bool whether the group is a meta-group or not (*neuron_type* is None for meta-groups)

Warning: Equality between NeuralGroup`s only compares the size and neuronal type, ``model` and param attributes. This means that groups differing only by their ids will register as equal.

Calling the class creates a group of neurons. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the neurons in an existing graph.
- **neuron_type** (*int, optional (default: 1)*) – Type of the neurons (1 for excitatory, -1 for inhibitory) or None if not relevant (only allowed for metag roups).
- **neuron_model** (*str, optional (default: None)*) – NEST model for the neuron.
- **neuron_param** (*dict, optional (default: model defaults)*) – Dictionary containing the parameters associated to the NEST model.

Returns A new NeuralGroup instance.

copy()
Return a deep copy of the group.

property has_model
Whether this group have been given a model for the simulation.

property ids
Ids of the nodes belonging to the group.

property nest_gids
Global ids associated to the neurons in the NEST simulator.

property neuron_model
Model that will be used to simulate the neurons of this group.

property neuron_param
Parameters associated to the group's neurons.

property neuron_type
Type of the neurons in the group (excitatory or inhibitory).

property properties
Properties of the neurons in this group, including *neuron_type*, *neuron_model* and *neuron_params*.

class nngt.**NeuralPop**(*size=None*, *parent=None*, *meta_groups=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.
- **size** – int, Returns the number of neurons in the population.
- **syn_spec** – dict, Dictionary containing informations about the synapses between the different groups in the population.
- **is_valid** – bool, Whether this population can be used to create a network in NEST.

Initialize NeuralPop instance.

Parameters

- **size** (*int, optional (default: 0)*) – Number of neurons that the population will contain.
- **parent** (*Network*, optional (default: None)) – Network associated to this population.
- **meta_groups** (dict of str/*NeuralGroup* items) – Optional set of groups. Contrary to the primary groups which define the population and must be disjoint, meta groups can overlap: a neuron can belong to several different meta groups.
- **with_models** (bool) – whether the population's groups contain models to use in NEST
- ***args** (*items for OrderedDict parent*)
- ****kwargs** (dict)

Returns **pop** (*NeuralPop* object.)

add_to_group(*group_name*, *ids*)
Add neurons to a specific group.

Parameters

- **group_name** (*str or int*) – Name or index of the group.
- **ids** (*list or 1D-array*) – Neuron ids.

copy()
Return a deep copy of the population.

create_group(*neurons*, *name*, *neuron_type=1*, *neuron_model=None*, *neuron_param=None*, *replace=False*)
Create a new group in the population.

Parameters

- **neurons** (*int or array-like*) – Desired number of neurons or list of the neurons indices.
- **name** (*str*) – Name of the group.
- **neuron_type** (*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.
- **replace** (*bool, optional (default: False)*) – Whether to override previous exiting meta group with same name.

`create_meta_group`(*neurons*, *name*, *neuron_param=None*, *replace=False*)

Create a new meta group and add it to the population.

Parameters

- **neurons** (*int or array-like*) – Desired number of neurons or list of the neurons indices.
- **name** (*str*) – Name of the group.
- **neuron_type** (*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.
- **replace** (*bool, optional (default: False)*) – Whether to override previous exiting meta group with same name.

`classmethod exc_and_inhib`(*size*, *iratio=0.2*, *en_model='aeif_cond_alpha'*, *en_param=None*, *in_model='aeif_cond_alpha'*, *in_param=None*, *syn_spec=None*, *parent=None*, *meta_groups=None*)

Make a NeuralPop with a given ratio of inhibitory and excitatory neurons.

Parameters

- **size** (*int*) – Number of neurons contained by the population.
- **iratio** (*float, optional (default: 0.2)*) – Fraction of the neurons that will be inhibitory.
- **en_model** (*str, optional (default: default_neuron)*) – Name of the NEST model that will be used to describe excitatory neurons.
- **en_param** (*dict, optional (default: default NEST parameters)*) – Parameters of the excitatory neuron model.
- **in_model** (*str, optional (default: default_neuron)*) – Name of the NEST model that will be used to describe inhibitory neurons.
- **in_param** (*dict, optional (default: default NEST parameters)*) – Parameters of the inhibitory neuron model.
- **syn_spec** (*dict, optional (default: static synapse)*) – Dictionary containg a directed edge between groups as key and the associated synaptic parameters for the post-synaptic neurons (i.e. those of the second group) as value. If provided, all connections between groups will be set according to the values contained in *syn_spec*. Valid keys are:
  - *('excitatory', 'excitatory')*

- *('excitatory', 'inhibitory')*
- *('inhibitory', 'excitatory')*
- *('inhibitory', 'inhibitory')*

- **parent** (*Network*, optional (default: None)) – Network associated to this population.
- **meta_groups** (list dict of str/*NeuralGroup* items) – Additional set of groups which can overlap: a neuron can belong to several different meta groups. Contrary to the primary 'excitatory' and 'inhibitory' groups, meta groups are therefore no necessarily disjoint. If all meta-groups have a name, they can be passed directly through a list; otherwise a dict is necessary.

See also:

`nest.Connect()`, `as`

property excitatory

Return the ids of all excitatory nodes inside the population.

classmethod from_groups(*groups*, *names=None*, *syn_spec=None*, *parent=None*, *meta_groups=None*, *with_models=True*)

Make a NeuralPop object from a (list of) *NeuralGroup* object(s).

Parameters

- **groups** (list of *NeuralGroup* objects) – Groups that will be used to form the population. Note that a given neuron can only belong to a single group, so the groups should form pairwise disjoints complementary sets.
- **names** (*list of str, optional (default: None)*) – Names that can be used as keys to retreive a specific group. If not provided, keys will be the group name (if not empty) or the position of the group in *groups*, stored as a string. In the latter case, the first group in a population named *pop* will be retreived by either *pop[0]* or *pop['0']*.
- **parent** (*Graph*, optional (default: None)) – Parent if the population is created from an exiting graph.
- **syn_spec** (*dict, optional (default: static synapse)*) – Dictionary containg a directed edge between groups as key and the associated synaptic parameters for the post-synaptic neurons (i.e. those of the second group) as value. If a 'default' entry is provided, all unspecified connections will be set to its value.
- **meta_groups** (list or dict of str/*NeuralGroup* items) – Additional set of groups which can overlap: a neuron can belong to several different meta groups. Contrary to the primary groups, meta groups do therefore no need to be disjoint. If all meta-groups have a name, they can be passed directly through a list; otherwise a dict is necessary.
- **with_model** (*bool, optional (default: True)*) – Whether the groups require models (set to False to use populations for graph theoretical purposes, without NEST interaction)

Example

For synaptic properties, if provided in *syn_spec*, all connections between groups will be set according to the values. Keys can be either group names or types (1 for excitatory, -1 for inhibitory). Because of this, several combination can be available for the connections between two groups. Because of this, priority is given to source (presynaptic properties), i.e. NNGT will look for the entry matching the first group name as source before looking for entries matching the second group name as target.

```
# we created groups `g1`, `g2`, and `g3`
prop = {
    ('g1', 'g2'): {'model': 'tsodyks2_synapse', 'tau_fac': 50.},
    ('g1', g3'): {'weight': 100.},
    ...
}
pop = NeuronalPop.from_groups(
    [g1, g2, g3], names=['g1', 'g2', 'g3'], syn_spec=prop)
```

Note: If the population is not generated from an existing *Graph* and the groups do not contain explicit ids, then the ids will be generated upon population creation: the first group, of size N0, will be associated the indices 0 to N0 - 1, the second group (size N1), will get N0 to N0 + N1 - 1, etc.

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_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.

property has_models
Whether all groups have been assigned a neuronal model.

property inhibitory
Return the ids of all inhibitory nodes inside the population.

property nest_gids
Return the NEST gids of the nodes inside the population.

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_neuron_param`(*params*, *neurons=None*, *group=None*)

Set the parameters of specific neurons or of a whole group.

New in version 1.0.

Parameters

- **params** (*dict*) – Dictionary containing parameters for the neurons. Entries can be either a single number (same for all neurons) or a list (one entry per neuron).
- **neurons** (*list of ints, optional (default: None)*) – Ids of the neurons whose parameters should be modified.
- **group** (*list of strings, optional (default: None)*) – List of strings containing the names of the groups whose parameters should be updated. When modifying neurons from a single group, it is still usefull to specify the group name to speed up the pace.

Note: If both *neurons* and *group* are None, all neurons will be modified.

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.

`property syn_spec`

The properties of the synaptic connections between groups. Returns a `dict` containing tuples as keys and dicts of parameters as values.

The keys are tuples containing the names of the groups in the population, with the projecting group first (presynaptic neurons) and the receiving group last (post-synaptic neurons).

Example

For a population of excitatory ("exc") and inhibitory ("inh") neurons.

```
syn_spec = {
    ("exc", "exc"): {'model': 'stdp_synapse', 'weight': 2.5},
    ("exc", "inh"): {'model': 'static_synapse'},
    ("exc", "inh"): {'model': 'stdp_synapse', 'delay': 5.},
    ("inh", "inh"): {
        'model': 'stdp_synapse', 'weight': 5.,
        'delay': ('normal', 5., 2.)}
```

(continues on next page)

(continued from previous page)

```
    }
}
```

classmethod uniform(*size*, *neuron_type=1*, *neuron_model='aeif_cond_alpha'*, *neuron_param=None*, *syn_model='static_synapse'*, *syn_param=None*, *parent=None*, *meta_groups=None*)

Make a NeuralPop of identical neurons belonging to a single "default" group.

Parameters

- **size** (*int*) – Number of neurons in the population.
- **neuron_type** (*int, optional (default: 1)*) – Type of the neurons in the population: 1 for excitatory or -1 for inhibitory.
- **neuron_model** (*str, optional (default: default neuron model)*) – Neuronal model for the simulator.
- **neuron_param** (*dict, optional (default: default neuron parameters)*) – Parameters associated to *neuron_model*.
- **syn_model** (*str, optional (default: default static synapse)*) – Synapse model for the simulator.
- **syn_param** (*dict, optional (default: default synaptic parameters)*) – Parameters associated to *syn_model*.
- **parent** (*Graph* object, optional (default: None)) – Parent graph described by the population.
- **meta_groups** (list or dict of str/*NeuralGroup* items) – Set of groups which can overlap: a neuron can belong to several different meta groups, i.e. they do no need to be disjoint. If all meta-groups have a name, they can be passed directly through a list; otherwise a dict is necessary.

class nngt.**Structure**(*size=None*, *parent=None*, *meta_groups=None*, ***kwargs*)

The basic class that contains groups of nodes and their properties.

Variables

- **ids** – lst, Returns the ids of nodes in the structure.
- **is_valid** – bool, Whether the structure is consistent with its associated network.
- **parent** – *Network*, Parent network.
- **size** – int, Returns the number of nodes in the structure.

Initialize Structure instance.

Parameters

- **size** (*int, optional (default: 0)*) – Number of nodes that the structure will contain.
- **parent** (*Network*, optional (default: None)) – Network associated to this structure.
- **meta_groups** (dict of str/*Group* items) – Optional set of groups. Contrary to the primary groups which define the structure and must be disjoint, meta groups can overlap: a neuron can belong to several different meta groups.
- ****kwargs** (dict)

Returns **struct** (*Structure* object.)

add_meta_group(*group*, *name=None*, *replace=False*)
Add an existing meta group to the structure.

Parameters

- **group** (*Group*) – Meta group.
- **name** (*str, optional (default: group name)*) – Name of the meta group.
- **replace** (*bool, optional (default: False)*) – Whether to override previous exiting meta group with same name.

Note: The name of the group is automatically updated to match the *name* argument.

add_to_group(*group_name*, *ids*)
Add nodes to a specific group.

Parameters

- **group_name** (*str or int*) – Name or index of the group.
- **ids** (*list or 1D-array*) – Node ids.

copy()
Return a deep copy of the structure.

create_group(*nodes*, *name*, *properties=None*, *replace=False*)
Create a new group in the structure.

Parameters

- **nodes** (*int or array-like*) – Desired number of nodes or list of the nodes indices.
- **name** (*str*) – Name of the group.
- **properties** (*dict, optional (default: None)*) – Properties associated to the nodes in this group.
- **replace** (*bool, optional (default: False)*) – Whether to override previous exiting meta group with same name.

create_meta_group(*nodes*, *name*, *properties=None*, *replace=False*)
Create a new meta group and add it to the structure.

Parameters

- **nodes** (*int or array-like*) – Desired number of nodes or list of the nodes indices.
- **name** (*str*) – Name of the group.
- **properties** (*dict, optional (default: None)*) – Properties associated to the nodes in this group.
- **replace** (*bool, optional (default: False)*) – Whether to override previous exiting meta group with same name.

classmethod from_groups(*groups*, *names=None*, *parent=None*, *meta_groups=None*)
Make a *Structure* object from a (list of) *Group* object(s).

Parameters

- **groups** (dict or list of *Group* objects) – Groups that will be used to form the structure. Note that a given node can only belong to a single group, so the groups should form pairwise disjoints complementary sets.

- **names** (*list of str, optional (default: None)*) – Names that can be used as keys to retreive a specific group. If not provided, keys will be the group name (if not empty) or the position of the group in *groups*, stored as a string. In the latter case, the first group in a structure named *struct* will be retreived by either *struct[0]* or *struct['0']*.
- **parent** (*Graph*, optional (default: None)) – Parent if the structure is created from an exiting graph.
- **meta_groups** (list or dict of str/*Group* items) – Additional set of groups which can overlap: a node can belong to several different meta groups. Contrary to the primary groups, meta groups do therefore no need to be disjoint. If all meta-groups have a name, they can be passed directly through a list; otherwise a dict is necessary.

Example

For synaptic properties, if provided in *syn_spec*, all connections between groups will be set according to the values. Keys can be either group names or types (1 for excitatory, -1 for inhibitory). Because of this, several combination can be available for the connections between two groups. Because of this, priority is given to source (presynaptic properties), i.e. NNGT will look for the entry matching the first group name as source before looking for entries matching the second group name as target.

```
# we already created groups `g1`, `g2`, and `g3`
struct = Structure.from_groups([g1, g2, g3],
                               names=['g1', 'g2', 'g3'])
```

Note: If the structure is not generated from an existing *Graph* and the groups do not contain explicit ids, then the ids will be generated upon structure creation: the first group, of size N0, will be associated the indices 0 to N0 - 1, the second group (size N1), will get N0 to N0 + N1 - 1, etc.

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

Parameters

- **nodes** (*int or array-like*) – IDs of the nodes 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_properties(*key=None*, *groups=None*, *nodes=None*)
Return the properties of nodes or groups of nodes in the structure.

Parameters

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

Returns **props** (`list`) – List of all dictionaries with properties.

property ids
Return all the ids of the nodes inside the structure.

property is_valid
Whether the structure is consistent with the associated network.

property parent
Parent *Network*, if it exists, otherwise None.

set_properties(*props*, *nodes=None*, *group=None*)
Set the parameters of specific nodes or of a whole group.

New in version 2.2.

Parameters

- **props** (*dict*) – Dictionary containing parameters for the nodes. Entries can be either a single number (same for all nodes) or a list (one entry per nodes).
- **nodes** (*list of ints, optional (default: None)*) – Ids of the nodes whose parameters should be modified.
- **group** (*list of strings, optional (default: None)*) – List of strings containing the names of the groups whose parameters should be updated. When modifying nodes from a single group, it is still usefull to specify the group name to speed up the pace.

Note: If both *nodes* and *group* are None, all nodes will be modified.

property size
Number of nodes in this structure.

NNGT

Package aimed at facilitating the analysis of Neural Networks and Graphs' Topologies in Python by providing a unified interface for network generation and analysis.

The library mainly provides algorithms for

1. generating networks
2. studying their topological properties
3. doing some basic spatial, topological, and statistical visualizations
4. interacting with neuronal simulators and analyzing neuronal activity

Available modules

analysis Tools to study graph topology and neuronal activity.

core Where the main classes are coded; however, most useful classes and methods for users are loaded at the main level (*nngt*) when the library is imported, so *nngt.core* should generally not be used.

generation Functions to generate specific networks.

geometry Tools to work on metric graphs (see PyNCulture).

io Tools for input/output operations.

lib Basic functions used by several most other modules.

simulation Tools to provide complex network generation with NEST and help analyze the influence of the network structure on neuronal activity.

plot Plot data or graphs using matplotlib.

Units

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

- *mm* for milimeters
- *cm* centimeters
- *dm* for decimeters
- *m* for meters

Main classes and functions

nngt.Graph(*args, **kwargs)	The basic graph class, which inherits from a library class such as `graph_tool.Graph`, `networkx.DiGraph`, or `igraph.Graph`.
nngt.Group([nodes, properties, name])	Class defining groups of nodes.
nngt.GroupProperty(size[, constraints, ...])	Class defining the properties needed to create groups of neurons from an existing *Graph* or one of its subclasses.
nngt.MetaGroup([nodes, properties, name])	Class defining a meta-group of nodes.
nngt.MetaNeuralGroup([nodes, neuron_type, ...])	Class defining a meta-group of neurons.
nngt.Network(*args, **kwargs)	The detailed class that inherits from *Graph* and implements additional properties to describe various biological functions and interact with the NEST simulator.
nngt.NeuralGroup([nodes, neuron_type, ...])	Class defining groups of neurons.
nngt.NeuralPop([size, parent, meta_groups, ...])	The basic class that contains groups of neurons and their properties.
nngt.SpatialGraph(*args, **kwargs)	The detailed class that inherits from *Graph* and implements additional properties to describe spatial graphs (i.e.
nngt.SpatialNetwork(*args, **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.
nngt.Structure([size, parent, meta_groups])	The basic class that contains groups of nodes and their properties.
nngt.generate(di_instructions, **kwargs)	Generate a *Graph* or one of its subclasses from a `dict` containing all the relevant informations.
nngt.get_config([key, detailed])	Get the NNGT configuration as a dictionary.
nngt.load_from_file(filename[, fmt, ...])	Load a Graph from a file.
nngt.num_mpi_processes()	Returns the number of MPI processes (1 if MPI is not used)
nngt.on_master_process()	Check whether the current code is executing on the master process (rank 0) if MPI is used.
nngt.save_to_file(graph, filename[, fmt, ...])	Save a graph to file.
nngt.seed([msd, seeds])	Seed the random generator used by NNGT (i.e.
nngt.set_config(config[, value, silent])	Set NNGT's configuration.
nngt.use_backend(backend[, reloading, silent])	Allows the user to switch to a specific graph library as backend.

Details

class nngt.Graph(*args, **kwargs)

The basic graph class, which inherits from a library class such as graph_tool.Graph, networkx.DiGraph, or igraph.Graph.

The objects provides several functions to easily access some basic properties.

Initialize Graph instance

Changed in version 2.0: Renamed *from_graph* to *copy_graph*.

Changed in version 2.2: Added *structure* 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.
- **copy_graph** (*Graph*, optional) – An optional *Graph* that will be copied.
- **structure** (*Structure*, optional (default: None)) – A structure dividing the graph into specific groups, which can be used to generate specific connectivities and visualise the connections in a more coarse-grained manner.
- **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.

Note: When using *copy_graph*, only the topological properties are copied (nodes, edges, and attributes), spatial and biological properties are ignored. To copy a graph exactly, use *copy()*.

Returns **self** (*Graph*)

class nngt.Group(*nodes=None*, *properties=None*, *name=None*, **kwargs*)

Class defining groups of nodes.

Its main variables are:

Variables

- **ids** – list of int the ids of the nodes in this group.
- **properties** – dict, optional (default: {}) properties associated to the nodes
- **is_metagroup** – bool whether the group is a meta-group or not.

Note: A *Group* contains a set of nodes that are unique; the size of the group is the number of unique nodes contained in the group. Passing non-unique nodes will automatically convert them to a unique set.

Warning: Equality between `Group`s only compares the size and ``properties` attributes. This means that groups differing only by their `ids` will register as equal.

Calling the class creates a group of nodes. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the nodes in an existing graph.
- **properties** (*dict, optional (default: {})*) – Dictionary containing the properties associated to the nodes.

Returns A new *Group* instance.

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 *Graph* 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** – str, 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.**MetaGroup**(*nodes=None*, *properties=None*, *name=None*, ***kwargs*)

Class defining a meta-group of nodes.

Its main variables are:

Variables **ids** – list of int the ids of the nodes in this group.

Calling the class creates a group of nodes. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the nodes in an existing graph.
- **name** (*str, optional (default: "Group N")*) – Name of the meta-group.

Returns A new `MetaGroup` object.

class nngt.**MetaNeuralGroup**(*nodes=None*, *neuron_type='undefined'*, *neuron_model=None*, *neuron_param=None*, *name=None*, ***kwargs*)

Class defining a meta-group of neurons.

Its main variables are:

Variables

- **ids** – `list` of `int` the ids of the neurons in this group.
- **is_metagroup** – `bool` whether the group is a meta-group or not (*neuron_type* is `None` for meta-groups)

Calling the class creates a group of neurons. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the neurons in an existing graph.
- **name** (*str, optional (default: "Group N")*) – Name of the meta-group.

Returns A new `MetaNeuralGroup` object.

class nngt.**Network**(**args*, ***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.
- **copy_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.
- **inh_weight_factor** (*float, optional (default: 1.)*) – Factor to apply to inhibitory synapses, to compensate for example the strength difference due to timescales between excitatory and inhibitory synapses.

Returns **self** (`Network`)

class nngt.**NeuralGroup**(*nodes=None*, *neuron_type='undefined'*, *neuron_model=None*, *neuron_param=None*, *name=None*, ***kwargs*)

Class defining groups of neurons.

Its main variables are:

Variables

- **ids** – list of int the ids of the neurons in this group.
- **neuron_type** – int the default is 1 for excitatory neurons; -1 is for inhibitory neurons; meta-groups must have *neuron_type* set to None
- **neuron_model** – str, 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)
- **is_metagroup** – bool whether the group is a meta-group or not (*neuron_type* is None for meta-groups)

Warning: Equality between NeuralGroup`s only compares the size and neuronal type, ``model` and param attributes. This means that groups differing only by their ids will register as equal.

Calling the class creates a group of neurons. The default is an empty group but it is not a valid object for most use cases.

Parameters

- **nodes** (*int or array-like, optional (default: None)*) – Desired size of the group or, a posteriori, NNGT indices of the neurons in an existing graph.
- **neuron_type** (*int, optional (default: 1)*) – Type of the neurons (1 for excitatory, -1 for inhibitory) or None if not relevant (only allowed for metag roups).
- **neuron_model** (*str, optional (default: None)*) – NEST model for the neuron.
- **neuron_param** (*dict, optional (default: model defaults)*) – Dictionary containing the parameters associated to the NEST model.

Returns A new NeuralGroup instance.

class nngt.**NeuralPop**(*size=None*, *parent=None*, *meta_groups=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.
- **size** – int, Returns the number of neurons in the population.
- **syn_spec** – dict, Dictionary containing informations about the synapses between the different groups in the population.
- **is_valid** – bool, Whether this population can be used to create a network in NEST.

Initialize NeuralPop instance.

Parameters

- **size** (*int, optional (default: 0)*) – Number of neurons that the population will contain.
- **parent** (*Network*, optional (default: None)) – Network associated to this population.
- **meta_groups** (dict of str/*NeuralGroup* items) – Optional set of groups. Contrary to the primary groups which define the population and must be disjoint, meta groups can overlap: a neuron can belong to several different meta groups.
- **with_models** (bool) – whether the population's groups contain models to use in NEST

- ***args** (*items for OrderedDict parent*)
- ****kwargs** (dict)

Returns **pop** (*NeuralPop* object.)

class nngt.SpatialGraph(**args*, ***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.

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.
- ****kwargs** (keyword arguments for *Graph* or) – *Shape* if no shape was given.

Returns **self** (SpatialGraph)

class nngt.SpatialNetwork(**args*, ***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 SpatialNetwork 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)) – Population from which the network will be built.

Returns **self** (*SpatialNetwork*)

class nngt.Structure(*size=None*, *parent=None*, *meta_groups=None*, ***kwargs*)

The basic class that contains groups of nodes and their properties.

Variables

- **ids** – lst, Returns the ids of nodes in the structure.
- **is_valid** – bool, Whether the structure is consistent with its associated network.

- **parent** – *Network*, Parent network.
- **size** – int, Returns the number of nodes in the structure.

Initialize Structure instance.

Parameters

- **size** (*int, optional (default: 0)*) – Number of nodes that the structure will contain.
- **parent** (*Network*, optional (default: None)) – Network associated to this structure.
- **meta_groups** (dict of str/*Group* items) – Optional set of groups. Contrary to the primary groups which define the structure and must be disjoint, meta groups can overlap: a neuron can belong to several different meta groups.
- ****kwargs** (dict)

Returns **struct** (*Structure* object.)

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

Generate a *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.

See also:

generation

nngt.**get_config**(*key=None*, *detailed=False*)

Get the NNGT configuration as a dictionary.

Note: This function has no MPI barrier on it.

---

nngt.**load_from_file**(*filename*, *fmt='auto'*, *separator=' '*, *secondary=';'*, *attributes=None*, *attributes_types=None*, *notifier='@'*, *ignore='#'*, *name='LoadedGraph'*, *directed=True*, *cleanup=False*)

Load a Graph from a file.

Changed in version 2.0: Added optional *attributes_types* and *cleanup* arguments.

Warning: Support for GraphML and DOT formats are currently limited and require one of the non-default backends (DOT requires graph-tool).

Parameters

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

- **separator** (*str, optional (default " ")*) – separator used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")
- **secondary** (*str, optional (default: ";")*) – Secondary separator 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. For "edge_list", attributes may also be present as additional columns after the source and the target.
- **attributes_types** (*dict, optional (default: str)*) – Backup information if the type of the attributes is not specified in the file. Values must be callables (types or functions) that will take the argument value as a string input and convert it to the proper type.
- **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` are of type (`list`, `bool`, `str`, `int`). Additional notifiers are `@type=SpatialGraph/Network/SpatialNetwork`, which must be followed by the relevant notifiers among `@shape`, `@structure`, and `@graph`.
- **ignore** (*str, optional (default: "#")*) – Ignore lines starting with the *ignore* string.
- **name** (*str, optional (default: from file information or 'LoadedGraph')*) – The name of the graph.
- **directed** (*bool, optional (default: from file information or True)*) – Whether the graph is directed or not.
- **cleanup** (*bool, optional (default: False)*) – If true, removes nodes before the first one that appears in the edges and after the last one and renumber the nodes from 0.

Returns **graph** (*Graph* or subclass) – Loaded graph.

nngt.**num_mpi_processes**()
Returns the number of MPI processes (1 if MPI is not used)

nngt.**on_master_process**()
Check whether the current code is executing on the master process (rank 0) if MPI is used.

Returns

- *True if rank is 0, if mpi4py is not present or if MPI is not used,*
- *otherwise False.*

nngt.**save_to_file**(*graph*, *filename*, *fmt='auto'*, *separator=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*)
Save a graph to file.

@todo: implement dot, xml/graphml, and gt formats

Parameters

- **graph** (*Graph* or subclass) – Graph to save.
- **filename** (*str*) – The path to the file.
- **fmt** (*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 as library, detected if *filename* ends with '.gt').

- **separator** (*str, optional (default " "))* – separator used to separate inputs in the case of custom formats (namely "neighbour" and "edge_list")
- **secondary** (*str, optional (default: ";"))* – Secondary separator used to separate attributes in the case of custom formats.
- **attributes** (list, optional (default: `None`)) – List of names for the edge attributes present in the graph that will be saved to disk; by default (`None`), all attributes will be saved.
- **notifier** (*str, optional (default: "@"))* – Symbol specifying the following as meaningfull information. Relevant information are formatted `@info_name=info_value`, with `info_name` in ("attributes", "attr_types", "directed", "name", "size"). Additional notifiers are `@type=SpatialGraph/Network/SpatialNetwork`, which are followed by the relevant notifiers among `@shape`, `@structure`, and `@graph` to separate the sections.

Note: Positions are saved as bytes by `numpy.nparray.tostring()`

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

Parameters

- **msd** (*int, optional*) – Master seed for numpy *RandomState*. Must be convertible to 32-bit unsigned integers.
- **seeds** (*list of ints, optional*) – Seeds for *RandomState* (when using MPI). Must be convertible to 32-bit unsigned integers, one entry per MPI process.

nngt.**set_config**(*config*, *value=None*, *silent=False*)
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)
```

Notes

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

This function has an MPI barrier on it, so it must always be called on all processes.

See also:

get_config()

nngt.**use_backend**(*backend*, *reloading=True*, *silent=False*)
Allows the user to switch to a specific graph library as backend.

Warning: If *Graph* objects have already been created, they will no longer be compatible with NNGT methods.

Parameters

- **backend** (*string*) – Name of a graph library among 'graph_tool', 'igraph', 'networkx', or 'nngt'.
- **reloading** (*bool, optional (default: True)*) – Whether the graph objects should be reloaded through *reload* (this should always be set to True except when NNGT is first initiated!)
- **silent** (*bool, optional (default: False)*) – Whether the changes made to the configuration should be logged at the DEBUG (True) or INFO (False) level.

Analysis module

Tools to analyze neuronal networks, using either their topological properties, their activity, or more importantly, taking both into account. See also *Consistent tools for graph analysis* for more details on consistent analysis across graph libraries.

Content

nngt.analysis.adjacency_matrix(graph[, ...])	Adjacency matrix of the graph.
nngt.analysis.all_shortest_paths(g, source, ...)	Yields all shortest paths from *source* to *target*.
nngt.analysis.assortativity(g, degree[, weights])	Returns the assortativity of the graph.
nngt.analysis.average_path_length(g[, ...])	Returns the average shortest path length between *sources* and *targets*.
nngt.analysis.bayesian_blocks(t[, x, sigma, ...])	Bayesian Blocks Implementation
nngt.analysis.betweenness(g[, btype, weights])	Returns the normalized betweenness centrality of the nodes and edges.
nngt.analysis.betweenness_distrib(graph[, ...])	Betweenness distribution of a graph.
nngt.analysis.binning(x[, bins, log])	Binning function providing automatic binning using Bayesian blocks in addition to standard linear and logarithmic uniform bins.
nngt.analysis.closeness(g[, weights, nodes, ...])	Returns the closeness centrality of some *nodes*.
nngt.analysis.connected_components(g[, ctype])	Returns the connected component to which each node belongs.
nngt.analysis.degree_distrib(graph[, ...])	Degree distribution of a graph.

continues on next page

Table 14 – continued from previous page

nngt.analysis.diameter(g[, directed, ...])	Returns the diameter of the graph.
nngt.analysis.get_b2([network, ...])	Return the B2 coefficient for the neurons.
nngt.analysis.get_firing_rate([network, ...])	Return the average firing rate for the neurons.
nngt.analysis.get_spikes([recorder, ...])	Return a 2D sparse matrix, where:
nngt.analysis.global_clustering(g[, ...])	Returns the global clustering coefficient.
nngt.analysis.global_clustering_binary_undirected(g)	Return the undirected global clustering coefficient.
nngt.analysis.local_closure(g[, directed, ...])	Compute the local closure for each node, as defined in [Yin2019] as the fraction of 2-walks that are closed.
nngt.analysis.local_clustering(g[, nodes, ...])	Local (weighted directed) clustering coefficient of the nodes, ignoring self-loops.
nngt.analysis.local_clustering_binary_undirected(g)	Return the undirected local clustering coefficient of some *nodes*.
nngt.analysis.node_attributes(network, ...)	Return node *attributes* for a set of *nodes*.
nngt.analysis.num_iedges(graph)	Returns the number of inhibitory connections.
nngt.analysis.reciprocity(g)	Calculate the edge reciprocity of the graph.
nngt.analysis.shortest_distance(g[, ...])	Returns the length of the shortest paths between *sources`and `targets*.
nngt.analysis.shortest_path(g, source, target)	Returns a shortest path between *source`and `target*.
nngt.analysis.small_world_propensity(g[, ...])	Returns the small-world propensity of the graph as first defined in [Muldoon2016].
nngt.analysis.spectral_radius(graph[, ...])	Spectral radius of the graph, defined as the eigenvalue of greatest module.
nngt.analysis.subgraph_centrality(graph[, ...])	Returns the subgraph centrality for each node in the graph.
nngt.analysis.total_firing_rate([network, ...])	Computes the total firing rate of the network from the spike times.
nngt.analysis.transitivity(g[, directed, ...])	Same as *global_clustering()*.
nngt.analysis.triangle_count(g[, nodes, ...])	Returns the number or the strength (also called intensity) of triangles for each node.
nngt.analysis.triplet_count(g[, nodes, ...])	Returns the number or the strength (also called intensity) of triplets for each node.

Details

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

Adjacency matrix of the graph.

Parameters

- **graph** (*Graph* or subclass) – Network to analyze.
- **types** (*bool, optional (default: False)*) – 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: False)*) – 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.

References

nngt.analysis.**all_shortest_paths**(*g*, *source*, *target*, *directed=None*, *weights=None*, *combine_weights='mean'*)

Yields all shortest paths from *source* to *target*. The algorithms returns an empty generator if there is no path between the nodes.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **source** (*int*) – Node from which the paths starts.
- **target** (*int, optional (default: all nodes)*) – Node where the paths ends.
- **directed** (bool, optional (default: `g.is_directed()`)) – Whether the edges should be considered as directed or not (automatically set to False if *g* is undirected).
- **weights** (*str or array, optional (default: binary)*) – Whether to use weighted edges to compute the distances. By default, all edges are considered to have distance 1.
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.

Returns **all_paths** (*generator*) – Generator yielding paths as lists of ints.

References

nngt.analysis.**assortativity**(*g*, *degree*, *weights=None*)

Returns the assortativity of the graph. This tells whether nodes are preferentially connected together depending on their degree.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **degree** (*str*) – The type of degree that should be considered.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.

References

nngt.analysis.**average_path_length**(*g*, *sources=None*, *targets=None*, *directed=None*, *weights=None*, *combine_weights='mean'*, *unconnected=False*)

Returns the average shortest path length between *sources* and *targets*. The algorithms raises an error if all nodes are not connected unless *unconnected* is set to True.

The average path length is defined as

$$L = \frac{1}{N_p} \sum_{u,v} d(u, v),$$

where N_p is the number of paths between *sources* and *targets*, and $d(u, v)$ is the shortest path distance from u to v.

If *sources* and *targets* are both None, then the total number of paths is $N_p = N(N-1)$, with N the number of nodes in the graph.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **sources** (*list of nodes, optional (default: all)*) – Nodes from which the paths must be computed.
- **targets** (*list of nodes, optional (default: all)*) – Nodes to which the paths must be computed.
- **directed** (bool, optional (default: `g.is_directed()`)) – Whether the edges should be considered as directed or not (automatically set to False if *g* is undirected).
- **weights** (*str or array, optional (default: binary)*) – Whether to use weighted edges to compute the distances. By default, all edges are considered to have distance 1.
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.
- **unconnected** (*bool, optional (default: False)*) – If set to true, ignores unconnected nodes and returns the average path length of the existing paths.

References

nngt.analysis.**bayesian_blocks**(*t*, *x=None*, *sigma=None*, *fitness='events'*, ***kwargs*)
Bayesian Blocks Implementation

This is a flexible implementation of the Bayesian Blocks algorithm described in Scargle 2012[1]

New in version 0.7.

Parameters

- **t** (*array_like*) – data times (one dimensional, length N)
- **x** (*array_like (optional)*) – data values
- **sigma** (*array_like or float (optional)*) – data errors
- **fitness** (*str or object*) – the fitness function to use. If a string, the following options are supported:
  - **'events'** [binned or unbinned event data] extra arguments are *p0*, which gives the false alarm probability to compute the prior, or *gamma* which gives the slope of the prior on the number of bins.
  - **'regular_events'** [non-overlapping events measured at multiples] of a fundamental tick rate, *dt*, which must be specified as an additional argument. The prior can be specified through *gamma*, which gives the slope of the prior on the number of bins.

[1] Scargle, J *et al.* (2012) http://adsabs.harvard.edu/abs/2012arXiv1207.5578S

– **'measures'** [fitness for a measured sequence with Gaussian errors] The prior can be specified using *gamma*, which gives the slope of the prior on the number of bins. If *gamma* is not specified, then a simulation-derived prior will be used.

Alternatively, the fitness can be a user-specified object of type derived from the FitnessFunc class.

Returns **edges** (*ndarray*) – array containing the (N+1) bin edges

Examples

Event data:

```
>>> t = np.random.normal(size=100)
>>> bins = bayesian_blocks(t, fitness='events', p0=0.01)
```

Event data with repeats:

```
>>> t = np.random.normal(size=100)
>>> t[80:] = t[:20]
>>> bins = bayesian_blocks(t, fitness='events', p0=0.01)
```

Regular event data:

```
>>> dt = 0.01
>>> t = dt * np.arange(1000)
>>> x = np.zeros(len(t))
>>> x[np.random.randint(0, len(t), len(t) / 10)] = 1
>>> bins = bayesian_blocks(t, fitness='regular_events', dt=dt, gamma=0.9)
```

Measured point data with errors:

```
>>> t = 100 * np.random.random(100)
>>> x = np.exp(-0.5 * (t - 50) ** 2)
>>> sigma = 0.1
>>> x_obs = np.random.normal(x, sigma)
>>> bins = bayesian_blocks(t, fitness='measures')
```

References

See also:

astroML.plotting.hist() histogram plotting function which can make use of bayesian blocks.

nngt.analysis.**betweenness**(*g*, *btype='both'*, *weights=None*)

Returns the normalized betweenness centrality of the nodes and edges.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **btype** (*str, optional (default 'both')*) – The centrality that should be returned (either 'node', 'edge', or 'both'). By default, both betweenness centralities are computed.

- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.

Returns

- **nb** (`numpy.ndarray`) – The nodes' betweenness if *btype* is 'node' or 'both'
- **eb** (`numpy.ndarray`) – The edges' betweenness if *btype* is 'edge' or 'both'

References

nngt.analysis.**betweenness_distrib**(*graph*, *weights=None*, *nodes=None*, *num_nbins='bayes'*, *num_ebins='bayes'*, *log=False*)

Betweenness distribution of a graph.

Parameters

- **graph** (*Graph* or subclass) – the graph to analyze.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **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.
- **num_bins** (*int, list or str, optional (default: 'bayes')*) – Any of the automatic methodes from `numpy.histogram()`, or 'bayes' will provide automatic bin optimization. Otherwise, an int for the number of bins can be provided, or the direct bins list.

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.**binning**(*x*, *bins='bayes'*, *log=False*)

Binning function providing automatic binning using Bayesian blocks in addition to standard linear and logarithmic uniform bins.

New in version 0.7.

Parameters

- **x** (*array-like*) – Array of data to be histogrammed
- **bins** (*int, list or 'auto', optional (default: 'bayes')*) – If *bins* is 'bayes', in use bayesian blocks for dynamic bin widths; if it is an int, the interval will be separated into
- **log** (*bool, optional (default: False)*) – Whether the bins should be evenly spaced on a logarithmic scale.

nngt.analysis.**closeness**(*g*, *weights=None*, *nodes=None*, *mode='out'*, *harmonic=True*, *default=nan*)

Returns the closeness centrality of some *nodes*.

Closeness centrality of a node *u* is defined, for the harmonic version, as the sum of the reciprocal of the shortest path distance d_{uv} from *u* to the N - 1 other nodes in the graph (if *mode* is "out", reciprocally d_{vu}, the distance to *u* from another node v, if *mode* is "in"):

$$C(u) = \frac{1}{N-1} \sum_{v \neq u} \frac{1}{d_{uv}},$$

or, using the arithmetic definition, as the reciprocal of the average shortest path distance to/from *u* over to all other nodes:

$$C(u) = \frac{n-1}{\sum_{v \neq u} d_{uv}},$$

where *d_{uv}* is the shortest-path distance from *u* to *v*, and *n* is the number of nodes in the component.

By definition, the distance is infinite when nodes are not connected by a path in the harmonic case (such that $\frac{1}{d(v,u)} = 0$), while the distance itself is taken as zero for unconnected nodes in the first equation.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **nodes** (*list, optional (default: all nodes)*) – The list of nodes for which the clutering will be returned
- **mode** (*str, optional (default: "out")*) – For directed graphs, whether the distances are computed from ("out") or to ("in") each of the nodes.
- **harmonic** (*bool, optional (default: True)*) – Whether the arithmetic or the harmonic (recommended) version of the closeness should be used.

Returns **c** (`numpy.ndarray`) – The list of closeness centralities, on per node.

References

nngt.analysis.**connected_components**(*g*, *ctype=None*)

Returns the connected component to which each node belongs.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **ctype** (*str, optional (default 'scc')*) – Type of component that will be searched: either strongly connected ('scc', by default) or weakly connected ('wcc').

Returns **cc, hist** (`numpy.ndarray`) – The component associated to each node (*cc*) and the number of nodes in each of the component (*hist*).

References

nngt.analysis.**degree_distrib**(*graph*, *deg_type='total'*, *nodes=None*, *weights=None*, *log=False*, *num_bins='bayes'*)

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").
- **nodes** (*list of ints, optional (default: None)*) – Restrict the distribution to a set of nodes (default: all nodes).
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **log** (*bool, optional (default: False)*) – use log-spaced bins.
- **num_bins** (*int, list or str, optional (default: 'bayes')*) – Any of the automatic methodes from `numpy.histogram()`, or 'bayes' will provide automatic bin optimization. Otherwise, an int for the number of bins can be provided, or the direct bins list.

See also:

`numpy.histogram()`, *binning()*

Returns

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

nngt.analysis.**diameter**(*g*, *directed=None*, *weights=None*, *combine_weights='mean'*, *is_connected=False*)

Returns the diameter of the graph.

Changed in version 2.3: Added *combine_weights* argument.

Changed in version 2.0: Added *directed* and *is_connected* arguments.

It returns infinity if the graph is not connected (strongly connected for directed graphs) unless *is_connected* is True, in which case it returns the longest existing shortest distance.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **directed** (bool, optional (default: `g.is_directed()`)) – Whether to compute the directed diameter if the graph is directed. If False, then the graph is treated as undirected. The option switches to False automatically if *g* is undirected.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.

  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.
- **is_connected** (*bool, optional (default: False)*) – If False, check whether the graph is connected or not and return infinite diameter if graph is unconnected. If True, the graph is assumed to be connected.

See also:

nngt.analysis.shortest_distance()

References

nngt.analysis.**get_b2**(*network=None*, *spike_detector=None*, *data=None*, *nodes=None*)
Return the B2 coefficient for the neurons.

Parameters

- **network** (*nngt.Network*, optional (default: None)) – Network for which the activity was simulated.
- **spike_detector** (*tuple of ints, optional (default: spike detectors)*) – GID of the "spike_detector" objects recording the network activity.
- **data** (*array-like of shape (N, 2), optionale (default: None)*) – Array containing the spikes data (first line must contain the NEST GID of the neuron that fired, second line must contain the associated spike time).
- **nodes** (*array-like, optional (default: all neurons)*) – NNGT ids of the nodes for which the B2 should be computed.

Returns **b2** (*array-like*) – B2 coefficient for each neuron in *nodes*.

nngt.analysis.**get_firing_rate**(*network=None*, *spike_detector=None*, *data=None*, *nodes=None*)
Return the average firing rate for the neurons.

Parameters

- **network** (*nngt.Network*, optional (default: None)) – Network for which the activity was simulated.
- **spike_detector** (*tuple of ints, optional (default: spike detectors)*) – GID of the "spike_detector" objects recording the network activity.
- **data** (`numpy.array` of shape (N, 2), optionale (default: None)) – Array containing the spikes data (first line must contain the NEST GID of the neuron that fired, second line must contain the associated spike time).
- **nodes** (*array-like, optional (default: all nodes)*) – NNGT ids of the nodes for which the B2 should be computed.

Returns **fr** (*array-like*) – Firing rate for each neuron in *nodes*.

nngt.analysis.**get_spikes**(*recorder=None*, *spike_times=None*, *senders=None*, *astype='ssp'*)
Return a 2D sparse matrix, where:

- each row i contains the spikes of neuron i (in NEST),
- each column j contains the times of the jth spike for all neurons.

Changed in version 1.0: Neurons are now located in the row corresponding to their NEST GID.

Parameters

- **recorder** (*tuple, optional (default: None)*) – Tuple of NEST gids, where the first one should point to the spike_detector which recorded the spikes.
- **spike_times** (*array-like, optional (default: None)*) – If *recorder* is not provided, the spikes' data can be passed directly through their *spike_times* and the associated *senders*.
- **senders** (*array-like, optional (default: None)*) – *senders[i]* corresponds to the neuron which fired at *spike_times[i]*.
- **astype** (*str, optional (default: "ssp")*) – Format of the returned data. Default is sparse lil_matrix ("ssp") with one row per neuron, otherwise "np" returns a (T, 2) array, with T the number of spikes (the first row being the NEST gid, the second the spike time).

Example

```
>>> get_spikes()
```

```
>>> get_spikes(recorder)
```

```
>>> times = [1.5, 2.68, 125.6]
>>> neuron_ids = [12, 0, 65]
>>> get_spikes(spike_times=times, senders=neuron_ids)
```

Note: If no arguments are passed to the function, the first spike_recorder available in NEST will be used. Neuron positions correspond to their GIDs in NEST.

Returns

- *CSR matrix containing the spikes sorted by neuron GIDs (rows) and time*
- *(columns).*

nngt.analysis.**global_clustering**(*g*, *directed=True*, *weights=None*, *method='continuous'*, *mode='total'*, *combine_weights='mean'*)

Returns the global clustering coefficient.

This corresponds to the ratio of triangles to the number of triplets. For directed and weighted cases, see definitions of generalized triangles and triplets in the associated functions below.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **directed** (*bool, optional (default: True)*) – Whether to compute the directed clustering if the graph is directed.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **method** (*str, optional (default: 'continuous')*) – Method used to compute the weighted clustering, either 'barrat' [Barrat2004], 'continuous' [Fardet2021], 'onnela' [Onnela2005], or 'zhang' [Zhang2005].
- **mode** (*str, optional (default: "total")*) – Type of clustering to use for directed graphs, among "total", "fan-in", "fan-out", "middleman", and "cycle" [Fagiolo2007].

- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.

References

See also:

triplet_count(), *triangle_count()*

nngt.analysis.**global_clustering_binary_undirected**(*g*)
Returns the undirected global clustering coefficient.

This corresponds to the ratio of undirected triangles to the number of undirected triads.

Parameters **g** (*Graph*) – Graph to analyze.

References

nngt.analysis.**local_closure**(*g*, *directed=True*, *weights=None*, *method=None*, *mode='cycle-out'*, *combine_weights='mean'*)
Compute the local closure for each node, as defined in [Yin2019] as the fraction of 2-walks that are closed.

For undirected binary or weighted adjacency matrices $W = \{w_{ij}\}$, the normal (or Zhang-like) definition is given by:

$$H_i^0 = \frac{\sum_{j\neq k} w_{ij}w_{jk}w_{ki}}{\sum_{j\neq k\neq i} w_{ij}w_{jk}} = \frac{W_{ii}^3}{\sum_{j\neq i} W_{ij}^2}$$

While a continuous version of the local closure is also proposed as:

$$H_i = \frac{\sum_{j\neq k} \sqrt[3]{w_{ij}w_{jk}w_{ki}}^2}{\sum_{j\neq k\neq i} \sqrt{w_{ij}w_{jk}}} = \frac{\left(W^{\left[\frac{2}{3}\right]}\right)^3_{ii}}{\sum_{j\neq i}\left(W^{\left[\frac{1}{2}\right]}\right)^2_{ij}}$$

with $W^{[\alpha]} = \{w_{ij}^\alpha\}$.

Directed versions of the local closure where defined as follow for a node i connected to nodes j and k:

- "cycle-out" is given by the pattern [(i, j), (j, k), (k, i)],
- "cycle-in" is given by the pattern [(k, j), (j, i), (i, k)],
- "fan-in" is given by the pattern [(k, j), (j, i), (k, i)],
- "fan-out" is given by the pattern [(i, j), (j, k), (i, k)].

See [Fardet2021] for more details.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **directed** (*bool, optional (default: True)*) – Whether to compute the directed clustering if the graph is directed.

- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **method** (*str, optional (default: 'continuous')*) – Method used to compute the weighted clustering, either 'normal'/'zhang' or 'continuous'.
- **mode** (*str, optional (default: "circle-out")*) – Type of clustering to use for directed graphs, among "circle-out", "circle-in", "fan-in", or "fan-out".
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.

References

nngt.analysis.**local_clustering**(*g*, *nodes=None*, *directed=True*, *weights=None*, *method='continuous'*, *mode='total'*, *combine_weights='mean'*)

Local (weighted directed) clustering coefficient of the nodes, ignoring self-loops.

If no weights are requested and the graph is undirected, returns the undirected binary clustering.

For all weighted cases, the weights are assumed to be positive and they are normalized to dimensionless values between 0 and 1 through a division by the highest weight.

The default *method* for weighted networks is the continuous definition [Fardet2021] and is defined as:

$$C_i = \frac{\sum_{jk} \sqrt[3]{w_{ij}w_{ik}w_{jk}}}{\sum_{j\neq k} \sqrt{w_{ij}w_{ik}}} = \frac{\left(W^{\left[\frac{2}{3}\right]}\right)^3_{ii}}{\left(s_i^{\left[\frac{1}{2}\right]}\right)^2 - s_i}$$

for undirected networks, with $W = \{w_{ij}\} = \tilde{W}/\max(\tilde{W})$ the normalized weight matrix, s_i the normalized strength of node i, and $s_i^{\left[\frac{1}{2}\right]} = \sum_k \sqrt{w_{ik}}$ the strength associated to the matrix $W^{\left[\frac{1}{2}\right]} = \{\sqrt{w_{ij}}\}$.

For directed networks, we used the total clustering defined in [Fagiolo2007] by default, hence the second equation becomes:

$$C_i = \frac{\frac{1}{2}\left(W^{\left[\frac{2}{3}\right]} + W^{\left[\frac{2}{3}\right],T}\right)^3_{ii}}{\left(s_i^{\left[\frac{1}{2}\right]}\right)^2 - 2s_i^{\leftrightarrow} - s_i}$$

with $s^{\leftrightarrow} = \sum_k \sqrt{w_{ik}w_{ki}}$ the reciprocal strength (associated to reciprocal connections).

For the other modes, see the generalized definitions in [Fagiolo2007].

Contrary to 'barrat' and 'onnela' [Saramaki2007], this method displays *all* following properties:

- fully continuous (no jump in clustering when weights go to zero),
- equivalent to binary clustering when all weights are 1,
- equivalence between no-edge and zero-weight edge cases,

- normalized (always between zero and 1).

Using either 'continuous' or 'zhang' is usually recommended for weighted graphs, see the discussion in [Fardet2021] for details.

Parameters

- **g** (*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.
- **directed** (*bool, optional (default: True)*) – Whether to compute the directed clustering if the graph is directed.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **method** (*str, optional (default: 'continuous')*) – Method used to compute the weighted clustering, either 'barrat' [Barrat2004]/[Clemente2018], 'continuous' [Fardet2021], 'onnela' [Onnela2005]/[Fagiolo2007], or 'zhang' [Zhang2005].
- **mode** (*str, optional (default: "total")*) – Type of clustering to use for directed graphs, among "total", "fan-in", "fan-out", "middleman", and "cycle" [Fagiolo2007].
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "sum": equivalent to mean due to weight normalization.

Returns **lc** (`numpy.ndarray`) – The list of clustering coefficients, on per node.

References

See also:

`undirected_binary_clustering()`, *global_clustering()*

nngt.analysis.**local_clustering_binary_undirected**(*g*, *nodes=None*)

Returns the undirected local clustering coefficient of some *nodes*.

If *g* is directed, then it is converted to a simple undirected graph (no parallel edges).

Parameters

- **g** (*Graph*) – Graph to analyze.
- **nodes** (*list, optional (default: all nodes)*) – The list of nodes for which the clustering will be returned

Returns **lc** (`numpy.ndarray`) – The list of clustering coefficients, on per node.

References

nngt.analysis.**node_attributes**(*network*, *attributes*, *nodes=None*, *data=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" * "closeness" * "in-degree", "out-degree", "total-degree" * "subgraph_centrality"
- **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.
- **data** (`numpy.array` of shape (N, 2), optional (default: None)) – Potential data on the spike events; if not None, it must contain the sender ids on the first column and the spike times on the second.

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.

For *Network* objects, this corresponds to the number of edges stemming from inhibitory nodes (given by *nngt.NeuralPop.inhibitory()*). Otherwise, counts the edges where the type attribute is -1.

nngt.analysis.**reciprocity**(*g*)
Calculate the edge reciprocity of the graph.

The reciprocity is defined as the number of edges that have a reciprocal edge (an edge between the same nodes but in the opposite direction) divided by the total number of edges. This is also the probability for any given edge, that its reciprocal edge exists. By definition, the reciprocity of undirected graphs is 1.

@todo: check whether we can get this for single nodes for all libraries.

Parameters **g** (*Graph*) – Graph to analyze.

References

nngt.analysis.**shortest_distance**(*g*, *sources=None*, *targets=None*, *directed=None*, *weights=None*, *combine_weights='mean'*)
Returns the length of the shortest paths between *sources`and `targets*. The algorithms return infinity if there are no paths between nodes.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **sources** (*list of nodes, optional (default: all)*) – Nodes from which the paths must be computed.
- **targets** (*list of nodes, optional (default: all)*) – Nodes to which the paths must be computed.
- **directed** (bool, optional (default: `g.is_directed()`)) – Whether the edges should be considered as directed or not (automatically set to False if *g* is undirected).
- **weights** (*str or array, optional (default: binary)*) – Whether to use weighted edges to compute the distances. By default, all edges are considered to have distance 1.
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:

- "sum": the sum of the edge attribute values will be used for the new edge.
- "mean": the mean of the edge attribute values will be used for the new edge.
- "min": the minimum of the edge attribute values will be used for the new edge.
- "max": the maximum of the edge attribute values will be used for the new edge.

Returns **distance** (*float, or 1d/2d numpy array of floats*) – Distance (if single source and single target) or distance array. For multiple sources and targets, the shape of the matrix is (S, T), with S the number of sources and T the number of targets; for a single source or target, return a 1d-array of length T or S.

References

nngt.analysis.**shortest_path**(*g*, *source*, *target*, *directed=None*, *weights=None*, *combine_weights='mean'*)

Returns a shortest path between *source`and `target*. The algorithms returns an empty list if there is no path between the nodes.

Parameters

- **g** (*Graph*) – Graph to analyze.
- **source** (*int*) – Node from which the path starts.
- **target** (*int*) – Node where the path ends.
- **directed** (bool, optional (default: `g.is_directed()`)) – Whether the edges should be considered as directed or not (automatically set to False if *g* is undirected).
- **weights** (*str or array, optional (default: binary)*) – Whether to use weighted edges to compute the distances. By default, all edges are considered to have distance 1.
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.

Returns **path** (*list of ints*) – Order of the nodes making up the path from *source* to *target*.

References

nngt.analysis.**small_world_propensity**(*g*, *directed=None*, *use_global_clustering=False*, *use_diameter=False*, *weights=None*, *combine_weights='mean'*, *clustering='continuous'*, *lattice=None*, *random=None*, *return_deviations=False*)

Returns the small-world propensity of the graph as first defined in [Muldoon2016].

$$\phi = 1 - \sqrt{\frac{\Pi_{[0,1]}(\Delta_C^2) + \Pi_{[0,1]}(\Delta_L^2)}{2}}$$

with Δ_C the clustering deviation, i.e. the relative global or average clustering of *g* compared to two reference graphs

$$\Delta_C = \frac{C_{latt} - C_g}{C_{latt} - C_{rand}}$$

and $Delta_L$ the deviation of the average path length or diameter, i.e. the relative average path length of g compared to that of the reference graphs

$$\Delta_L = \frac{L_g - L_{rand}}{L_{latt} - L_{rand}}.$$

In both cases, *latt* and *rand* refer to the equivalent lattice and Erdos-Renyi (ER) graphs obtained by rewiring g to obtain respectively the highest and lowest combination of clustering and average path length.

Both deviations are clipped to the [0, 1] range in case some graphs have a higher clustering than the lattice or a lower average path length than the ER graph.

Parameters

- **g** (*Graph* object) – Graph to analyze.
- **directed** (*bool, optional (default: True)*) – Whether to compute the directed clustering if the graph is directed. If False, then the graph is treated as undirected. The option switches to False automatically if *g* is undirected.
- **use_global_clustering** (*bool, optional (default: True)*) – If False, then the average local clustering is used instead of the global clustering.
- **use_diameter** (*bool, optional (default: False)*) – Use the diameter instead of the average path length to have more global information. Ccan also be much faster in some cases, especially using graph-tool as the backend.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.
- **clustering** (*str, optional (default: 'continuous')*) – Method used to compute the weighted clustering coefficients, either 'barrat' [Barrat2004], 'continuous' (recommended), or 'onnela' [Onnela2005].
- **lattice** (*nngt.Graph*, optional (default: generated from *g*)) – Lattice to use as reference (since its generation is deterministic, enables to avoid multiple generations when running the algorithm several times with the same graph)
- **random** (*nngt.Graph*, optional (default: generated from *g*)) – Random graph to use as reference. Can be useful for reproducibility or for very sparse graphs where ER algorithm would statistically lead to a disconnected graph.
- **return_deviations** (*bool, optional (default: False)*) – If True, the deviations are also returned, in addition to the small-world propensity.

Note: If *weights* are provided, the distance calculation uses the inverse of the weights. This implementation differs slightly from the original implementation as it can also use the global instead of the average clustering coefficient, the diameter instead of the avreage path length, and it is generalized to directed networks.

References

Returns

- **phi** (*float in [0, 1]*) – The small-world propensity.
- **delta_l** (*float*) – The average path-length deviation (if *return_deviations* is True).
- **delta_c** (*float*) – The clustering deviation (if *return_deviations* is True).

See also:

nngt.analysis.average_path_length(), *nngt.analysis.diameter()*, *nngt.analysis.global_clustering()*, *nngt.analysis.local_clustering()*, *nngt.generation.lattice_rewire()*, *nngt.generation.random_rewire()*

nngt.analysis.**spectral_radius**(*graph*, *typed=True*, *weights=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.
- **weights** (*bool, optional (default: True)*) – Whether weights should be taken into account, defaults to the "weight" edge attribute if present.

Returns *the spectral radius as a float.*

nngt.analysis.**subgraph_centrality**(*graph*, *weights=True*, *nodes=None*, *normalize='max_centrality'*)
Returns the subgraph centrality for each node in the graph.

Defined according to [Estrada2005] as:

$$sc(i) = e^W_{ii}$$

where W is the (potentially weighted and normalized) adjacency matrix.

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).
- **nodes** (*array-like container with node ids, optional (default = all nodes)*) – Nodes for which the subgraph centrality should be returned (all centralities are computed anyway in the algorithm).
- **normalize** (*str or False, optional (default: "max_centrality")*) – Whether the centrality should be normalized. Accepted normalizations are "max_eigenvalue" (the matrix is divided by its largest eigenvalue), "max_centrality" (the largest centrality is one), and `False` to get the non-normalized centralities.

Returns **centralities** (`numpy.ndarray`) – The subgraph centrality of each node.

References

nngt.analysis.**total_firing_rate**(*network=None*, *spike_detector=None*, *nodes=None*, *data=None*, *kernel_center=0.0*, *kernel_std=30.0*, *resolution=None*, *cut_gaussian=5.0*)

Computes the total firing rate of the network from the spike times. Firing rate is obtained as the convolution of the spikes with a Gaussian kernel characterized by a standard deviation and a temporal shift.

New in version 0.7.

Parameters

- **network** (*nngt.Network*, optional (default: None)) – Network for which the activity was simulated.
- **spike_detector** (*tuple of ints, optional (default: spike detectors)*) – GID of the "spike_detector" objects recording the network activity.
- **data** (`numpy.array` of shape (N, 2), optionale (default: None)) – Array containing the spikes data (first line must contain the NEST GID of the neuron that fired, second line must contain the associated spike time).
- **kernel_center** (*float, optional (default: 0.)*) – Temporal shift of the Gaussian kernel, in ms.
- **kernel_std** (*float, optional (default: 30.)*) – Characteristic width of the Gaussian kernel (standard deviation) in ms.
- **resolution** (float or array, optional (default: *0.1*kernel_std*)) – The resolution at which the firing rate values will be computed. Choosing a value smaller than *kernel_std* is strongly advised. If resolution is an array, it will be considered as the times were the firing rate should be computed.
- **cut_gaussian** (*float, optional (default: 5.)*) – Range over which the Gaussian will be computed. By default, we consider the 5-sigma range. Decreasing this value will increase speed at the cost of lower fidelity; increasing it with increase the fidelity at the cost of speed.

Returns

- **fr** (*array-like*) – The firing rate in Hz.
- **times** (*array-like*) – The times associated to the firing rate values.

nngt.analysis.**transitivity**(*g*, *directed=True*, *weights=None*)

Same as *global_clustering()*.

nngt.analysis.**triangle_count**(*g*, *nodes=None*, *directed=True*, *weights=None*, *method='normal'*, *mode='total'*, *combine_weights='mean'*)

Returns the number or the strength (also called intensity) of triangles for each node.

Parameters

- **g** (*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.
- **directed** (*bool, optional (default: True)*) – Whether to compute the directed clustering if the graph is directed.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **method** (*str, optional (default: 'normal')*) – Method used to compute the weighted triangles, either 'normal', where the weights are directly used, or the definitions associated to the

weighted clustering: 'barrat' [Barrat2004], 'continuous', 'onnela' [Onnela2005], or 'zhang' [Zhang2005].

- **mode** (*str, optional (default: "total")*) – Type of clustering to use for directed graphs, among "total", "fan-in", "fan-out", "middleman", and "cycle" [Fagiolo2007].
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.

Returns **tr** (*array*) – Number or weight of triangles to which each node belongs.

References

nngt.analysis.**triplet_count**(*g*, *nodes=None*, *directed=True*, *weights=None*, *method='normal'*, *mode='total'*, *combine_weights='mean'*)

Returns the number or the strength (also called intensity) of triplets for each node.

For binary networks, the triplets of node i are defined as:

$$T_i = \sum_{j,k} a_{ij} a_{ik}$$

Parameters

- **g** (*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.
- **directed** (*bool, optional (default: True)*) – Whether to compute the directed clustering if the graph is directed.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **method** (*str, optional (default: 'continuous')*) – Method used to compute the weighted triplets, either 'normal', where the edge weights are directly used, or the definitions used for weighted clustering coefficients, 'barrat' [Barrat2004], 'continuous', 'onnela' [Onnela2005], or 'zhang' [Zhang2005].
- **mode** (*str, optional (default: "total")*) – Type of clustering to use for directed graphs, among "total", "fan-in", "fan-out", "middleman", and "cycle" [Fagiolo2007].
- **combine_weights** (*str, optional (default: 'mean')*) – How to combine the weights of reciprocal edges if the graph is directed but *directed* is set to False. It can be:
  - "sum": the sum of the edge attribute values will be used for the new edge.
  - "mean": the mean of the edge attribute values will be used for the new edge.
  - "min": the minimum of the edge attribute values will be used for the new edge.
  - "max": the maximum of the edge attribute values will be used for the new edge.

Returns **tr** (*array*) – Number or weight of triplets to which each node belongs.

References

Generation module

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

Content

Generation functions

nngt.generation.all_to_all([nodes, ...])	Generate a graph where all nodes are connected.
nngt.generation.circular(coord_nb[, ...])	Generate a circular graph.
nngt.generation.distance_rule(scale[, rule, ...])	Create a graph using a 2D distance rule to create the connection between neurons.
nngt.generation.erdos_renyi([density, ...])	Generate a random graph as defined by Erdos and Renyi but with a reciprocity that can be chosen.
nngt.generation.fixed_degree(degree[, ...])	Generate a random graph with constant in- or out-degree.
nngt.generation.from_degree_list(degrees[, ...])	Generate a random graph from a given list of degrees.
nngt.generation.gaussian_degree(avg, std[, ...])	Generate a random graph with constant in- or out-degree.
nngt.generation.newman_watts(coord_nb[, ...])	Generate a (potentially small-world) graph using the Newman-Watts algorithm.
nngt.generation.price_scale_free(m[, c, ...])	Generate a Price graph model (Barabasi-Albert if undirected).
nngt.generation.random_scale_free(in_exp, ...)	Generate a free-scale graph of given reciprocity and otherwise devoid of correlations.
nngt.generation.sparse_clustered(c[, nodes, ...])	Generate a sparse random graph with given average clustering coefficient and degree.
nngt.generation.watts_strogatz(coord_nb[, ...])	Generate a (potentially small-world) graph using the Watts-Strogatz algorithm.

Connectors

nngt.generation.connect_nodes(network, ...)	Function to connect nodes with a given graph model.
nngt.generation.connect_groups(network, ...)	Function to connect groups with a given graph model.
nngt.generation.connect_neural_types(...[, ...])	Function to connect excitatory and inhibitory population with a given graph model.

Rewiring functions

nngt.generation.random_rewire(g[, ...])	Generate a new rewired graph from *g*.
nngt.generation.lattice_rewire(g[, ...])	Build a (generally irregular) lattice by rewiring the edges of a graph.

Details

nngt.generation.**all_to_all**(*nodes=0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='AllToAll'*, *shape=None*, *positions=None*, *population=None*, ***kwargs*)

Generate a graph where all nodes are connected.

New in version 1.0.

Parameters

- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **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*).

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

Returns **graph_all** (*Graph*, or subclass) – A new generated graph.

nngt.generation.**circular**(*coord_nb*, *reciprocity=1.0*, *reciprocity_choice='random'*, *nodes=0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='Circular'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a circular graph.

The nodes are placed on a circle and connected to their *coord_nb* closest neighbours. If the graph is directed, the number of connections depends on the value of *reciprocity*: if `reciprocity == 0.`, then only half of all possible connections will be created, so that no bidirectional edges exist; on the other hand, for `reciprocity == 1.`, all possible edges are created; for intermediate values of *reciprocity*, the number of edges increases linearly as `0.5*(1 + reciprocity / (2 - reciprocity))*nodes*coord_nb`.

Parameters

- **coord_nb** (*int*) – The number of neighbours for each node on the initial topological lattice (must be even).

- **reciprocity** (*double, optional (default: 1.)*) – Proportion of reciprocal edges in the graph.
- **reciprocity_choice** (*str, optional (default: "random")*) – How reciprocal edges should be chosen, which can be either "random" or "closest". If the latter option is used, then connections between first neighbours are rendered reciprocal first, then between second neighbours, etc.
- **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)*) – 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_circ** (*Graph* or subclass)

nngt.generation.**connect_groups**(*network*, *source_groups*, *target_groups*, *graph_model*, *density=None*, *edges=None*, *avg_deg=None*, *unit='um'*, *weighted=True*, *directed=True*, *multigraph=False*, *check_existing=True*, *ignore_invalid=False*, ***kwargs*)

Function to connect groups with a given graph model.

Changed in version 2.0: Added *check_existing* and *ignore_invalid* arguments.

Parameters

- **network** (Network or SpatialNetwork) – The network to connect.
- **source_groups** (str, NeuralGroup, or iterable) – Names of the source groups (which contain the pre-synaptic neurons) or directly the group objects themselves.
- **target_groups** (str, NeuralGroup, or iterable) – Names of the target groups (which contain the post-synaptic neurons) or directly the group objects themselves.
- **graph_model** (*string*) – The name of the connectivity model (among "erdos_renyi", "random_scale_free", "price_scale_free", and "newman_watts").
- **check_existing** (*bool, optional (default: True)*) – Check whether some of the edges that will be added already exist in the graph.
- **ignore_invalid** (*bool, optional (default: False)*) – Ignore invalid edges: they are not added to the graph and are silently dropped. Unless this is set to true, an error is raised if an existing edge is re-generated.

- **kwargs** (*keyword arguments*) – Specific model parameters. or edge attributes specifiers such as *weights* or *delays*.

Note: For graph generation methods which set the properties of a specific degree (e.g. *gaussian_degree()*), the groups which have their property sets are the *source_groups*.

---

nngt.generation.**connect_neural_groups**(**args*, ***kwargs*)
Deprecatd alias of *connect_groups()*.

nngt.generation.**connect_neural_types**(*network*, *source_type*, *target_type*, *graph_model*, *density=None*, *edges=None*, *avg_deg=None*, *unit='um'*, *weighted=True*, *directed=True*, *multigraph=False*, *check_existing=True*, *ignore_invalid=False*, ***kwargs*)
Function to connect excitatory and inhibitory population with a given graph model.

Changed in version 2.0: Added *check_existing* and *ignore_invalid* arguments.

Parameters

- **network** (`Network` or `SpatialNetwork`) – The network to connect.
- **source_type** (*int or list*) – The type of source neurons (`1` for excitatory, `-1` for inhibitory neurons).
- **target_type** (*int or list*) – 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").
- **check_existing** (*bool, optional (default: True)*) – Check whether some of the edges that will be added already exist in the graph.
- **ignore_invalid** (*bool, optional (default: False)*) – Ignore invalid edges: they are not added to the graph and are silently dropped. Unless this is set to true, an error is raised if an existing edge is re-generated.
- **kwargs** (*keyword arguments*) – Specific model parameters. or edge attributes specifiers such as *weights* or *delays*.

Note: For graph generation methods which set the properties of a specific degree (e.g. *gaussian_degree()*), the nodes which have their property sets are the *source_type*.

---

nngt.generation.**connect_nodes**(*network*, *sources*, *targets*, *graph_model*, *density=None*, *edges=None*, *avg_deg=None*, *unit='um'*, *weighted=True*, *directed=True*, *multigraph=False*, *check_existing=True*, *ignore_invalid=False*, ***kwargs*)
Function to connect nodes with a given graph model.

Changed in version 2.0: Added *check_existing* and *ignore_invalid* arguments.

Parameters

- **network** (`Network` or `SpatialNetwork`) – The network to connect.
- **sources** (*list*) – Ids of the source nodes.
- **targets** (*list*) – Ids of the target nodes.
- **graph_model** (*string*) – The name of the connectivity model (among "erdos_renyi", "random_scale_free", "price_scale_free", and "newman_watts").

- **check_existing** (*bool, optional (default: True)*) – Check whether some of the edges that will be added already exist in the graph.
- **ignore_invalid** (*bool, optional (default: False)*) – Ignore invalid edges: they are not added to the graph and are silently dropped. Unless this is set to true, an error is raised if an existing edge is re-generated.
- ****kwargs** (*keyword arguments*) – Specific model parameters. or edge attributes specifiers such as *weights* or *delays*.

Note: For graph generation methods which set the properties of a specific degree (e.g. `gaussian_degree()`), the nodes which have their property sets are the *sources*.

---

nngt.generation.**distance_rule**(*scale*, *rule='exp'*, *shape=None*, *neuron_density=1000.0*, *max_proba=- 1.0*, *nodes=0*, *density=None*, *edges=None*, *avg_deg=None*, *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), "gaussian" (Gaussian), or "lin" (linear).
- **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.
- **p** (*float, optional*) – Normalization factor for the distance rule; it is equal to the probability of connection when testing a node at zero distance.
- **density** (*double, optional*) – 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)*) – 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**(*density=None*, *nodes=0*, *edges=None*, *avg_deg=None*, *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

- **density** (*double, optional (default: -1.)*) – Structural density given by *edges / nodes*^{2}. It is also the probability for each possible edge in the graph to exist.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **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'.
- **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.**from_degree_list**(*degrees*, *degree_type='in'*, *weighted=True*, *directed=True*, *multigraph=False*, *name='DL'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a random graph from a given list of degrees.

Parameters

- **degrees** (*list*) – The list of degrees for each node in the graph.
- **degree_type** (*str, optional (default: 'in')*) – The type of the fixed degree, among `'in'`, `'out'` or `'total'`.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **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_dl** (*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.

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' (or the full version: 'in-degree'...) @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.**lattice_rewire**(*g*, *target_reciprocity=1.0*, *node_attr_constraints=None*, *edge_attr_constraints=None*, *weight=None*, *weight_constraint='distance'*, *distance_sort='inverse'*)

Build a (generally irregular) lattice by rewiring the edges of a graph.

New in version 2.0.

The lattice is based on a circular graph, meaning that the nodes are placed on a circle and connected based on the topological distance between them, the distance being defined through the positive modulo:

$$d_{ij} = (i - j)\%N$$

with N the number of nodes in the graph.

Parameters

- **g** (*Graph*) – Graph based on which the lattice will be generated.
- **target_reciprocity** (*float, optional (default: 1.)*) – Value of reciprocity that should be aimed at. Depending on the number of edges, it may not be possible to reach this value exactly.
- **node_attr_constraints** (*str, optional (default: randomize all attributes)*) – Whether attribute randomization is constrained: either "preserve", where all nodes keep their attributes, or "together", where attributes are randomized by groups (all attributes of a given node are sent to the same new node). By default, attributes are completely and separately randomized.
- **edge_attr_constraints** (str, optional (default: randomize all but *weight*)) – Whether attribute randomization is constrained. If "distance" is used, then all number attributes (float or int) are sorted and are first associated to the shortest or longest edges depending on the value of *distance_sort*. Note that, for directed graphs, if a reciprocal edge exists, it is immediately assigned the next highest (respectively lowest) attribute after that of its directed couterpart. If "together" is used, edges attributes are randomized by groups (all attributes of a given edge are sent to the same new edge) either randomly if *weight* is None, or following the constrained *weight* attribute. By default, attributes are completely and separately randomized (except for *weight* if it has been provided).
- **weight** (*str, optional (default: None)*) – Whether a specific edge attribute should play the role of weight and have special constraints.
- **weight_constraint** (*str, optional (default: "distance")*) – Same as *edge_attr_constraints* ` but only applies to *weight* and can only be "distance" or None since "together" was related to *weight*.
- **distance_sort** (*str, optional (default: "inverse")*) – How attributes are sorted with edge distance: either "inverse", with the shortest edges being assigned the largest weights, or with a "linear" sort, where shortest edges are assigned the lowest weights.

nngt.generation.**newman_watts**(*coord_nb*, *proba_shortcut=None*, *reciprocity_circular=1.0*, *reciprocity_choice_circular='random'*, *nodes=0*, *edges=None*, *weighted=True*, *directed=True*, *multigraph=False*, *name='NW'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a (potentially small-world) graph using the Newman-Watts algorithm.

For directed networks, the reciprocity of the initial circular network can be chosen.

Changed in version 2.0: Added the *reciprocity_circular* and *reciprocity_choice_circular* options.

Parameters

- **coord_nb** (*int*) – The number of neighbours for each node on the initial topological lattice (must be even).
- **proba_shortcut** (*double, optional*) – Probability of adding a new random (shortcut) edge for each existing edge on the initial lattice. If *edges* is provided, then will be computed automatically as `edges / (coord_nb * nodes * (1 + reciprocity_circular) / 2)`
- **reciprocity_circular** (*double, optional (default: 1.)*) – Proportion of reciprocal edges in the initial circular graph.
- **reciprocity_choice_circular** (*str, optional (default: "random")*) – How reciprocal edges should be chosen in the initial circular graph. This can be either "random" or "closest". If the latter option is used, then connections between first neighbours are rendered reciprocal first, then between second neighbours, etc.

- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **edges** (*int (optional)*) – The number of edges between the nodes.
- **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_nw** (*Graph* or subclass)

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

---

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

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

Parameters

- **m** (*int*) – The number of edges each new node will make.
- **c** (*double, optional (0 if undirected, else 1)*) – Constant added to the probability of a vertex receiving an edge.
- **gamma** (*double, optional (default: 1)*) – Preferential attachment power.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **reciprocity** (*float, optional (default: 0)*) – Reciprocity of the graph (between 0 and 1). For directed graphs, this will be the probability of the target node connecting back to the source node when a new edge is added.
- **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*).

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

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

Notes

The (generalized) Price network is either a directed or undirected graph (the latter is better known as the Barabási-Albert network). It is generated via a growth process, adding a new node at each step and connecting it to m previous nodes, chosen with probability:

$$p \propto k^{\gamma} + c$$

where k is the (in-)degree of the vertex.

We must therefore have $c \geq 0$ for directed graphs and $c > -1$ for undirected graphs.

If the *reciprocity* r is non-zero, each targeted node reciprocates the connection with probability r. Expected reciprocity of the final graph is $2r/(1+r)$.

If $\gamma = 1$, and *reciprocity* is zero, the tail of resulting in-degree distribution of the directed case is given by

$$P_{k_{in}} \sim k_{in}^{-(2+c/m)},$$

or for the undirected case

$$P_k \sim k^{-(3+c/m)}.$$

However, if $\gamma \neq 1$, the in-degree distribution is not scale-free.

nngt.generation.**random_rewire**(*g*, *constraints=None*, *node_attr_constraints=None*, *edge_attr_constraints=None*, ***kwargs*)

Generate a new rewired graph from *g*.

New in version 2.0.

Parameters

- **g** (*Graph*) – Base graph based on which a new rewired graph will be generated.
- **constraints** (*str, optional (default: no constraints)*) – Defines which properties of *g* will be maintained in the rewired graph. By default, the graph is completely rewired into an Erdos-Renyi model. Available constraints are "in-degree", "out-degree", "total-degree", "all-degrees", and "clustering".
- **node_attr_constraints** (*str, optional (default: randomize all attributes)*) – Whether attribute randomization is constrained: either "preserve", where all nodes keep their attributes, or "together", where attributes are randomized by groups (all attributes of a given node are sent to the same new node). By default, attributes are completely and separately randomized.
- **edge_attr_constraints** (*str, optional (default: randomize all attributes)*) – Whether attribute randomization is constrained. If *constraints* is "in-degree" (respectively "out-degree") or "degrees", this can be "preserve_in" (respectively "preserve_out"), in which case all attributes of a given edge are moved together to a new incoming (respectively outgoing) edge of the same node. Regardless of *constraints*, "together" can be used so that edges attributes are randomized by groups (all attributes of a given edge are sent to the same new edge). By default, attributes are completely and separately randomized.

- ****kwargs** (*optional keyword arguments*) – These are optional arguments in the case *constraints* is "clustering". In that case, the user can provide both:
  - *rtol* : float, optional (default: 5%) The tolerance on the relative error to the average clustering for the rewired graph.
  - *connected* : bool, optional (default: False) Whether the generated graph should be connected (this reduces the precision of the final clustering).
  - *method* : str, optional (default: "star-component") Defines how the initially disconnected components of the generated graph should be connected among themselves. Available methods are "sequential", where the components are connected sequentially, forming a long thread and increasing the graph's diameter, "star-component", where all disconnected components are connected to random nodes in the largest component, "central-node" , where all disconnected components are connected to the same node in the largest component, and "random", where components are connected randomly.

nngt.generation.**random_scale_free**(*in_exp*, *out_exp*, *nodes=0*, *density=None*, *edges=None*, *avg_deg=None*, *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: 0)*) – The number of nodes in the graph.
- **density** (*double, optional*) – 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)*) – 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.**sparse_clustered**(*c*, *nodes=0*, *edges=None*, *avg_deg=None*, *connected=True*, *rtol=None*, *exact_edge_nb=False*, *weighted=True*, *directed=True*, *multigraph=False*, *name='FC'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a sparse random graph with given average clustering coefficient and degree.

New in version 2.5.

The original algorithm is adapted from [newman-clustered-2003] and leads to a graph with approximate clustering coefficient and number of edges.

Warning: This algorithm can only give reasonable results for sparse graphs and will raise an error if the requested graph density is above c.

Nodes are distributed among μ overlapping groups of size ν and, each time two nodes belong to a common group, they are connected with a probability $p = c$.

For sparse graphs, the average (in/out-)degree can be approximmated as $k = \mu p(\nu-1)$, and the average clustering as:

$$C^{(u)} = \frac{p\,[p(\nu-1)-1]}{k-1}$$

for undirected graphs and

$$C^{(d)} = \frac{p\mu\,[p(2\nu-3)-1]}{2k-1-p}$$

for all clustering modes in directed graphs.

From these relations, we compute μ and ν as:

$$\nu^{(u)} = 1 + \frac{1}{p} + \frac{C^{(u)}(k-1)}{p^2}$$

or

$$\nu^{(d)} = \frac{3}{2} + \frac{1}{2p} + \frac{C^{(u)}(2k-1-p)}{2p^2}$$

and

$$\mu = \frac{k}{p(\nu-1)}.$$

Parameters

- **c** (*float*) – Desired value for the final average clustering in the graph.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **edges** (*int, optional*) – The number of edges between the nodes
- **avg_deg** (*double, optional*) – Average degree of the neurons given by *edges / nodes*.
- **connected** (*bool, optional (default: True)*) – Whether the resulting graph must be connected (True) or may be unconnected (False).
- **rtol** (*float, optional (default: not checked)*) – Tolerance on the relative error between the target clustering c and the actual clustering of the final graph. If the algorithm leads to a relative error greater than *rtol*, then an error is raised.

- **exact_edge_nb** (*bool, optional (default: False)*) – Whether the final graph should have precisely the number of edges required.
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – Whether the graph should be 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.
- ****kwargs** (*keyword arguments*) – If connected is True, *method* can be passed to define how the components should be connected among themselves. Available methods are "sequential", where the components are connected sequentially, forming a long thread and increasing the graph's diameter, "star-component", where all components are connected to the largest one, and "random", where components are connected randomly, or "central-node", where one node from the largest component is chosen to reconnect all disconnected components. If not provided, defaults to "random".

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

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

References

nngt.generation.**watts_strogatz**(*coord_nb*, *proba_shortcut=None*, *reciprocity_circular=1.0*, *reciprocity_choice_circular='random'*, *shuffle='random'*, *nodes=0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='WS'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

Generate a (potentially small-world) graph using the Watts-Strogatz algorithm.

For directed networks, the reciprocity of the initial circular network can be chosen.

New in version 2.0.

Parameters

- **coord_nb** (*int*) – The number of neighbours for each node on the initial topological lattice (must be even).
- **proba_shortcut** (*double, optional*) – Probability of adding a new random (shortcut) edge for each existing edge on the initial lattice. If *edges* is provided, then will be computed automatically as `edges / (coord_nb * nodes * (1 + reciprocity_circular) / 2)`

- **reciprocity_circular** (*double, optional (default: 1.)*) – Proportion of reciprocal edges in the initial circular graph.
- **reciprocity_choice_circular** (*str, optional (default: "random")*) – How reciprocal edges should be chosen in the initial circular graph. This can be either "random" or "closest". If the latter option is used, then connections between first neighbours are rendered reciprocal first, then between second neighbours, etc.
- **shuffle** (*str, optional (default: 'random')*) – Whether to shuffle only 'targets' (out-degree of all nodes remains constant), 'sources' (in-degree remains constant), or randomly the source or the target for each edge ('random') in the case of directed graphs.
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **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_nw** (*Graph* or subclass)

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

Geometry module

This module is a direct copy of the SENeC package PyNCulture. Therefore, in the examples below, you will have to import `nngt` instead of `PyNCulture` and replace `pnc` by `nngt.geometry`.

Overview

nngt.geometry.Area(shell[, holes, unit, ...])	Specialized Shape that stores additional properties regarding the interactions with the neurons.
nngt.geometry.Shape(shell[, holes, unit, ...])	Class containing the shape of the area where neurons will be distributed to form a network.
nngt.geometry.culture_from_file(filename[, ...])	Generate a culture from an SVG, a DXF, or a WKT/WKB file.
nngt.geometry.plot_shape(shape[, axis, m, ...])	Plot a shape (you should set the *axis* aspect to 1 to respect the proportions).
nngt.geometry.pop_largest(shapes)	Returns the largest shape, removing it from the list.

continues on next page

Table 18 – continued from previous page

nngt.geometry.shapes_from_file(filename[, ...])	Generate a set of Shape objects from an SVG, a DXF, or a WKT/WKB file.

Principle

Module dedicated to the description of the spatial boundaries of neuronal cultures. This allows for the generation of neuronal networks that are embedded in space.

The shapely library is used to generate and deal with the spatial environment of the neurons.

Examples

Basic features

The module provides a backup Shape object, which can be used with only the *numpy* and *scipy* libraries. It allows for the generation of simple rectangle, disk and ellipse shapes.

```
import matplotlib.pyplot as plt

import PyNCulture as nc


fig, ax = plt.subplots()

''' Choose a shape (uncomment the desired line) '''
# culture = nc.Shape.rectangle(15, 20, (5, 0))
culture = nc.Shape.disk(20, (5, 0))
# culture = nc.Shape.ellipse((20, 5), (5, 0))

''' Generate the neurons inside '''
pos = culture.seed_neurons(neurons=1000, xmax=0., ymax=0.)

''' Plot '''
nc.plot_shape(culture, ax, show=False)
ax.scatter(pos[:, 0], pos[:, 1], s=2, zorder=2)

plt.show()
```

All these features are of course still available with the more advanced Shape object which inherits from shapely.geometry.Polygon.

Complex shapes from files

```
import matplotlib.pyplot as plt

import PyNCulture as nc


''' Choose a file '''
culture_file = "culture_from_filled_polygons.svg"
# culture_file = "culture_with_holes.svg"
# culture_file = "culture.dxf"

shapes = nc.shapes_from_file(culture_file, min_x=-5000., max_x=5000.)

''' Plot the shapes '''
fig, ax = plt.subplots()
fig.suptitle("shapes")

for p in shapes:
    nc.plot_shape(p, ax, show=False)

plt.show()

''' Make a culture '''
fig2, ax2 = plt.subplots()
plt.title("culture")

culture = nc.culture_from_file(culture_file, min_x=-5000., max_x=5000.)

nc.plot_shape(culture, ax2)

''' Add neurons '''
fig3, ax3 = plt.subplots()
plt.title("culture with neurons")

culture_bis = nc.culture_from_file(culture_file, min_x=-5000., max_x=5000.)
pos = culture_bis.seed_neurons(neurons=1000, xmax=0)

nc.plot_shape(culture_bis, ax3, show=False)
ax3.scatter(pos[:, 0], pos[:, 1], s=2, zorder=3)

plt.show()
```

Content

class nngt.geometry.**Area**(*shell*, *holes=None*, *unit='um'*, *height=0.0*, *name='area'*, *properties=None*)

Specialized *Shape* that stores additional properties regarding the interactions with the neurons.

Each Area is characteristic of a given substrate and height. These two properties are homogeneous over the whole area, meaning that the neurons interact in the same manner with an Area reagardless of their position inside.

The substrate is described through its modulation of the neuronal properties compared to their default behavior. Thus, a given area will modulate the speed, wall affinity, etc, of the growth cones that are growing above it.

Initialize the *Shape* object and the underlying `shapely.geometry.Polygon`.

Parameters

- **shell** (*array-like object of shape (N, 2)*) – List of points defining the external border of the shape.
- **holes** (*array-like, optional (default: None)*) – List of array-like objects of shape (M, 2), defining empty regions inside the shape.
- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'.
- **height** (*float, optional (default: 0.)*) – Height of the area.
- **name** (*str, optional (default: "area")*) – The name of the area.
- **properties** (*dict, optional (default: default neuronal properties)*) – Dictionary containing the list of the neuronal properties that are modified by the substrate. Since this describes how the default property is modulated, all values must be positive reals or NaN.

property areas

Returns the dictionary containing the Shape's areas.

copy()

Create a copy of the current Area.

classmethod from_shape(*shape*, *height=0.0*, *name='area'*, *properties=None*, *unit='um'*, *min_x=None*, *max_x=None*)

Create an *Area* from a *Shape* object.

Parameters **shape** (*Shape*) – Shape that should be converted to an Area.

Returns *Area* object.

class nngt.geometry.**Shape**(*shell*, *holes=None*, *unit='um'*, *parent=None*, *default_properties=None*)

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

area

Area of the shape in the *Shape*'s *Shape.unit()* squared (μm^2, mm^2, cm^2, dm^2 or m^2).

Type double

centroid

Position of the center of mass of the current shape in *unit*.

Type tuple of doubles

See also:

`Parent`

Initialize the *Shape* object and the underlying `shapely.geometry.Polygon`.

Parameters

- **exterior** (*array-like object of shape (N, 2)*) – List of points defining the external border of the shape.
- **interiors** (*array-like, optional (default: None)*) – List of array-like objects of shape (M, 2), defining empty regions inside the shape.
- **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.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

add_area(*area*, *height=None*, *name=None*, *properties=None*, *override=False*)
Add a new area to the *Shape*. If the new area has a part that is outside the main *Shape*, it will be cut and only the intersection between the area and the container will be kept.

Parameters

- **area** (*Area* or *Shape*, or shapely.Polygon.) – Delimitation of the area. Only the intersection between the parent *Shape* and this new area will be kept.
- **name** (*str, optional, default ("areaX" where X is the number of areas)*) – Name of the area, under which it can be retrieved using the *Shape.area()* property of the *Shape* object.
- **properties** (*dict, optional (default: None)*) – Properties of the area. If *area* is a *Area*, then this is not necessary.
- **override** (*bool, optional (default: False)*) – If True, the new area will be made over existing areas that will be reduced in consequence.

add_hole(*hole*)
Make a hole in the shape.

New in version 0.4.

property areas
Returns the dictionary containing the Shape's areas.

contains_neurons(*positions*)
Check whether the neurons are contained in the shape.

New in version 0.4.

Parameters **positions** (*point or 2D-array of shape (N, 2)*)

Returns **contained** (*bool or 1D boolean array of length N*) – True if the neuron is contained, False otherwise.

copy()
Create a copy of the current Shape.

property default_areas
Returns the dictionary containing only the default areas.

New in version 0.4.

static disk(*radius*, *centroid=(0.0, 0.0)*, *unit='um'*, *parent=None*, *default_properties=None*)
Generate a disk of given radius and center (*centroid*).

Parameters

- **radius** (*float*) – Radius of the disk in *unit*
- **centroid** (*tuple of floats, optional (default: (0., 0.))*) – Position of the rectangle's center of mass in *unit*

- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'
- **parent** (nngt.Graph or subclass, optional (default: None)) – The parent container.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

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

static ellipse(*radii*, *centroid=(0.0, 0.0)*, *unit='um'*, *parent=None*, *default_properties=None*)

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

Parameters

- **radii** (*tuple of floats*) – Couple (rx, ry) containing the radii of the two axes in *unit*
- **centroid** (*tuple of floats, optional (default: (0., 0.))*) – Position of the rectangle's center of mass in *unit*
- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'
- **parent** (nngt.Graph or subclass, optional (default: None)) – The parent container.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

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

static from_file(*filename*, *min_x=None*, *max_x=None*, *unit='um'*, *parent=None*, *interpolate_curve=50*, *default_properties=None*)

Create a shape from a DXF, an SVG, or a WTK/WKB file.

New in version 0.3.

Parameters

- **filename** (*str*) – Path to the file that should be loaded.
- **min_x** (*float, optional (default: -5000.)*) – Absolute horizontal position of the leftmost point in the environment in *unit* (default: 'um'). If None, no rescaling occurs.
- **max_x** (*float, optional (default: 5000.)*) – Absolute horizontal position of the rightmost point in the environment in *unit*. If None, no rescaling occurs.
- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'.
- **parent** (nngt.Graph object) – The parent which will become a nngt.SpatialGraph.
- **interpolate_curve** (*int, optional (default: 50)*) – Number of points that should be used to interpolate a curve.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

static from_polygon(*polygon*, *min_x=None*, *max_x=None*, *unit='um'*, *parent=None*, *default_properties=None*)

Create a shape from a shapely.geometry.Polygon.

Parameters

- **polygon** (shapely.geometry.Polygon) – The initial polygon.

- **min_x** (*float, optional (default: -5000.)*) – Absolute horizontal position of the leftmost point in the environment in *unit* If None, no rescaling occurs.
- **max_x** (*float, optional (default: 5000.)*) – Absolute horizontal position of the rightmost point in the environment in *unit* If None, no rescaling occurs.
- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'
- **parent** (*nngt.Graph* object) – The parent which will become a *nngt.SpatialGraph*.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

static from_wkt(*wtk*, *min_x=None*, *max_x=None*, *unit='um'*, *parent=None*, *default_properties=None*)
Create a shape from a WKT string.

New in version 0.2.

Parameters

- **wtk** (*str*) – The WKT string.
- **min_x** (*float, optional (default: -5000.)*) – Absolute horizontal position of the leftmost point in the environment in *unit* If None, no rescaling occurs.
- **max_x** (*float, optional (default: 5000.)*) – Absolute horizontal position of the rightmost point in the environment in *unit* If None, no rescaling occurs.
- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'
- **parent** (*nngt.Graph* object) – The parent which will become a *nngt.SpatialGraph*.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

See also:

Shape.from_polygon()

property non_default_areas
Returns the dictionary containing all Shape's areas except the default ones.

New in version 0.4.

property parent
Return the parent of the *Shape*.

random_obstacles(*n*, *form*, *params=None*, *heights=None*, *properties=None*, *etching=0*, *on_area=None*)
Place random obstacles inside the shape.

New in version 0.4.

Parameters

- **n** (*int or float*) – Number of obstacles if *n* is an `int`, otherwise represents the fraction of the shape's bounding box that should be occupied by

  the obstacles' bounding boxes.
- **form** (*str or Shape*) – Form of the obstacles, among "disk", "ellipse", "rectangle", or a custom shape.

- **params** (*dict, optional (default: None)*) – Dictionnary containing the instructions to build a predefined form ("disk", "ellipse", "rectangle"). See their creation methods for details. Leave *None* when using a custom shape.
- **heights** (*float or list, optional (default: None)*) – Heights of the obstacles. If None, the obstacle will considered as a "hole" in the structure, i.e. an uncrossable obstacle.
- **properties** (*dict or list, optional (default: None)*) – Properties of the obstacles if they constitue areas (only used if *heights* is not None). If not provided and *heights* is not None, will default to the "default_area" properties.
- **etching** (*float, optional (default: 0)*) – Etching of the obstacles' corners (rounded corners).

`static rectangle`(*height*, *width*, *centroid=(0.0, 0.0)*, *unit='um'*, *parent=None*, *default_properties=None*)

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

Parameters

- **height** (*float*) – Height of the rectangle in *unit*
- **width** (*float*) – Width of the rectangle in *unit*
- **centroid** (*tuple of floats, optional (default: (0., 0.))*) – Position of the rectangle's center of mass in *unit*
- **unit** (*string (default: 'um')*) – Unit in the metric system among 'um' (μm), 'mm', 'cm', 'dm', 'm'
- **parent** (*nngt.Graph* or subclass, optional (default: None)) – The parent container.
- **default_properties** (*dict, optional (default: None)*) – Default properties of the environment.

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

`property return_quantity`

Whether *seed_neurons* returns positions with units by default.

New in version 0.5.

`seed_neurons`(*neurons=None*, *container=None*, *on_area=None*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *soma_radius=0*, *unit=None*, *return_quantity=None*)

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

Parameters

- **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*.
- **container** (*Shape*, optional (default: None)) – Subshape acting like a mask, in which the neurons must be contained. The resulting area where the neurons are generated is the `intersection()` between of the current shape and the *container*.
- **on_area** (*str or list, optional (default: None)*) – Area(s) where the seeded neurons should be.
- **xmin** (*double, optional (default: lowest abscissa of the Shape)*) – Limit the area where neurons will be seeded to the region on the right of *xmin*.
- **xmax** (*double, optional (default: highest abscissa of the Shape)*) – Limit the area where neurons will be seeded to the region on the left of *xmax*.
- **ymin** (*double, optional (default: lowest ordinate of the Shape)*) – Limit the area where neurons will be seeded to the region on the upper side of *ymin*.

- **ymax** (*double, optional (default: highest ordinate of the Shape)*) – Limit the area where neurons will be seeded to the region on the lower side of *ymax*.
- **unit** (*string (default: None)*) – Unit in which the positions of the neurons will be returned, among 'um', 'mm', 'cm', 'dm', 'm'.
- **return_quantity** (*bool, optional (default: False)*) – Whether the positions should be returned as `pint.Quantity` objects (requires Pint).
- **.. versionchanged:: 0.5** – Accepts *pint* units and *return_quantity* argument.

Note: If both *container* and *on_area* are provided, the intersection of the two is used.

Returns **positions** (array of double with shape (N, 2) or *pint.Quantity* if) – *return_quantity* is *True*.

set_parent(*parent*)
Set the parent *nngt.Graph*.

set_return_units(*b*)
Set the default behavior for positions returned by *seed_neurons*. If *True*, then the positions returned are quantities with units (from the *pint* library), otherwise they are simply numpy arrays.

New in version 0.5.

Note: *set_return_units(True)* requires *pint* to be installed on the system, otherwise an error will be raised.

property unit
Return the unit for the *Shape* coordinates.

nngt.geometry.**culture_from_file**(*filename*, *min_x=None*, *max_x=None*, *unit='um'*, *parent=None*, *interpolate_curve=50*, *internal_shapes_as='holes'*, *default_properties=None*, *other_properties=None*)
Generate a culture from an SVG, a DXF, or a WKT/WKB file.

Valid file needs to contain only closed objects among: rectangles, circles, ellipses, polygons, and closed curves. The objects do not have to be simply connected.

Changed in version 0.6: Added *internal_shapes_as* and *other_properties* keyword parameters.

Parameters

- **filename** (*str*) – Path to the SVG, DXF, or WKT/WKB file.
- **min_x** (*float, optional (default: -5000.)*) – Position of the leftmost coordinate of the shape's exterior, in *unit*.
- **max_x** (*float, optional (default: 5000.)*) – Position of the rightmost coordinate of the shape's exterior, in *unit*.
- **unit** (*str, optional (default: 'um')*) – Unit of the positions, among micrometers ('um'), milimeters ('mm'), centimeters ('cm'), decimeters ('dm'), or meters ('m').
- **parent** (*nngt.Graph* or subclass, optional (default: None)) – Assign a parent graph if working with NNGT.
- **interpolate_curve** (*int, optional (default: 50)*) – Number of points by which a curve should be interpolated into segments.

- **internal_shapes_as** (*str, optional (default: "holes")*) – Defines how additional shapes contained in the main environment should be processed. If "holes", then these shapes are substracted from the main environment; if "areas", they are considered as areas.
- **default_properties** (*dict, optional (default: None)*) – Properties of the default area of the culture.
- **other_properties** (*dict, optional (default: None)*) – Properties of the non-default areas of the culture (internal shapes if *internal_shapes_as* is set to "areas").

Returns **culture** (*Shape* object) – Shape, vertically centred around zero, such that $min(y) + max(y) = 0$.

nngt.geometry.**plot_shape**(*shape*, *axis=None*, *m=''*, *mc='#999999'*, *fc='#8888ff'*, *ec='#444444'*, *alpha=0.5*, *brightness='height'*, *show_contour=True*, *show=True*, ***kwargs*)

Plot a shape (you should set the *axis* aspect to 1 to respect the proportions).

Parameters

- **shape** (*Shape*) – Shape to plot.
- **axis** (`matplotlib.axes.Axes` instance, optional (default: None)) – Axis on which the shape should be plotted. By default, a new figure is created.
- **m** (*str, optional (default: invisible)*) – Marker to plot the shape's vertices, matplotlib syntax.
- **mc** (*str, optional (default: "#999999")*) – Color of the markers.
- **fc** (*str, optional (default: "#8888ff")*) – Color of the shape's interior.
- **ec** (*str, optional (default: "#444444")*) – Color of the shape's edges.
- **alpha** (*float, optional (default: 0.5)*) – Opacity of the shape's interior.
- **brightness** (*str, optional (default: height)*) – Show how different other areas are from the 'default_area' (lower values are darker, higher values are lighter). Difference can concern the 'height', or any of the *properties* of the *Area* objects.
- **show_contour** (*bool, optional (default: True)*) – Whether the shapes should be drawn with a contour.
- **show** (*bool, optional (default: True)*) – Whether the plot should be displayed immediately.
- ****kwargs** (keywords arguments for `matplotlib.patches.PathPatch`)

nngt.geometry.**pop_largest**(*shapes*)

Returns the largest shape, removing it from the list. If *shapes* is a `shapely.geometry.MultiPolygon`, returns the largest `shapely.geometry.Polygon` without modifying the object.

New in version 0.3.

Parameters **shapes** (list of *Shape* objects or MultiPolygon.)

nngt.geometry.**shapes_from_file**(*filename*, *min_x=None*, *max_x=None*, *unit='um'*, *parent=None*, *interpolate_curve=50*, *default_properties=None*, ***kwargs*)

Generate a set of *Shape* objects from an SVG, a DXF, or a WKT/WKB file.

Valid file needs to contain only closed objects among: rectangles, circles, ellipses, polygons, and closed curves. The objects do not have to be simply connected.

New in version 0.3.

Parameters

- **filename** (*str*) – Path to the SVG, DXF, or WKT/WKB file.

- **min_x** (*float, optional (default: -5000.)*) – Position of the leftmost coordinate of the shape's exterior, in *unit*.
- **max_x** (*float, optional (default: 5000.)*) – Position of the rightmost coordinate of the shape's exterior, in *unit*.
- **unit** (*str, optional (default: 'um')*) – Unit of the positions, among micrometers ('um'), milimeters ('mm'), centimeters ('cm'), decimeters ('dm'), or meters ('m').
- **parent** (*nngt.Graph* or subclass, optional (default: None)) – Assign a parent graph if working with NNGT.
- **interpolate_curve** (*int, optional (default: 50)*) – Number of points by which a curve should be interpolated into segments.

Returns culture (*Shape* object) – Shape, vertically centred around zero, such that $min(y) + max(y) = 0$.

Lib module

Tools for the other modules.

Warning: These tools have been designed primarily for internal use throughout the library and often work only in very specific situations (e.g. *find_idx_nearest()* works only on sorted arrays), so make sure you read their doc carefully before using them.

Content

nngt.lib.InvalidArgument	Error raised when an argument is invalid.
nngt.lib.delta_distrib([graph, elist, num, ...])	Delta distribution for edge attributes.
nngt.lib.find_idx_nearest(array, values)	Find the indices of the nearest elements of *values* in a sorted *array*.
nngt.lib.gaussian_distrib(graph[, elist, ...])	Gaussian distribution for edge attributes.
nngt.lib.is_integer(obj)	Return whether the object is an integer
nngt.lib.is_iterable(obj)	Return whether the object is iterable
nngt.lib.lin_correlated_distrib(graph[, ...])	
nngt.lib.log_correlated_distrib(graph[, ...])	
nngt.lib.lognormal_distrib(graph[, elist, ...])	Lognormal distribution for edge attributes.
nngt.lib.nonstring_container(obj)	Returns true for any iterable which is not a string or byte sequence.
nngt.lib.uniform_distrib(graph[, elist, ...])	Uniform distribution for edge attributes.

Details

class nngt.lib.**InvalidArgument**
Error raised when an argument is invalid.

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** (*list of edges, optional (default: all edges)*) – Generate values for only a subset of edges.
- **value** (*float, optional (default: 1.)*) – Value of the delta distribution.
- **Returns** (numpy.ndarray) – Attribute value for each edge in *graph*.

nngt.lib.**find_idx_nearest**(*array*, *values*)
Find the indices of the nearest elements of *values* in a sorted *array*.

> **Warning:** Both `array` and `values` should be *numpy.array* objects and *array* MUST be sorted in increasing order.

Parameters

- **array** (*reference list or np.ndarray*)
- **values** (double, list or array of values to find in *array*)

Returns **idx** (int or array representing the index of the closest value in *array*)

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

Parameters

- **graph** (*Graph* or subclass) – Graph for which an edge attribute will be generated.
- **elist** (*list of edges, optional (default: all edges)*) – Generate values for only a subset of edges.
- **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*.

nngt.lib.**is_integer**(*obj*)
Return whether the object is an integer

nngt.lib.**is_iterable**(*obj*)
Return whether the object is iterable

nngt.lib.**lin_correlated_distrib**(*graph*, *elist=None*, *correl_attribute='betweenness'*, *noise_scale=None*, *lower=None*, *upper=None*, *slope=None*, *offset=0.0*, *last_edges=False*, ***kwargs*)

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=None*, *scale=None*, ***kwargs*)
Lognormal distribution for edge attributes.

Parameters

- **graph** (*Graph* or subclass) – Graph for which an edge attribute will be generated.
- **elist** (*list of edges, optional (default: all edges)*) – Generate values for only a subset of edges.
- **position** (*float, optional (default: 0.)*) – Average of the normal distribution (i.e. log of the actual mean of the lognormal distribution).
- **scale** (*float, optional (default: 1.5)*) – Standard deviation of the normal distribution.
- **Returns** (numpy.ndarray) – Attribute value for each edge in *graph*.

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

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

Parameters

- **graph** (*Graph* or subclass) – Graph for which an edge attribute will be generated.
- **elist** (*list of edges, optional (default: all edges)*) – Generate values for only a subset of edges.
- **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*.

Plot module

Functions for plotting graphs and graph properties.

The following features are provided:

- basic graph plotting
- plotting the distribution of some attribute over the graph
- animation of some recorded activity

Content

nngt.plot.Animation2d(source, multimeter[, ...])	Class to plot the raster plot, firing-rate, and average trajectory in a 2D phase-space for a network activity.
nngt.plot.AnimationNetwork(source, network)	Class to plot the raster plot, firing-rate, and space-embedded spiking activity (neurons on the graph representation flash when spiking) in time.
nngt.plot.betweenness_distribution(network)	Plotting the betweenness distribution of a graph.
nngt.plot.chord_diagram(network[, weights, ...])	Plot a chord diagram.
nngt.plot.compare_population_attributes(...)	Compare node *attributes* between two sets of nodes.
nngt.plot.correlation_to_attribute(network, ...)	For each node plot the value of *reference_attributes* against each of the *other_attributes* to check for correlations.
nngt.plot.degree_distribution(network[, ...])	Plotting the degree distribution of a graph.
nngt.plot.draw_network(network[, nsize, ...])	Draw a given graph/network.

continues on next page

Table 20 – continued from previous page

nngt.plot.edge_attributes_distribution(...)	Return node *attributes* for a set of *nodes*.
nngt.plot.hive_plot(network, radial[, axes, ...])	Draw a hive plot of the graph.
nngt.plot.library_draw(network[, nsize, ...])	Draw a given Graph using the underlying library's drawing functions.
nngt.plot.node_attributes_distribution(...)	Return node *attributes* for a set of *nodes*.
nngt.plot.palette_continuous([numbers])	
nngt.plot.palette_discrete([numbers])	

Details

class nngt.plot.**Animation2d**(*source*, *multimeter*, *start=0.0*, *timewindow=None*, *trace=5.0*, *x='time'*, *y='V_m'*, *sort_neurons=None*, *network=None*, *interval=50*, *vector_field=False*, ***kwargs*)

Class to plot the raster plot, firing-rate, and average trajectory in a 2D phase-space for a network activity.

Generate a SubplotAnimation instance to plot a network activity.

Parameters

- **source** (*tuple*) – NEST gid of the ``spike_detector``(s) which recorded the network.
- **multimeter** (*tuple*) – NEST gid of the ``multimeter``(s) which recorded the network.
- **timewindow** (*double, optional (default: None)*) – Time window which will be shown for the spikes and self.second.
- **trace** (*double, optional (default: 5.)*) – Interval of time (ms) over which the data is overlayed in red.
- **x** (*str, optional (default: "time")*) – Name of the *x*-axis variable (must be either "time" or the name of a NEST recordable in the *multimeter*).
- **y** (*str, optional (default: "V_m")*) – Name of the *y*-axis variable (must be either "time" or the name of a NEST recordable in the *multimeter*).
- **vector_field** (*bool, optional (default: False)*) – Whether the $\dot{x}$ and $\dot{y}$ arrows should be added to phase space. Requires additional 'dotx' and 'doty' arguments which are user defined functions to compute the derivatives of *x* and *x* in time. These functions take 3 parameters, which are *x*, *y*, and *time_dependent*, where the last parameter is a list of doubles associated to recordables from the neuron model (see example for details). These recordables must be declared in a *time_dependent* parameter.
- **sort_neurons** (*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", 'B2') or a user-defined list of sorted neuron ids. Sorting is performed by increasing value of the *sort_neurons* property from bottom to top inside each group.
- ****kwargs** (*dict, optional (default: {})*) – Optional arguments such as 'make_rate', 'num_xarrows', 'num_yarrows', 'dotx', 'doty', 'time_dependent', 'recordables', 'arrow_scale'.

class nngt.plot.**AnimationNetwork**(*source*, *network*, *resolution=1.0*, *start=0.0*, *timewindow=None*, *trace=5.0*, *show_spikes=False*, *sort_neurons=None*, *decimate_connections=False*, *interval=50*, *repeat=True*, *resting_size=None*, *active_size=None*, ***kwargs*)

Class to plot the raster plot, firing-rate, and space-embedded spiking activity (neurons on the graph representation

flash when spiking) in time.

Generate a SubplotAnimation instance to plot a network activity.

Parameters

- **source** (*tuple*) – NEST gid of the ``spike_detector``(s) which recorded the network.
- **network** (*SpatialNetwork*) – Network embedded in space to plot the actvity of the neurons in space.
- **resolution** (*double, optional (default: None)*) – Time resolution of the animation.
- **timewindow** (*double, optional (default: None)*) – Time window which will be shown for the spikes and self.second.
- **trace** (*double, optional (default: 5.)*) – Interval of time (ms) over which the data is overlayed in red.
- **show_spikes** (*bool, optional (default: True)*) – Whether a spike trajectory should be displayed on the network.
- **sort_neurons** (*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", 'B2') or a user-defined list of sorted neuron ids. Sorting is performed by increasing value of the *sort_neurons* property from bottom to top inside each group.
- ****kwargs** (*dict, optional (default: {})*) – Optional arguments such as 'make_rate', or all arguments for the *nngt.plot.draw_network()*.

nngt.plot.**betweenness_distribution**(*network*, *btype='both'*, *weights=False*, *nodes=None*, *logx=False*, *logy=False*, *num_nbins=None*, *num_ebins=None*, *axes=None*, *colors=None*, *norm=False*, *legend_location='right'*, *title=None*, *show=True*, ***kwargs*)

Plotting the betweenness distribution of a graph.

Changed in version 2.5.0: Added *title* argument.

Parameters

- **graph** (*Graph* or subclass) – the graph to analyze.
- **btype** (*string, optional (default: "both")*) – type of betweenness to display ("node", "edge" or "both")
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **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.
- **num_nbins** (*int or 'auto', optional (default: None):*) – Number of bins used to sample the node distribution. Defaults to *max(num_nodes / 50., 10)*.
- **num_ebins** (*int or 'auto', optional (default: None):*) – Number of bins used to sample the edge distribution. Defaults to *max(num_edges / 500., 10)* ('auto' method will be slow).
- **axes** (list of `matplotlib.axis.Axis`, optional (default: new ones)) – Axes which should be used to plot the histogram, if None, new ones are created.

- **legend_location** (*str, optional (default; 'right')*) – Location of the legend.
- **title** (*str, optional (default: auto-generated)*) – Title of the axis.
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.

nngt.plot.**chord_diagram**(*network*, *weights=True*, *names=None*, *order=None*, *width=0.1*, *pad=2.0*, *gap=0.03*, *chordwidth=0.7*, *axis=None*, *colors=None*, *cmap=None*, *alpha=0.7*, *use_gradient=False*, *chord_colors=None*, *show=False*, ***kwargs*)

Plot a chord diagram.

Parameters

- **network** (a *nngt.Graph* object) – Network used to plot the chord diagram.
- **weights** (*bool or str, optional (default: 'weight' attribute)*) – Weights used to plot the connections.
- **names** (*str or list of str, optional (default: no names)*) – Names of the nodes that will be displayed, either a node attribute or a custom list (must be ordered following the nodes' indices).
- **order** (*list, optional (default: order of the matrix entries)*) – Order in which the arcs should be placed around the trigonometric circle.
- **width** (*float, optional (default: 0.1)*) – Width/thickness of the ideogram arc.
- **pad** (*float, optional (default: 2)*) – Distance between two neighboring ideogram arcs. Unit: degree.
- **gap** (*float, optional (default: 0.03)*) – Distance between the arc and the beginning of the cord.
- **chordwidth** (*float, optional (default: 0.7)*) – Position of the control points for the chords, controlling their shape.
- **axis** (*matplotlib axis, optional (default: new axis)*) – Matplotlib axis where the plot should be drawn.
- **colors** (list, optional (default: from *cmap*)) – List of user defined colors or floats.
- **cmap** (*str or colormap object (default: viridis)*) – Colormap that will be used to color the arcs and chords by default. See *chord_colors* to use different colors for chords.
- **alpha** (*float in [0, 1], optional (default: 0.7)*) – Opacity of the chord diagram.
- **use_gradient** (*bool, optional (default: False)*) – Whether a gradient should be use so that chord extremities have the same color as the arc they belong to.
- **chord_colors** (*str, or list of colors, optional (default: None)*) – Specify color(s) to fill the chords differently from the arcs. When the keyword is not used, chord colors default to the colomap given by *colors*. Possible values for *chord_colors* are:
  - a single color (do not use an RGB tuple, use hex format instead), e.g. "red" or "#ff0000"; all chords will have this color
  - a list of colors, e.g. `["red", "green", "blue"]`, one per node (in this case, RGB tuples are accepted as entries to the list). Each chord will get its color from its associated source node, or from both nodes if *use_gradient* is True.
- **show** (*bool, optional (default: False)*) – Whether the plot should be displayed immediately via an automatic call to *plt.show()*.

- **kwargs** (*keyword arguments*) – Available kwargs are:

Name	Type	Purpose and possible values
fontcolor	str or list	Color of the names
fontsize	int	Size of the font for names
rotate_names	(list of) bool(s)	Rotate names by 90°
sort	str	Either "size" or "distance"
zero_entry_size	float	Size of zero-weight reciprocal

nngt.plot.**compare_population_attributes**(*network*, *attributes*, *nodes=None*, *reference_nodes=None*, *num_bins='auto'*, *reference_color='gray'*, *title=None*, *logx=False*, *logy=False*, *show=True*, ***kwargs*)

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: 'auto')*) – 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. Defaults to unsupervised Bayesian blocks method.
- **logx** (*bool or list, optional (default: False)*) – Use log-spaced bins.
- **logy** (*bool or list, optional (default: False)*) – use logscale for the node count.

nngt.plot.**correlation_to_attribute**(*network*, *reference_attribute*, *other_attributes*, *attribute_type='node'*, *nodes=None*, *edges=None*, *fig=None*, *title=None*, *show=True*)

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

Changed in version 2.0: Added *fig* argument.

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"
  - "in-strength", "out-strength", "total-strength"
  - "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*) – Attributes that will be compared to the reference.
- **attribute_type** (*str, optional (default: 'node')*) – Whether we are dealing with 'node' or 'edge' attributes
- **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.
- **edges** (*list, optional (default: all edges)*) – Edges for which the attributes should be returned.
- **fig** (`matplotlib.figure.Figure`, optional (default: new Figure)) – Figure to which the plot should be added.
- **title** (*str, optional (default: automatic).*) – Custom title, use "" to remove the automatic title.
- **show** (*bool, optional (default: True)*) – Whether the plot should be displayed immediately.

nngt.plot.**degree_distribution**(*network*, *deg_type='total'*, *nodes=None*, *num_bins='doane'*, *weights=False*, *logx=False*, *logy=False*, *axis=None*, *colors=None*, *norm=False*, *show=True*, *title=None*, ***kwargs*)

Plotting the degree distribution of a graph.

Changed in version 2.5.0: Removed unused *axis_num* argument.

Parameters

- **graph** (*Graph* or subclass) – The graph to analyze.
- **deg_type** (*string or N-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** (*str, int or N-tuple, optional (default: 'doane'):*) – Number of bins used to sample the distribution. Defaults to 'doane'. Use to 'auto' for numpy automatic selection or 'bayes' for unsupervised Bayesian blocks method.
- **weights** (*bool or str, optional (default: binary edges)*) – Whether edge weights should be considered; if `None` or `False` then use binary edges; if `True`, uses the 'weight' edge attribute, otherwise uses any valid edge attribute required.
- **logx** (*bool, optional (default: False)*) – Use log-spaced bins.
- **logy** (*bool, optional (default: False)*) – Use logscale for the degree count.
- **axis** (`matplotlib.axes.Axes` instance, optional (default: new one)) – Axis which should be used to plot the histogram, if None, a new one is created.
- **colors** (*(list of) matplotlib colors, optional (default: from palette)*) – Colors associated to each degree type.
- **title** (*str, optional (default: no title)*) – Title of the axis.
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.
- ****kwargs** (keyword arguments for `matplotlib.axes.Axes.bar()`.)

nngt.plot.**draw_network**(*network*, *nsize='total-degree'*, *ncolor=None*, *nshape='o'*, *esize=None*, *ecolor='k'*, *curved_edges=False*, *threshold=0.5*, *decimate_connections=None*, *spatial=True*, *restrict_sources=None*, *restrict_targets=None*, *restrict_nodes=None*, *restrict_edges=None*, *show_environment=True*, *fast=False*, *size=(600, 600)*, *xlims=None*, *ylims=None*, *dpi=75*, *axis=None*, *colorbar=False*, *cb_label=None*, *layout=None*, *show=False*, ***kwargs*)

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", "in/out/total-strength", or "betweenness".
- **ncolor** (*float, array of floats or string, optional*) – 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 or "in/out/total-degree", "betweenness" and "group". Default to red or one color per group in the graph if not specified.
- **nshape** (*char, array of chars, or groups, optional (default: "o")*) – Shape of the nodes (see Matplotlib markers). When using groups, they must be pairwise disjoint; markers will be selected iteratively from the matplotlib default 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** (*str, char, float or array, optional (default: "k")*) – Edge color. If ecolor="groups", edges color will depend on the source and target groups, i.e. only edges from and toward same groups will have the same color.
- **curved_edges** (*bool, optional (default: False)*) – Whether the edges should be curved or straight.
- **threshold** (*float, optional (default: 0.5)*) – Size under which edges are not plotted.
- **decimate_connections** (*int, optional (default: keep all connections)*) – Plot only one connection every *decimate_connections*. Use -1 to hide all edges.
- **spatial** (*bool, optional (default: True)*) – If True, use the neurons' positions to draw them.
- **restrict_sources** (*str, group, or list, optional (default: all)*) – Only draw edges starting from a restricted set of source nodes.
- **restrict_targets** (*str, group, or list, optional (default: all)*) – Only draw edges ending on a restricted set of target nodes.
- **restrict_nodes** (*str, group, or list, optional (default: plot all nodes)*) – Only draw a subset of nodes.
- **restrict_edges** (*list of edges, optional (default: all)*) – Only draw a subset of edges.
- **show_environment** (*bool, optional (default: True)*) – Plot the environment if the graph is spatial.
- **fast** (*bool, optional (default: False)*) – Use a faster algorithm to plot the edges. Zooming on the drawing made using this method leaves the size of the nodes and edges unchanged, it is therefore not recommended when size consistency matters, e.g. for some spatial representations.
- **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).

- **axis** (*matplotlib axis, optional (default: create new axis)*) – Axis on which the network will be plotted.
- **colorbar** (*bool, optional (default: False)*) – Whether to display a colorbar for the node colors or not.
- **cb_label** (*str, optional (default: None)*) – A label for the colorbar.
- **layout** (*str, optional (default: random or spatial positions)*) – Name of a standard layout to structure the network. Available layouts are: "circular" or "random". If no layout is provided and the network is spatial, then node positions will be used by default.
- **show** (*bool, optional (default: True)*) – Display the plot immediately.
- ****kwargs** (*dict*) – Optional keyword arguments.

Name	Type	Purpose and possible values
node_cmap	str	"magma" for continuous variables and "Set1" for groups)
title	str	Title of the plot
max_*	float	Maximum value for *nsize* or *esize*
min_*	float	Minimum value for *nsize* or *esize*
nalpha	float	Node opacity in [0, 1]`, default 1
ealpha	float	Edge opacity, default 0.5
*border_color	color	or edges (e). Default to black.
*border_width	float	(e). Default to .5 for nodes and .3 for edges (if *fast* is False).
simple_nodes	bool	are always the same size) or patches (change size with zoom).

nngt.plot.**edge_attributes_distribution**(*network*, *attributes*, *edges=None*, *num_bins='auto'*, *logx=False*, *logy=False*, *norm=False*, *title=None*, *axtitles=None*, *colors=None*, *axes=None*, *show=True*, ***kwargs*)

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

Changed in version 2.5.0: Added *axtitles* and *axes* arguments.

Parameters

- **network** (*Graph*) – The graph where the *nodes* belong.
- **attributes** (*str or list*) – Attributes which should be returned (e.g. "betweenness", "delay", "weight").
- **edges** (*list, optional (default: all edges)*) – Edges for which the attributes should be returned.
- **num_bins** (*int or list, optional (default: 'auto')*) – Number of bins to plot the distributions. If only one int is provided, it is used for all attributes, otherwise a list containing one int per attribute in *attributes* is required. Defaults to unsupervised Bayesian blocks method.
- **logx** (*bool or list, optional (default: False)*) – Use log-spaced bins.
- **logy** (*bool or list, optional (default: False)*) – use logscale for the node count.
- **norm** (*bool, optional (default: False)*) – Whether the histogram should be normed such that the sum of the counts is 1.
- **title** (*str, optional (default: no title)*) – Title of the figure.
- **axtitles** (*list of str, optional (default: auto-generated)*) – Titles of the axes. Use "" or False to turn them of.
- **colors** (*(list of) matplotlib colors, optional (default: from palette)*) – Colors associated to each degree type.

- **axes** (list of `matplotlib.axis.Axis`, optional (default: new ones)) – Axess which should be used to plot the histograms, if None, a new axis is created for each attribute.
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.
- ****kwargs** (keyword arguments for `matplotlib.axes.Axes.bar()`.)

nngt.plot.**hive_plot**(*network*, *radial*, *axes=None*, *axes_bins=None*, *axes_range=None*, *axes_angles=None*, *axes_labels=None*, *axes_units=None*, *intra_connections=True*, *highlight_nodes=None*, *highlight_edges=None*, *nsize=None*, *esize=None*, *max_nsize=10*, *max_esize=1*, *axes_colors=None*, *edge_colors=None*, *edge_alpha=0.05*, *nborder_color='k'*, *nborder_width=0.2*, *show_names=True*, *show_circles=False*, *axis=None*, *tight=True*, *show=False*)

Draw a hive plot of the graph.

Note: For directed networks, the direction of intra-axis connections is counter-clockwise. For inter-axes connections, the default edge color is closest to the color of the source group (i.e. from a red group to a blue group, edge color will be a reddish violet , while from blue to red, it will be a blueish violet).

Parameters

- **network** (*Graph*) – Graph to plot.
- **radial** (*str, list of str or array-like*) – Values that will be used to place the nodes on the axes. Either one identical property is used for all axes (traditional hive plot) or one radial coordinate per axis is used (custom hive plot). If radial is a string or a list of strings, then these must correspond to the names of node attributes stored in the graph.
- **axes** (*str, or list of str, optional (default: one per radial coordinate)*) – Name of the attribute(s) that will be used to make each of the axes (i.e. each group of nodes). This can be either "groups" if the graph has a structure or is a *Network*, a list of (Meta)Group names, or any (list of) node attribute(s). If a single node attribute is used, *axes_bins* must be provided to make one axis for each range of values. If there are multiple radial coordinates, then leaving *axes* blanck will plot all nodes on each of the axes (one per radial coordinate).
- **axes_bins** (*int or array-like, optional (default: all nodes on each axis)*) – Required if there is a single radial coordinate and a single axis entry: provides the bins that will be used to separate the nodes into groups (one per axis). For N axes, there must therefore be N + 1 entries in *axes_bins*, or *axis_bins* must be equal to N, in which case the nodes are separated into N evenly sized bins.
- **axes_units** (*str, optional*) – Units used to scale the axes. Either "native" to have them scaled between the minimal and maximal radial coordinates among all axes, "rank", to use the min and max ranks of the nodes on all axes, or "normed", to have each axis go from zero (minimal local radial coordinate) to one (maximal local radial coordinate). "native" is the default if there is a single radial coordinate, "normed" is the default for multiple coordinates.
- **axes_angles** (*list of angles, optional (default: automatic)*) – Angles for each of the axes, by increasing degree. If *intra_connections* is True, then angles of duplicate axes must be adjacent, e.g. `[a1, a1bis, a2, a2bis, a3, a3bis]`.
- **axes_labels** (*str or list of str, optional*) – Label of each axis. For binned axes, it can be automatically formatted via the three entries `{name}`, `{start}`, `{stop}`. E.g. "{name} in [{start}, {stop}]" would give "CC in [0, 0.2]" for a first axis and "CC in [0.2, 0.4]" for a second axis.

- **intra_connections** (*bool, optional (default: True)*) – Show connections between nodes belonging to the same axis. If true, then each axis is duplicated to display intra-axis connections.
- **highlight_nodes** (*list of nodes, optional (default: all nodes)*) – Highlight a subset of nodes and their connections, all other nodes and connections will be gray.
- **highlight_edges** (*list of edges, optional (default: all edges)*) – Highlight a subset of edges; all other connections will be gray.
- **nsize** (*float, str, or array-like, optional (default: automatic)*) – Size of the nodes on the axes. Either a fixed size, the name of a node attribute, or a list of user-defined values.
- **esize** (*float or str, optional (default: 1)*) – Size of the edges. Either a fixed size or the name of an edge attribute.
- **max_nsize** (*float, optional (default: 10)*) – Maximum node size if *nsize* is an attribute or a list of user-defined values.
- **max_esize** (*float, optional (default: 1)*) – Maximum edge size if *esize* is an attribute.
- **axes_colors** (*valid matplotlib color/colormap, optional (default: Set1)*) – Color associated to each axis.
- **nborder_color** (*matplotlib color, optional (default: "k")*) – Color of the node's border. or floats in [0, 1] defining the position in the palette.
- **nborder_width** (*float, optional (default: 0.2)*) – Width of the border.
- **edge_colors** (*valid matplotlib color/colormap, optional (default: auto)*) – Color of the edges. By default it is the intermediate color between two axes colors. To provide custom colors, they must be provided as a dictionnary of axes edges `{(0, 0): "r", (0, 1): "g", (1, 0): "b"}` with default color being black.
- **edge_alpha** (*float, optional (default: 0.05)*) – Edge opacity.
- **show_names** (*bool, optional (default: True)*) – Show axes names and properties.
- **show_circles** (*bool, optional (default: False)*) – Show the circles associated to the maximum value of each axis.
- **axis** (*matplotlib axis, optional (default: create new axis)*) – Axis on which the network will be plotted.
- **tight** (*bool, optional (default: True)*) – Set figure layout to tight (set to False if plotting multiple axes on a single figure).
- **show** (*bool, optional (default: True)*) – Display the plot immediately.

nngt.plot.**library_draw**(*network*, *nsize='total-degree'*, *ncolor=None*, *nshape='o'*, *nborder_color='k'*, *nborder_width=0.5*, *esize=1.0*, *ecolor='k'*, *ealpha=0.5*, *curved_edges=False*, *threshold=0.5*, *decimate_connections=None*, *spatial=True*, *restrict_sources=None*, *restrict_targets=None*, *restrict_nodes=None*, *restrict_edges=None*, *show_environment=True*, *size=(600, 600)*, *xlims=None*, *ylims=None*, *dpi=75*, *axis=None*, *colorbar=False*, *show_labels=False*, *layout=None*, *show=False*, ***kwargs*)

Draw a given *Graph* using the underlying library's drawing functions.

New in version 2.0.

Warning: When using igraph or graph-tool, if you want to use the *axis* argument, then you must first switch the matplotlib backend to its cairo version using e.g. `plt.switch_backend("Qt5Cairo")` if your normal backend is Qt5 ("Qt5Agg").

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 or "in/out/total-degree", "betweenness" and "group". Default to red or one color per group in the graph if not specified.
- **nshape** (*char, array of chars, or groups, optional (default: "o")*) – Shape of the nodes (see Matplotlib markers). When using groups, they must be pairwise disjoint; markers will be selected iteratively from the matplotlib default 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** (*str, char, float or array, optional (default: "k")*) – Edge color. If ecolor="groups", edges color will depend on the source and target groups, i.e. only edges from and toward same groups will have the same color.
- **threshold** (*float, optional (default: 0.5)*) – Size under which edges are not plotted.
- **decimate_connections** (*int, optional (default: keep all connections)*) – Plot only one connection every *decimate_connections*. Use -1 to hide all edges.
- **spatial** (*bool, optional (default: True)*) – If True, use the neurons' positions to draw them.
- **restrict_sources** (*str, group, or list, optional (default: all)*) – Only draw edges starting from a restricted set of source nodes.
- **restrict_targets** (*str, group, or list, optional (default: all)*) – Only draw edges ending on a restricted set of target nodes.
- **restrict_nodes** (*str, group, or list, optional (default: plot all nodes)*) – Only draw a subset of nodes.
- **restrict_edges** (*list of edges, optional (default: all)*) – Only draw a subset of edges.
- **show_environment** (*bool, optional (default: True)*) – Plot the environment if the graph is spatial.
- **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).
- **colorbar** (*bool, optional (default: False)*) – Whether to display a colorbar for the node colors or not.

- **axis** (*matplotlib axis, optional (default: create new axis)*) – Axis on which the network will be plotted.
- **layout** (*str, optional (default: library-dependent or spatial positions)*) – Name of a standard layout to structure the network. Available layouts are: "circular", "spring-block", "random". If no layout is provided and the network is spatial, then node positions will be used by default.
- **show** (*bool, optional (default: True)*) – Display the plot immediately.
- ****kwargs** (*dict*) – Optional keyword arguments.

Name	Type	Purpose and possible values
node_cmap	str	"magma" for continuous variables and "Set1" for groups)
title	str	Title of the plot
max_*	float	Maximum value for *nsize* or *esize*
min_*	float	Minimum value for *nsize* or *esize*
annotate	bool	Use annotations to show node information (default: True)
annotations	str or list	such as a node attribute or a list of values. (default: node id)

nngt.plot.**node_attributes_distribution**(*network*, *attributes*, *nodes=None*, *num_bins='auto'*, *logx=False*, *logy=False*, *norm=False*, *title=None*, *axtitles=None*, *colors=None*, *axes=None*, *show=True*, ***kwargs*)

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

Changed in version 2.5.0: Added *axtitles* and *axes* arguments.

Parameters

- **network** (*Graph*) – The graph where the *nodes* belong.
- **attributes** (*str or list*) – Attributes which should be returned, among: * any user-defined node attribute * "betweenness" * "clustering" * "closeness" * "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: 'auto')*) – Number of bins to plot the distributions. If only one int is provided, it is used for all attributes, otherwise a list containing one int per attribute in *attributes* is required. Defaults to unsupervised Bayesian blocks method.
- **logx** (*bool or list, optional (default: False)*) – Use log-spaced bins.
- **logy** (*bool or list, optional (default: False)*) – use logscale for the node count.
- **norm** (*bool, optional (default: False)*) – Whether the histogram should be normed such that the sum of the counts is 1.
- **title** (*str, optional (default: no title)*) – Title of the figure.
- **axtitles** (*list of str, optional (default: auto-generated)*) – Titles of the axes. Use "" or False to turn them of.
- **colors** (*(list of) matplotlib colors, optional (default: from palette)*) – Colors associated to each degree type.
- **axes** (list of matplotlib.axis.Axis, optional (default: new ones)) – Axess which should be used to plot the histograms, if None, a new axis is created for each attribute.
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.
- ****kwargs** (keyword arguments for matplotlib.axes.Axes.bar().)

nngt.plot.**palette_continuous**(*numbers=None*)

nngt.plot.**palette_discrete**(*numbers=None*)

Known bugs

- Calling `nngt.geospatial` or `nngt.simulation` directly in python causes a `ValueError: module object substituted in sys.modules during a lazy load` which I don't know how to avoided... use `from nngt.geospatial/simulation import whatever_you_want` or `import nngt.geospatial/simulation as ng/ns` instead.
- See the issue trackers on Codeberg or GitHub for up-to-date lists.

2.3 Tutorial

This page provides a step-by-step walkthrough of the basic features of NNGT.

To run this tutorial, it is recommended to use either IPython or Jupyter, since they will provide automatic autocompletion of the various functions, as well as easy access to the docstring help.

First, import the NNGT package:

```
>>> import nngt
```

Then, you will be able to use the help from IPython by typing, for instance:

```
>>> nngt.Graph?
```

In Jupyter, the docstring can be viewed using `Shift` + `Tab`.

The source file for the tutorial can be found here: doc/examples/introductory_tutorial.py.

Note: For a list of example files, see the 'examples' directory on GitHub.

For specific tutorials see also:

- *Graph generation*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Activity analysis*
- *Properties of graph components*

Content:

2.3.1 NNGT properties and configuration

Upon loading, NNGT will display its current configuration, e.g.:

```
# ----------- #
# NNGT loaded #
# ----------- #
Graph library:  igraph 0.7.1
Multithreading: True (1 thread)
MPI:            False
Plotting:       True
NEST support:   NEST 2.14.0
Shapely:        1.6.1
SVG support:    True
DXF support:    False
Database:       False
```

Let's walk through this configuration:

- the backend used here is `igraph`, so all graph-theoretical tools will be derived from those of the igraph library and we're using version 0.7.1.
- Multithreaded algorithms will be used, currently running on only one thread (see *Parallelism* for more details)
- MPI algorithms are not in use (you cannot use both MT and MPI at the same time)
- Plotting is available because the matplotlib library is installed
- NEST is installed on the machine (version 2.14), so NNGT automatically loaded it
- Shapely is also available, which allows the creation of complex structures for space-embedded networks (see *Geometry module* for more details)
- Importing SVG files to generate spatial structures is possible, meaning that the svg.path module is installed.
- Importing DXF files to generate spatial structures is not possible because the dxfgrabber module is not installed.
- Using the database is not possible because peewee is not installed.

In general, most of NNGT options can be found/set through the *`get_config()`*/*`set_config()`* functions, or made permanent by modifying the `~/.nngt/nngt.conf` configuration file.

2.3.2 The `Graph` object

Basic functions

Let's create an empty *Graph*:

```
g = nngt.Graph()
```

We can then add some nodes to it

```
g.new_node(10)            # create nodes 0, 1, ... to 9
print(g.node_nb(), '\n')  # returns 10
```

And create edges between these nodes:

```
g.new_edge(1, 4)          # create one connection going from 1 to 4
print(g.edge_nb())        # returns 1
g.new_edges([(0, 3), (5, 9), (9, 3)])
print(g.edge_nb(), '\n')  # returns 4
```

Node and edge attributes

Adding a node with specific attributes:

```
g2 = nngt.Graph()

# add a new node with attributes
attributes = {
    'size': 2.,
    'color': 'blue',
    'a': 5,
    'blob': []
}

attribute_types = {
    'size': 'double',
    'color': 'string',
    'a': 'int',
    'blob': 'object'
}

g2.new_node(attributes=attributes, value_types=attribute_types)
print(g2.node_attributes, '\n')
```

By default, nodes that are added without specifying attribute values will get their attributes filled with default values which depend on the type:

- `NaN` for "double"
- 0 for "int"
- "" for "string"
- `None` for "object"

```
g2.new_node(2)
# for a double attribute like 'size', default value is NaN
print(g2.get_node_attributes(name="size"))
# for a string attribute like 'color', default value is ""
print(g2.get_node_attributes(name="color"))
# for an int attribute like 'a', default value is 0
print(g2.get_node_attributes(name='a'))
# for an object attribute like 'blob', default value is None
print(g2.get_node_attributes(name='blob'), '\n')
```

Adding several nodes and attributes at the same time:

```
g2.new_node(3, attributes={'size': [4., 5., 1.], 'color': ['r', 'g', 'b']},
            value_types={'size': 'double', 'color': 'string'})
print(g2.node_attributes['size'])
print(g2.node_attributes['color'], '\n')
```

Attributes can also be created afterwards:

```
import numpy as np
g3 = nngt.Graph(nodes=100)
g3.new_node_attribute('size', 'double',
                      values=np.random.uniform(0, 20, 100))
print(g3.node_attributes['size'][:5], '\n')
```

All the previous techniques can also be used with *new_edge()* or *new_edges()*, and *new_edge_attribute()*. Note that attributes can also be set selectively:

```
edges = g3.new_edges(np.random.randint(0, 50, (10, 2)), ignore_invalid=True)
g3.new_edge_attribute('rank', 'int')
g3.set_edge_attribute('rank', val=2, edges=edges[:3, :])
print(g3.edge_attributes['rank'], '\n')
```

2.3.3 Generating and analyzing more complex networks

NNGT provides a whole set of methods to connect nodes in specific fashions inside a graph. These methods are present in the *nngt.generation* module, and the network properties can then be plotted and analyzed via the tools present in the *nngt.plot* and *nngt.analysis* modules.

```
from nngt import generation as ng
from nngt import analysis as na
from nngt import plot as nplt
```

NNGT implements some fast generation tools to create several of the standard networks, such as Erdős-Rényi:

```
g = ng.erdos_renyi(nodes=1000, avg_deg=100)

if nngt.get_config("with_plot"):
    nplt.degree_distribution(g, ('in', 'total'), show=False)

print("Clustering ER: {}".format(na.global_clustering(g)))
```

More heterogeneous networks, with scale-free degree distribution (but no correlations like in Barabasi-Albert networks and user-defined exponents) are also implemented:

```
g = ng.random_scale_free(1.8, 3.2, nodes=1000, avg_deg=100)

if nngt.get_config("with_plot"):
    nplt.degree_distribution(g, ('in', 'out'), num_bins=30, logx=True,
                             logy=True, show=True)

print("Clustering SF: {}".format(na.global_clustering(g)))
```

For more details, see the full page on *Graph generation*.

2.3.4 Using random numbers

By default, NNGT uses the *numpy* random-number generators (RNGs) which are seeded automatically when *numpy* is loaded.

However, you can seed the RNGs manually using the following command:

```
nngt.set_config("msd", 0)
```

which will seed the master seed to 0 (or any other value you enter). Once seeded manually, a NNGT script will always give the same results provided the same number of thread is being used.

Indeed, when using multithreading, sub-RNGs are used (one per thread). By default, these RNGs are seeded from the master seed as $msd + n + 1$ where n is the thread number, starting from zero. If needed, these sub-RNGs can also be seeded manually using (for 4 threads)

```
nngt.set_config("seeds", [1, 2, 3, 4])
```

Warning: When using NEST, the simulator's RNGs must be seeded separately using the NEST commands; see the NEST user manual for details.

2.3.5 Structuring nodes: `Group` and `Structure`

The *`Group`* allows the creation of nodes that belong together. You can then make a complex *`Structure`* from these groups and connect them with specific connectivities using the *`connect_groups()`* function.

```
''' ------------------------ #
# Creating a structured graph #
# ------------------------ '''

room1 = nngt.Group(25)
room2 = nngt.Group(50)
room3 = nngt.Group(40)
room4 = nngt.Group(35)

names = ["R1", "R2", "R3", "R4"]

struct = nngt.Structure.from_groups((room1, room2, room3, room4), names)
```

(continues on next page)

(continued from previous page)

```
g = nngt.Graph(structure=struct)

for room in struct:
    nngt.generation.connect_groups(g, room, room, "all_to_all")

nngt.generation.connect_groups(g, (room1, room2), struct, "erdos_renyi",
                               avg_deg=10, ignore_invalid=True)

nngt.generation.connect_groups(g, room3, room1, "erdos_renyi", avg_deg=20)

nngt.generation.connect_groups(g, room4, room3, "erdos_renyi", avg_deg=10)

if nngt.get_config("with_plot"):
    # chord diagram
    sg = g.get_structure_graph()

    nngt.plot.chord_diagram(sg, names="name", sort="distance",
                            use_gradient=True, show=True)

    # spring-block layout
    nngt.plot.library_draw(g, node_cmap="viridis", show=True)
```

For more details, see the full page on *Groups, structures, and neuronal populations*.

2.3.6 The same with neurons: `NeuralGroup`, `NeuralPop`

The *NeuralGroup* allows the creation of nodes that belong together. You can then make a population from these groups and connect them with specific connectivities using the *connect_groups()* function.

```
''' ---------------------------------- #
# Complete groups for NEST simulations #
# ---------------------------------- '''

# to make a complete neuronal group, one must include a valid neuronal type,
# model and (optionally) associated parameters

pyr = NeuralGroup(800, neuron_type=1, neuron_model="iaf_psc_alpha",
                  neuron_param={"tau_m": 50.}, name="pyramidal_cells")

fsi = NeuralGroup(200, neuron_type=-1, neuron_model="iaf_psc_alpha",
                  neuron_param={"tau_m": 20.},
                  name="fast_spiking_interneurons")


''' -------------------------- #
# Creating neuronal populations #
# -------------------------- '''

pop = NeuralPop.from_groups((pyr, fsi))
```

(continues on next page)

(continued from previous page)

```
# making populations from scratch
pop = nngt.NeuralPop(with_models=False)                # empty population
pop.create_group(200, "first_group")                   # create excitatory group
pop.create_group(5, "second_group", neuron_type=-1)    # create inhibitory group
```

For more details, see the full page on *Groups, structures, and neuronal populations*.

2.3.7 Real neuronal networks and NEST interaction: the `Network`

Besides connectivity, the main interest of the *NeuralGroup* is that you can pass it the biological properties that the neurons belonging to this group will share.

Since we are using NEST, these properties are:

- the model's name
- its non-default properties
- the synapses that the neurons have and their properties
- the type of the neurons (1 for excitatory or -1 for inhibitory)

```
''' Create groups with different parameters '''
# adaptive spiking neurons
base_params = {
    'E_L': -60., 'V_th': -58., 'b': 20., 'tau_w': 100.,
    'V_reset': -65., 't_ref': 2., 'g_L': 10., 'C_m': 250.
}
# oscillators
params1, params2 = base_params.copy(), base_params.copy()
params1.update(
    {'E_L': -65., 'b': 40., 'I_e': 200., 'tau_w': 400., "V_th": -57.})
# bursters
params2.update({'b': 25., 'V_reset': -55., 'tau_w': 300.})

oscill = nngt.NeuralGroup(
    nodes=400, neuron_model='aeif_psc_alpha', neuron_type=1,
    neuron_param=params1)

burst = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', neuron_type=1,
    neuron_param=params2)

adapt = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', neuron_type=1,
    neuron_param=base_params)

synapses = {
    'default': {model: 'tsodyks2_synapse'},
    ('oscillators', 'bursters'): {model: 'tsodyks2_synapse', 'U': 0.6},
    ('oscillators', 'oscillators'): {model: 'tsodyks2_synapse', 'U': 0.7},
    ('oscillators', 'adaptive'): {model: 'tsodyks2_synapse', 'U': 0.5}
}
```

(continues on next page)

(continued from previous page)

```
'''
Create the population that will represent the neuronal
network from these groups
'''
pop = nngt.NeuralPop.from_groups(
    [oscill, burst, adapt],
    names=['oscillators', 'bursters', 'adaptive'], syn_spec=synapses)

'''
Create the network from this population,
using a Gaussian in-degree
'''
net = ng.gaussian_degree(
    100., 15., population=pop, weights=155., delays=5.)
```

Once this network is created, it can simply be sent to nest through the command: `gids = net.to_nest()`, and the NEST gids are returned.

In order to access the gids from each group, you can do:

```
oscill_gids = net.nest_gid[oscill.ids]
```

For more details to use NNGT with NEST, see *Interacting with the NEST simulator*.

2.3.8 Underlying graph objects and libraries

Starting with version 2.0 of NNGT, the library no longer uses inheritance but composition to provide access to the underlying graph object, which is stored in the *graph* attribute of the *Graph* class.

It can simply be accessed via:

```
g = nngt.Graph()

library_graph = g.graph
```

Using *graph* attribute, on can directly use functions of the underlying graph library (networkx, igraph, or graph-tool) if their equivalent is not yet provided in NNGT – see *Consistent tools for graph analysis* for implemented functions.

Warning: One notable exception to this behaviour relates to the creation and deletion of nodes or edges, for which you have to use the functions provided by NNGT. As a general rule, any operation that might alter the graph structure should be done through NNGT and never directly by calling functions or methods on the *graph* attribute.

Apart from this, you can use any analysis or drawing tool from the graph library.

Example using graph-tool

```
>>> import graph_tool as gt
>>> import matplotlib.pyplot as plt
>>> print(gt.centrality.closeness(g.graph))
>>> gt.draw.graph_draw(g.graph)
>>> nngt.plot.draw_network(g)
>>> plt.show()
```

Example using igraph

```
>>> import igraph as ig
>>> import matplotlib.pyplot as plt
>>> print(g.graph.closeness(mode='out'))
>>> ig.plot(g.graph)
>>> nngt.plot.draw_network(g)
>>> plt.show()
```

Example using networkx

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> print(nx.closeness_centrality(g.graph.reverse()))
>>> nx.draw(g.graph)
>>> nngt.plot.draw_network(g)
>>> plt.show()
```

Note: People testing these 3 codes will notice that all closeness results are different (though I made sure the functions of each libraries worked on the same outgoing edges)! This example is given voluntarily to remind you, when using these libraries, to check that they indeed compute what you think they do and what are the underlying hypotheses or definitions.

To avoid such issues and make sure that results are the same with all libraries, use the functions provided in *Consistent tools for graph analysis*.

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Parallelism*
- *Groups, structures, and neuronal populations*
- *Interacting with the NEST simulator*
- *Activity analysis*
- *Properties of graph components*

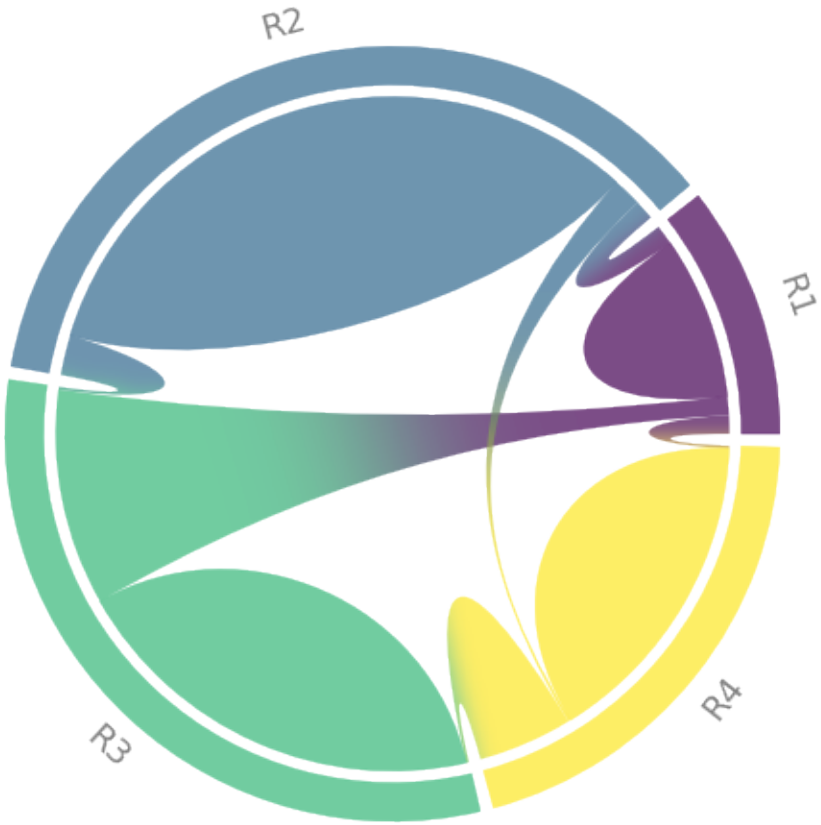
2.4 Gallery

This page contains a set of examples about different ways of visualizing graphs and their properties using NNGT.

2.4.1 Visualizing graph structures

The following examples show how to use NNGT to draw graphs in ways that make their structural properties stand out.

Chord diagram




```
import matplotlib.pyplot as plt

import nngt

plt.rcParams["figure.facecolor"] = (0, 0, 0, 0)

nngt.seed(0)


# create a structured graph
```

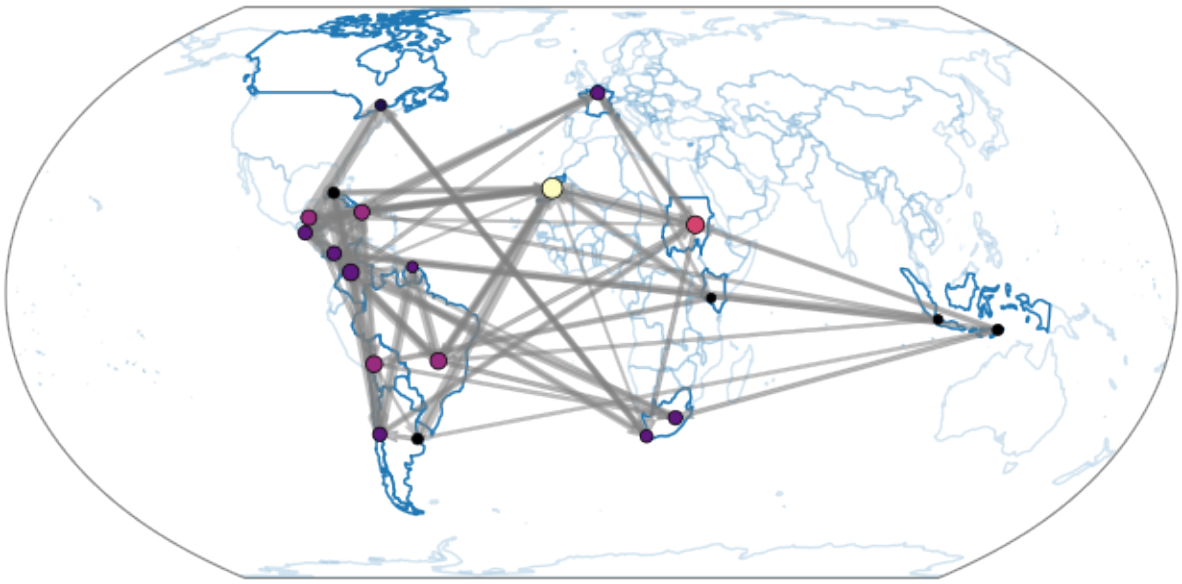
(continues on next page)

(continued from previous page)

```
room1 = nngt.Group(25)
room2 = nngt.Group(50)
room3 = nngt.Group(40)
room4 = nngt.Group(35)

names = ["R1", "R2", "R3", "R4"]

struct = nngt.Structure.from_groups((room1, room2, room3, room4), names)

g = nngt.Graph(structure=struct)

for room in struct:
    nngt.generation.connect_groups(g, room, room, "all_to_all")

nngt.generation.connect_groups(g, (room1, room2), struct, "erdos_renyi",
                               avg_deg=10, ignore_invalid=True)

nngt.generation.connect_groups(g, room3, room1, "erdos_renyi", avg_deg=20)

nngt.generation.connect_groups(g, room4, room3, "erdos_renyi", avg_deg=10)


# get the structure graph and plot

sg = g.get_structure_graph()

nngt.plot.chord_diagram(sg, names="name", sort="distance", fontcolor="grey",
                        use_gradient=True, show=True)
```

Total running time of the script: (0 minutes 1.553 seconds)

Geospatial networks



```
import os

import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import numpy as np

import nngt
import nngt.geospatial as ng


plt.rcParams.update({
    'axes.edgecolor': 'grey', 'xtick.color': 'grey', 'ytick.color': 'grey',
    "figure.facecolor": (0, 0, 0, 0), "axes.facecolor": (0, 0, 0, 0),
    "axes.labelcolor": "grey", "text.color": "grey"
})


nngt.seed(2)


# take random countries
```

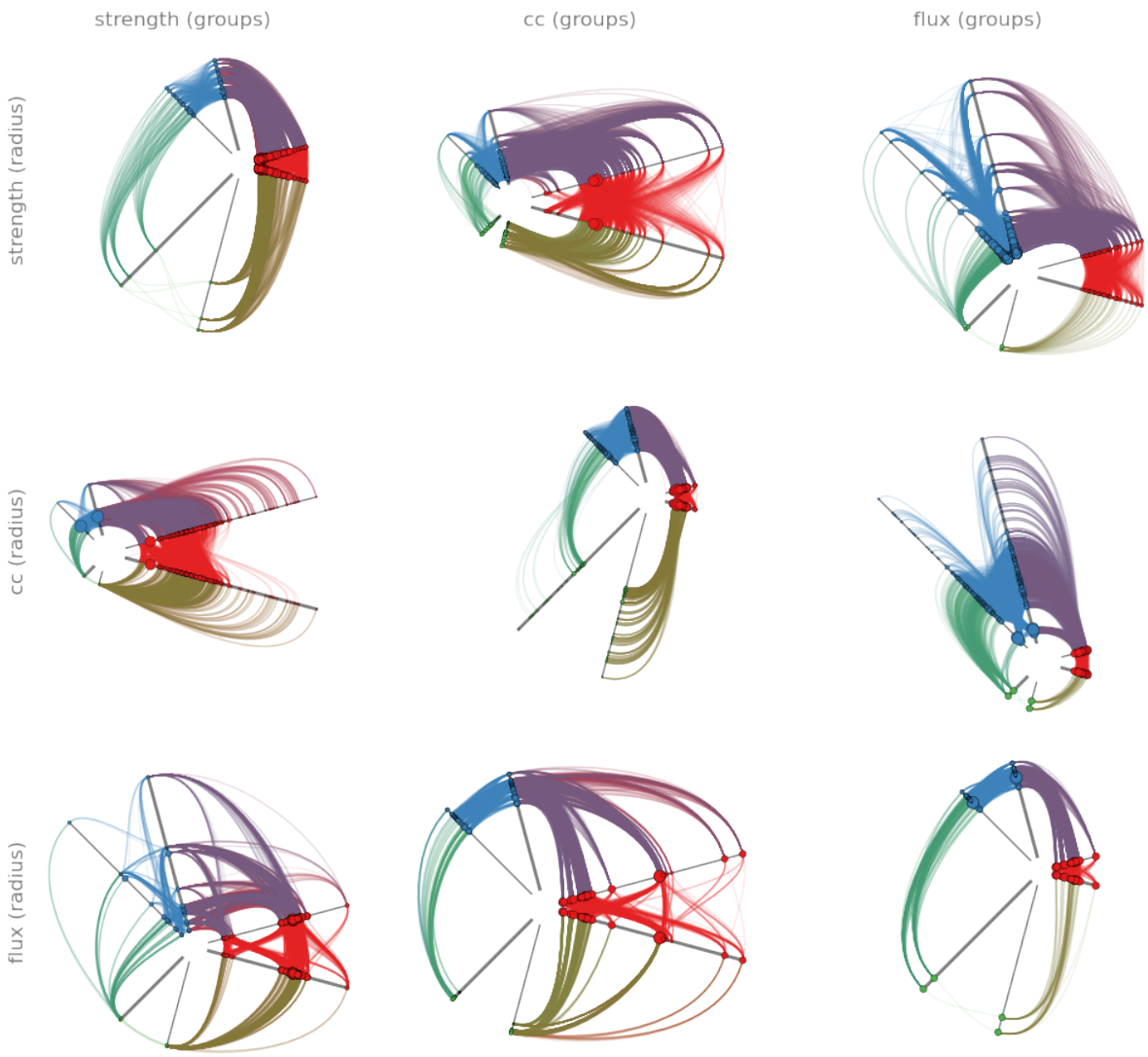
(continues on next page)

(continued from previous page)

```
num_nodes = 20

world = ng.maps["adaptive"]
units = nngt._rng.choice(50, num_nodes, replace=False)
codes = list(world.iloc[units].SU_A3)

# make random network
g = nngt.generation.erdos_renyi(nodes=num_nodes, avg_deg=3)

# add the A3 code for each country (that's the crucial part that will link
# the graph to the geospatial data)
g.new_node_attribute("code", "string", codes)

g.set_weights(nngt._rng.exponential(2, g.edge_nb()))

# plot using draw_map and the A3 codes stored in "code"
ng.draw_map(g, "code", ncolor="in-degree", esize="weight", threshold=0,
            ecolor="grey", proj=ccrs.EqualEarth(), max_nsize=20, show=False)

if nngt.get_config("with_plot"):
    plt.tight_layout()
    plt.show()
```

Total running time of the script: (0 minutes 0.357 seconds)

Hive plot panel




```
import inspect
from os.path import abspath, dirname

import matplotlib.pyplot as plt

import nngt


plt.rcParams.update({
    "figure.facecolor": (0, 0, 0, 0), "text.color": "grey"
})


dirpath  = dirname(inspect.getframeinfo(inspect.currentframe()).filename)
rootpath = abspath(dirpath + "/../../..")
```

(continues on next page)

(continued from previous page)

```
# load graph

g = nngt.load_from_file(rootpath + "/testing/Networks/rat_brain.graphml",
                        attributes=["weight"], cleanup=True,
                        attributes_types={"weight": float})


# prepare attributes

cc = nngt.analysis.local_clustering(g, weights="weight")

g.new_node_attribute("cc", "double", values=cc)

g.new_node_attribute("strength", "double",
                     values=g.get_degrees(weights="weight"))

flux = g.get_degrees("out") - g.get_degrees("in")

g.new_node_attribute("flux", "double", values=flux)


# figure parameters

cc_bins = [0, 0.1, 0.25, 0.6]

todo = ["strength", "cc", "flux"]
bins = [3, cc_bins, 3]


# make plot

fig, axes = plt.subplots(len(todo), len(todo), figsize=(10, 9))

for i in range(len(todo)):
    radial = todo[i]

    for j in range(len(todo)):
        ax_name = todo[j]
        ax_bins = bins[j]

        ax = axes[i, j]

        if i == 0:
            ax.set_title(ax_name + " (groups)")

        size = todo[list(set([0, 1, 2]).difference([i, j]))[0]]

        nngt.plot.hive_plot(
            g, radial, axes=ax_name, edge_alpha=0.1, nsize=size, max_nsize=50,
            axes_bins=ax_bins, axes_units="native", axis=ax, show_names=False)
```

(continues on next page)

(continued from previous page)

```
for i in range(len(todo)):
    fig.text(0.03, 0.83 - i*0.33, todo[i] + " (radius)", rotation=90,
             fontsize="large", va="center")

plt.tight_layout()

plt.show()
```

Total running time of the script: (0 minutes 15.511 seconds)

Layouts for topological representations

```
import os

import matplotlib as mpl
import matplotlib.pyplot as plt

import nngt


plt.rcParams.update({
    "figure.facecolor": (0, 0, 0, 0),
    "axes.labelcolor": "grey", "text.color": "grey"
})


nngt.seed(0)


# set matplotlib backend depending on the library
mpl_backend = mpl.get_backend()

if nngt.get_config("backend") in ("graph-tool", "igraph"):
    if mpl_backend.startswith("Qt4"):
        if mpl_backend != "Qt4Cairo":
            plt.switch_backend("Qt4Cairo")
    elif mpl_backend.startswith("Qt5"):
        if mpl_backend != "Qt5Cairo":
            plt.switch_backend("Qt5Cairo")
    elif mpl_backend.startswith("GTK"):
        if mpl_backend != "GTK3Cairo":
            plt.switch_backend("GTK3Cairo")
    else:
        plt.switch_backend("cairo")


# prepare figure and parameters

fig = plt.figure(figsize=(10, 8), constrained_layout=False)
```

(continues on next page)

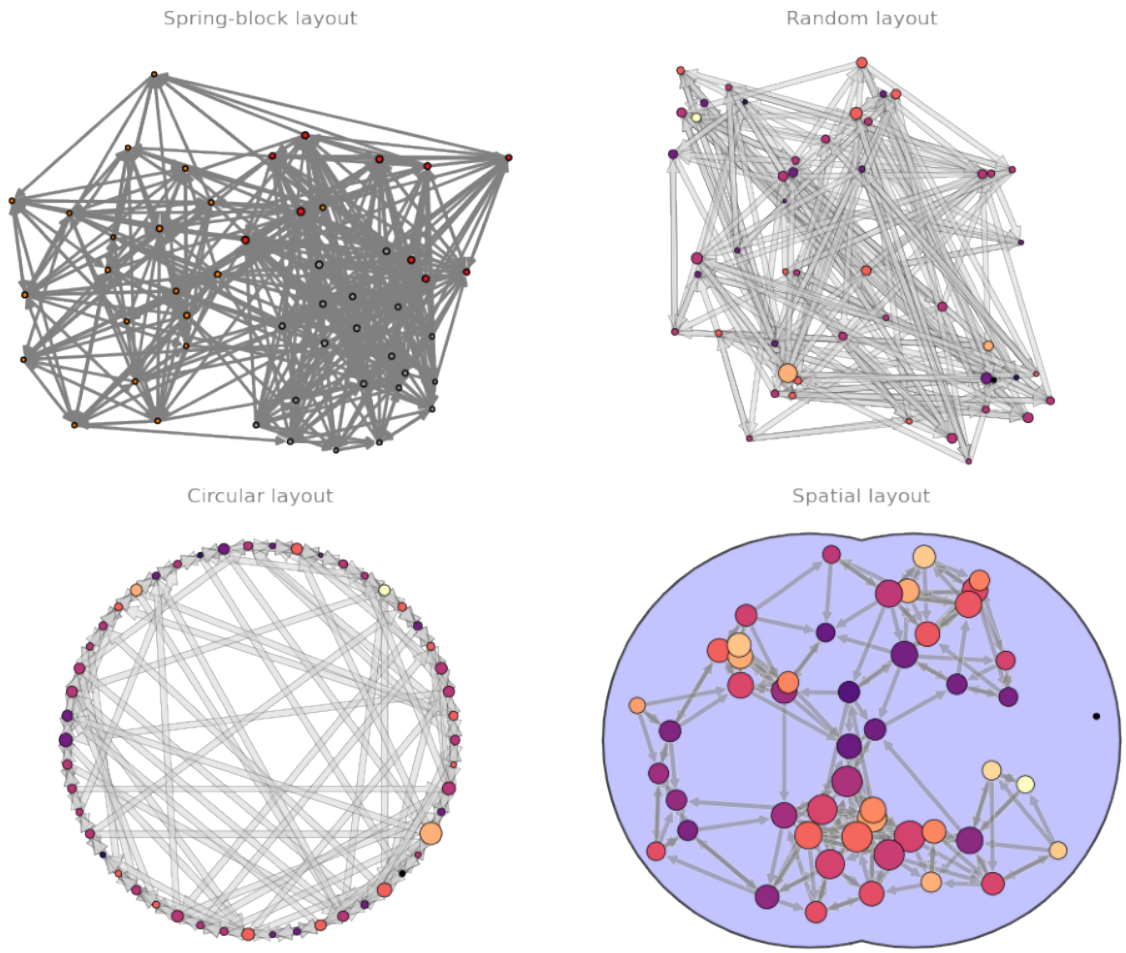
(continued from previous page)

```
gs = fig.add_gridspec(nrows=2, ncols=2, left=0, right=1, bottom=0, top=0.97,
                      wspace=0, hspace=0.05)

axes = [fig.add_subplot(gs[i, j]) for i in (0, 1) for j in (0, 1)]

num_nodes = 50


# spring-block layout for structured graph

room1 = nngt.Group(10)
room2 = nngt.Group(20)
room3 = nngt.Group(20)

names = ["R1", "R2", "R3"]

struct = nngt.Structure.from_groups((room1, room2, room3), names)

g = nngt.Graph(structure=struct)

for room in struct:
    nngt.generation.connect_groups(g, room, room, "erdos_renyi", avg_deg=5)

nngt.generation.connect_groups(g, (room1, room2), struct, "erdos_renyi",
                               avg_deg=3, ignore_invalid=True)

nngt.generation.connect_groups(g, room3, room1, "erdos_renyi", avg_deg=5)

nngt.plot.library_draw(g, tight=False, axis=axes[0], ecolor="grey",
                       show=False)

axes[0].set_title("Spring-block layout")


# random layout

sw = nngt.generation.watts_strogatz(4, 0.3, nodes=num_nodes)

betw = nngt.analysis.betweenness(sw, "node")

nngt.plot.draw_network(sw, nsize=betw, ncolor="out-degree", axis=axes[1],
                       ecolor="lightgrey", tight=False, show=False)

axes[1].set_title("Random layout")


# circular layout for small-world networks

nngt.plot.draw_network(sw, nsize=betw, ncolor="out-degree", layout="circular",
                       ecolor="lightgrey", axis=axes[2],
                       show=False, tight=False)
```

(continues on next page)

(continued from previous page)

```
axes[2].set_title("Circular layout")


# spatial layout

c1 = nngt.geometry.Shape.disk(100)
c2 = nngt.geometry.Shape.disk(100, centroid=(50, 0))

shape = nngt.geometry.Shape.from_polygon(c1.union(c2))

max_nsize = 15
npos  = shape.seed_neurons(num_nodes, soma_radius=0.5*max_nsize)

g = nngt.generation.distance_rule(10, shape=shape, nodes=num_nodes, avg_deg=5,
                                  positions=npos)

cc = nngt.analysis.local_clustering(g)

nngt.plot.draw_network(g, ncolor=cc, axis=axes[3], ecolor="grey", show=False,
                       eborder_width=0.5, eborder_color="w", esize=10,
                       max_nsize=max_nsize, tight=False)

axes[3].set_title("Spatial layout")

# save figure

fname = os.getcwd() + "/layouts.png"

plt.savefig(fname)
plt.switch_backend(mpl_backend)

img = plt.imread(fname)

_, ax = plt.subplots(figsize=(10, 8))
ax.imshow(img)

ax.axis('off')

plt.tight_layout()
plt.show()

try:
    os.remove(fname)
except:
    pass
```



Note that the last lines are just a little trick to make the figure be automatically detected by Sphinx-gallery. For normal use cases you can just do a regular `plt.show()`.

Total running time of the script: (0 minutes 3.711 seconds)

2.4.2 Visualizing graph properties

Plot the degree distributions of a graph

```
import nngt
import nngt.plot as nplt

import matplotlib.pyplot as plt

plt.rcParams.update({
    'axes.edgecolor': 'grey', 'xtick.color': 'grey', 'ytick.color': 'grey',
    "figure.facecolor": (0, 0, 0, 0), "axes.facecolor": (0, 0, 0, 0),
    "axes.labelcolor": "grey", "text.color": "grey", "legend.facecolor": "none"
})
```

(continues on next page)

(continued from previous page)

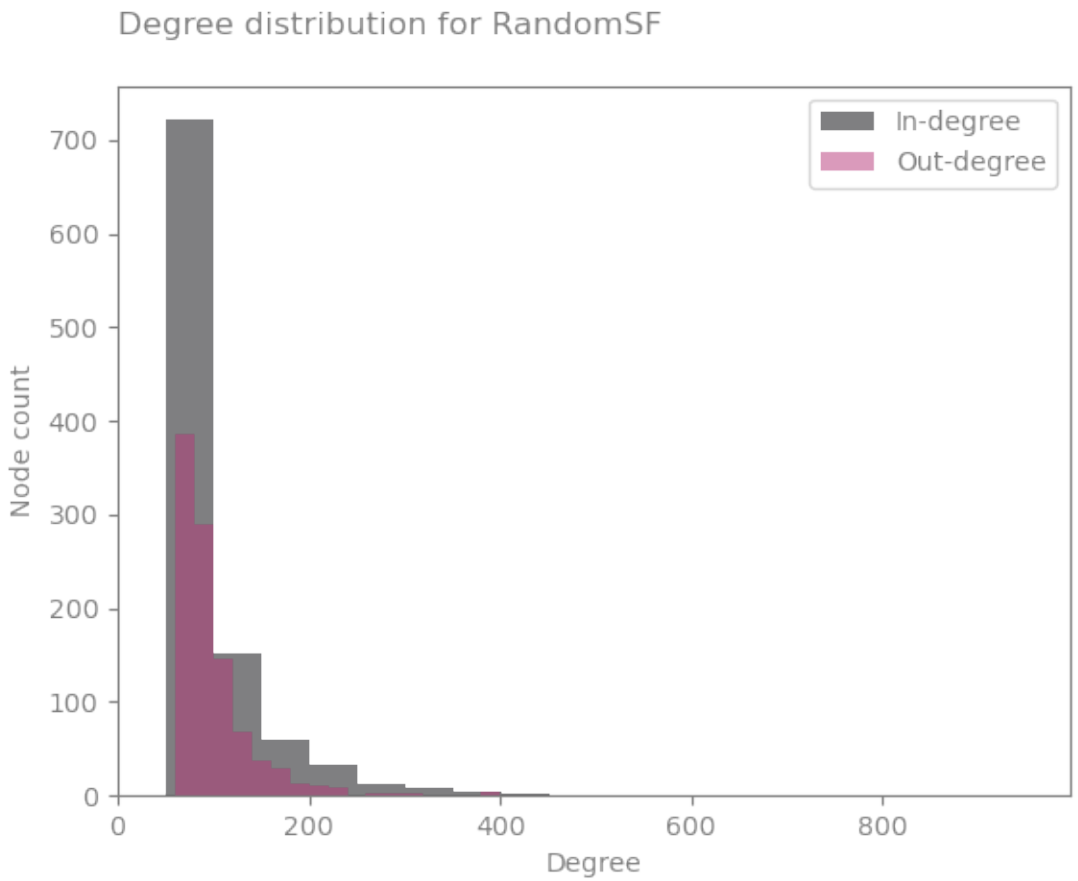
```
nngt.seed(0)
```

First, let's create a scale-free network

```
g = nngt.generation.random_scale_free(2.1, 3.2, nodes=1000, avg_deg=100)
```

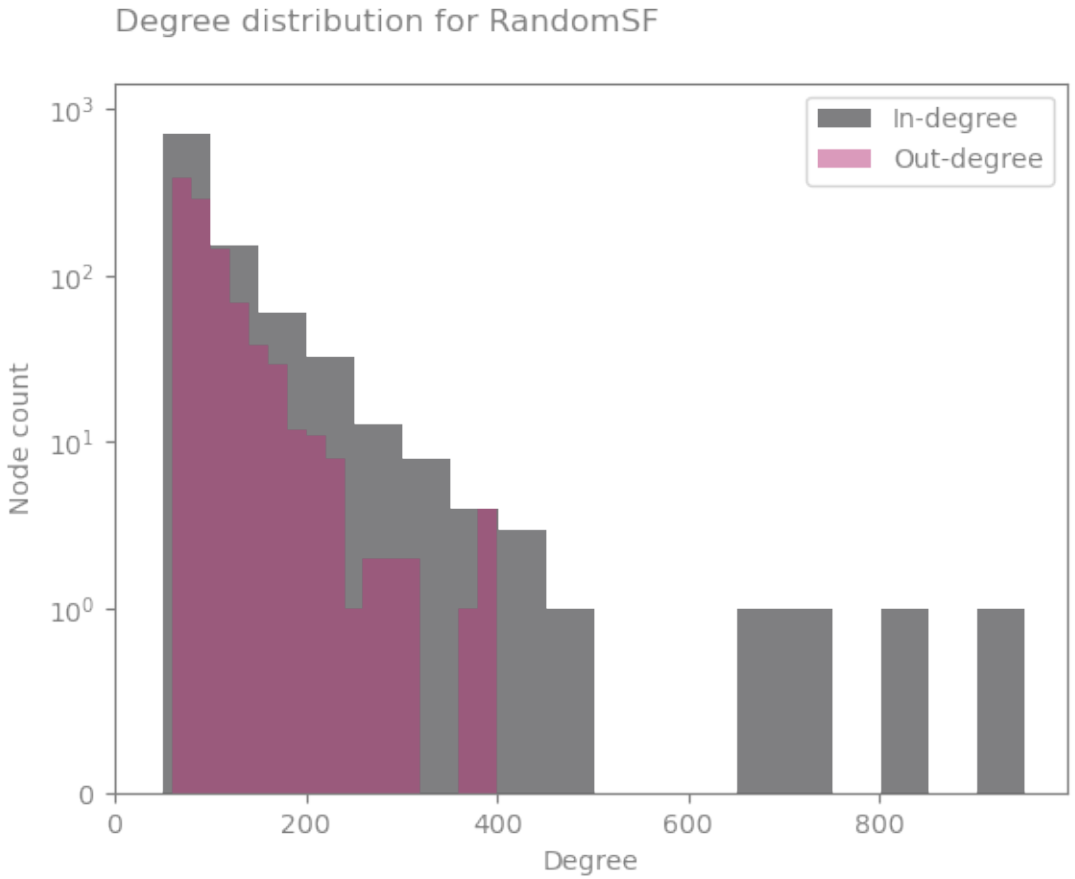
Plot the degree distribution

```
nplt.degree_distribution(g, deg_type=["in", "out"], show=True)
```



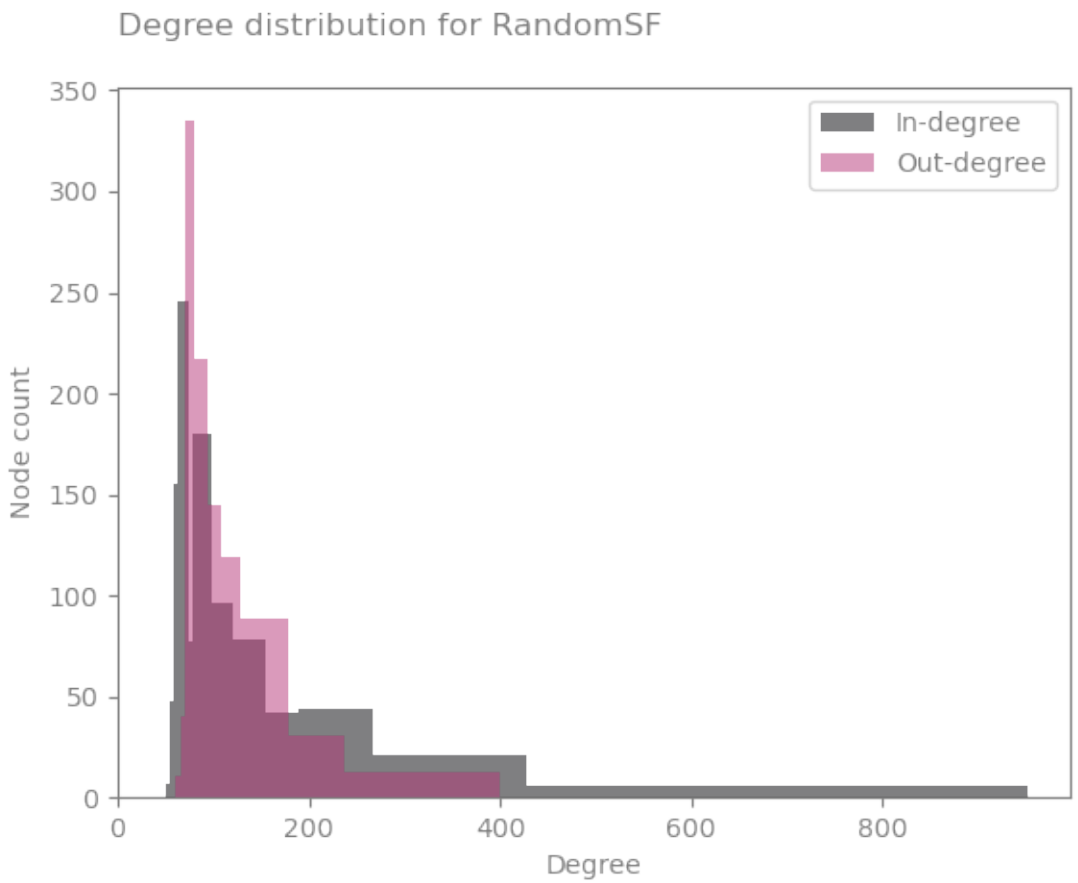
It's not bad… but we don't see much! Let's move a more relevant scale

```
nplt.degree_distribution(g, deg_type=["in", "out"], logy=True, show=True)
```



Or we can use Bayesian binning

```
nplt.degree_distribution(g, deg_type=["in", "out"], num_bins="bayes",
                         show=True)
```



Total running time of the script: (0 minutes 1.559 seconds)

Plot the betweenness distributions of a graph

```
import nngt
import nngt.plot as nplt
from nngt.geometry import Shape

import matplotlib.pyplot as plt

plt.rcParams.update({
    'axes.edgecolor': 'grey', 'xtick.color': 'grey', 'ytick.color': 'grey',
    "figure.facecolor": (0, 0, 0, 0), "axes.facecolor": (0, 0, 0, 0),
    "axes.labelcolor": "grey", "text.color": "grey", "legend.facecolor": "none"
})


nngt.seed(0)
```

Let's start by making a random exponential graph

```
shape = Shape.disk(100)
```

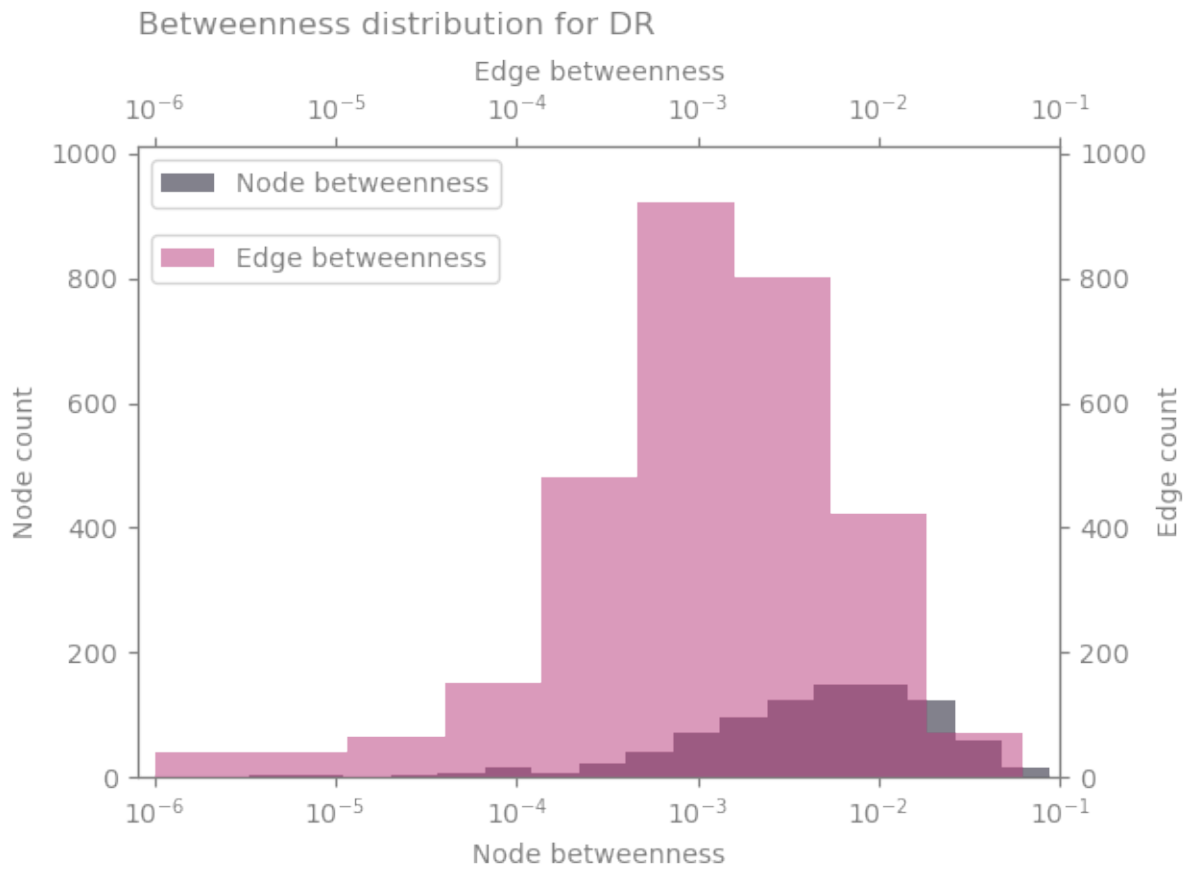
(continues on next page)

(continued from previous page)

```
g = nngt.generation.distance_rule(5, shape=shape, nodes=1000, avg_deg=3)
```

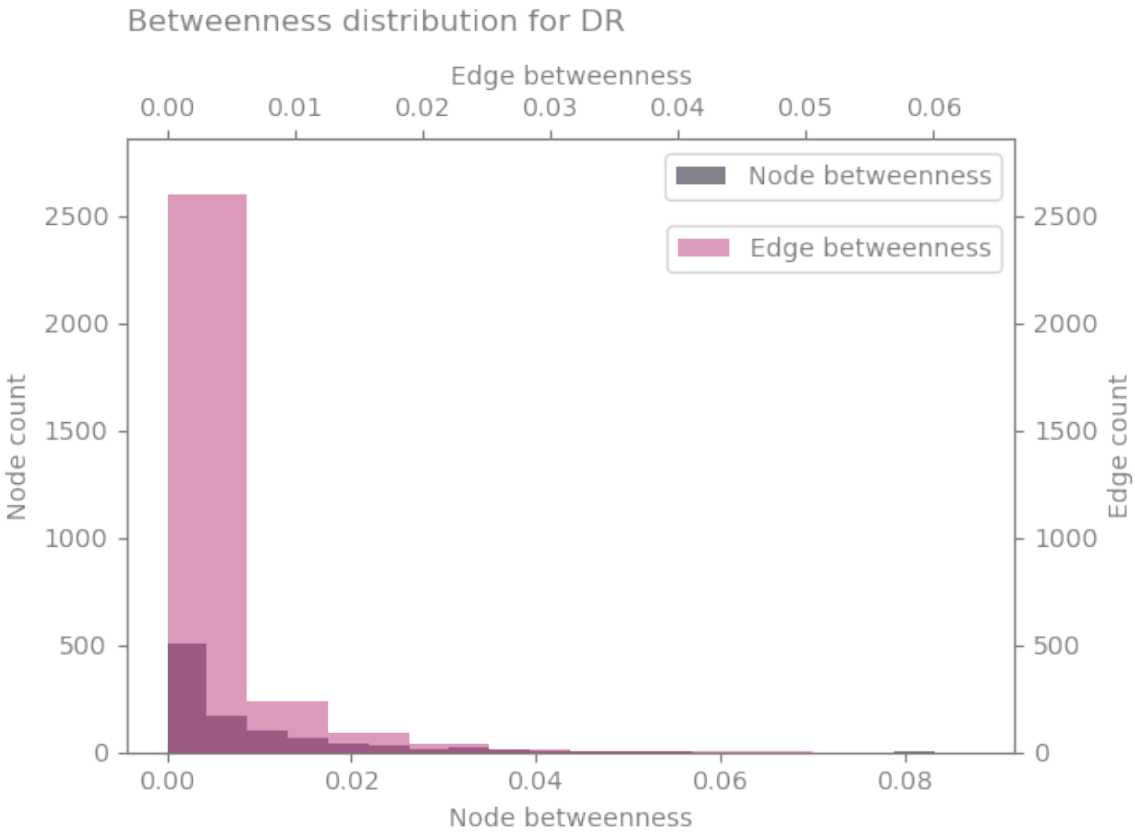
then we can plot the betweenness

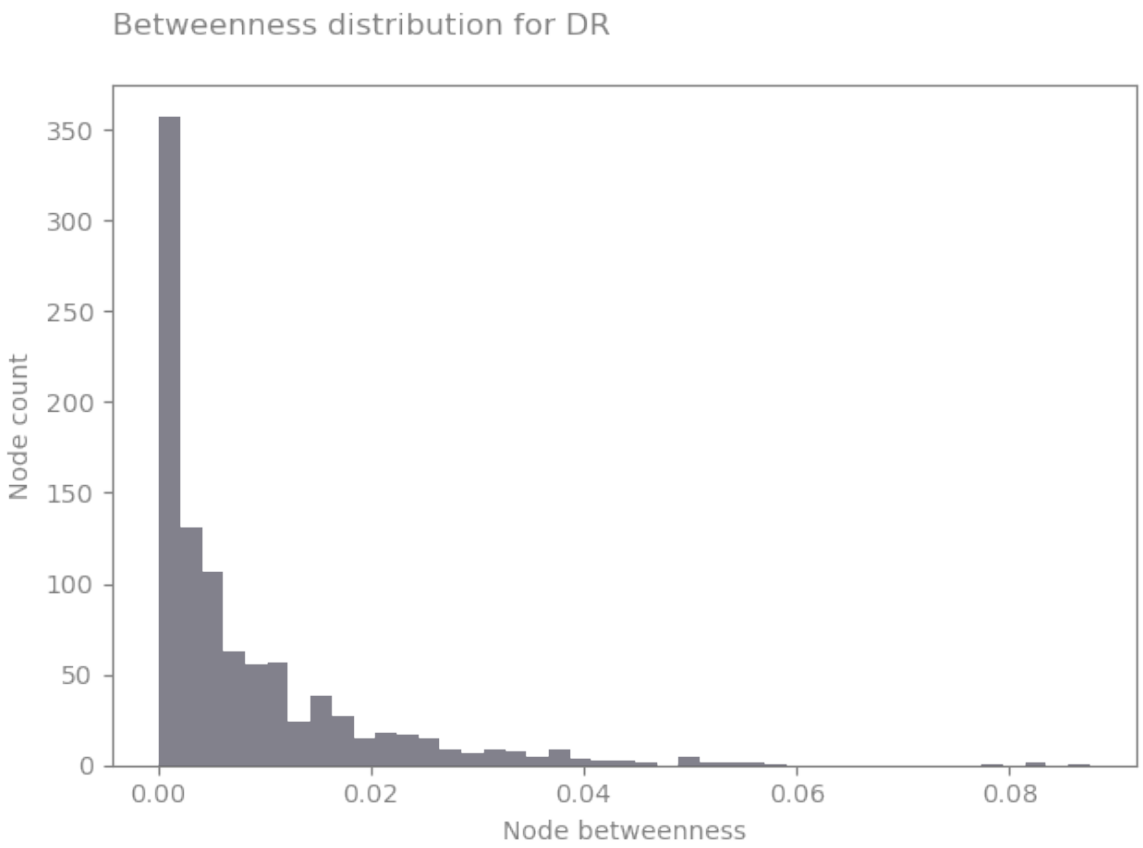
```
nplt.betweenness_distribution(g, logx=True, show=True, legend_location='left')
```



we can of course change various parameters and plot only the nodes

```
nplt.betweenness_distribution(g, logx=False, show=True)

nplt.betweenness_distribution(g, btype="node", num_nbins="auto", alpha=0.5,
                              show=True)
```

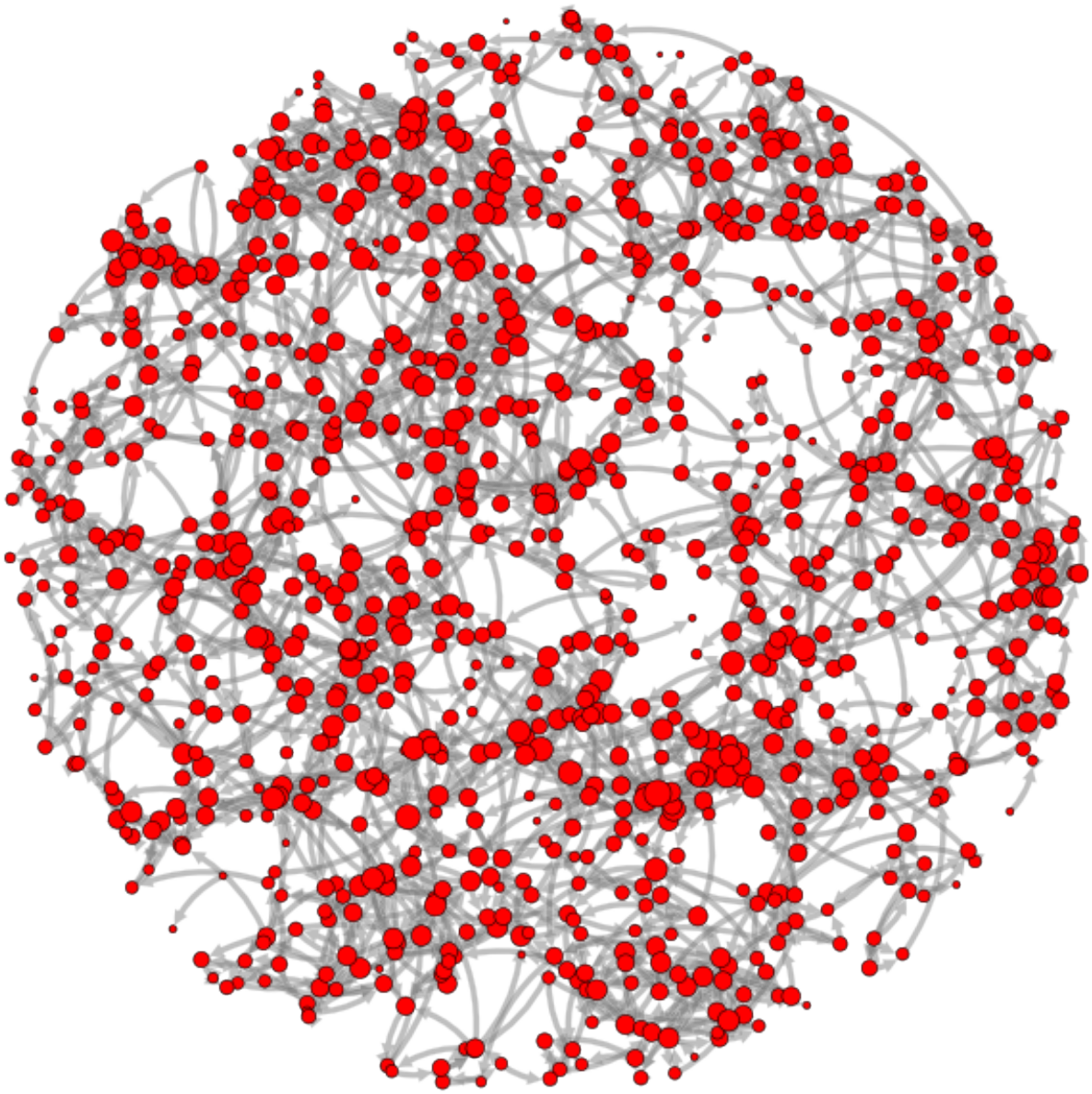

Betweenness distribution for DR
Edge betweenness
0.00
0.01
0.02
0.03
0.04
0.05
0.06
Node betweenness
Edge betweenness
2500
2000
1500
1000
500
0
Node count
Edge count
0.00
0.02
0.04
0.06
0.08
Node betweenness



-

By the way, this is the graph we're looking at

```
nplt.draw_network(g, max_nsize=5, max_esize=4, ecolor="grey", eborder_color="w",
                  curved_edges=True, show_environment=False, show=True)
```



Total running time of the script: (0 minutes 31.865 seconds)

Plot various graph properties

```
import nngt
import nngt.plot as nplt
from nngt.geometry import Shape

import matplotlib.pyplot as plt

plt.rcParams.update({
    'axes.edgecolor': 'grey', 'xtick.color': 'grey', 'ytick.color': 'grey',
```

(continues on next page)

(continued from previous page)

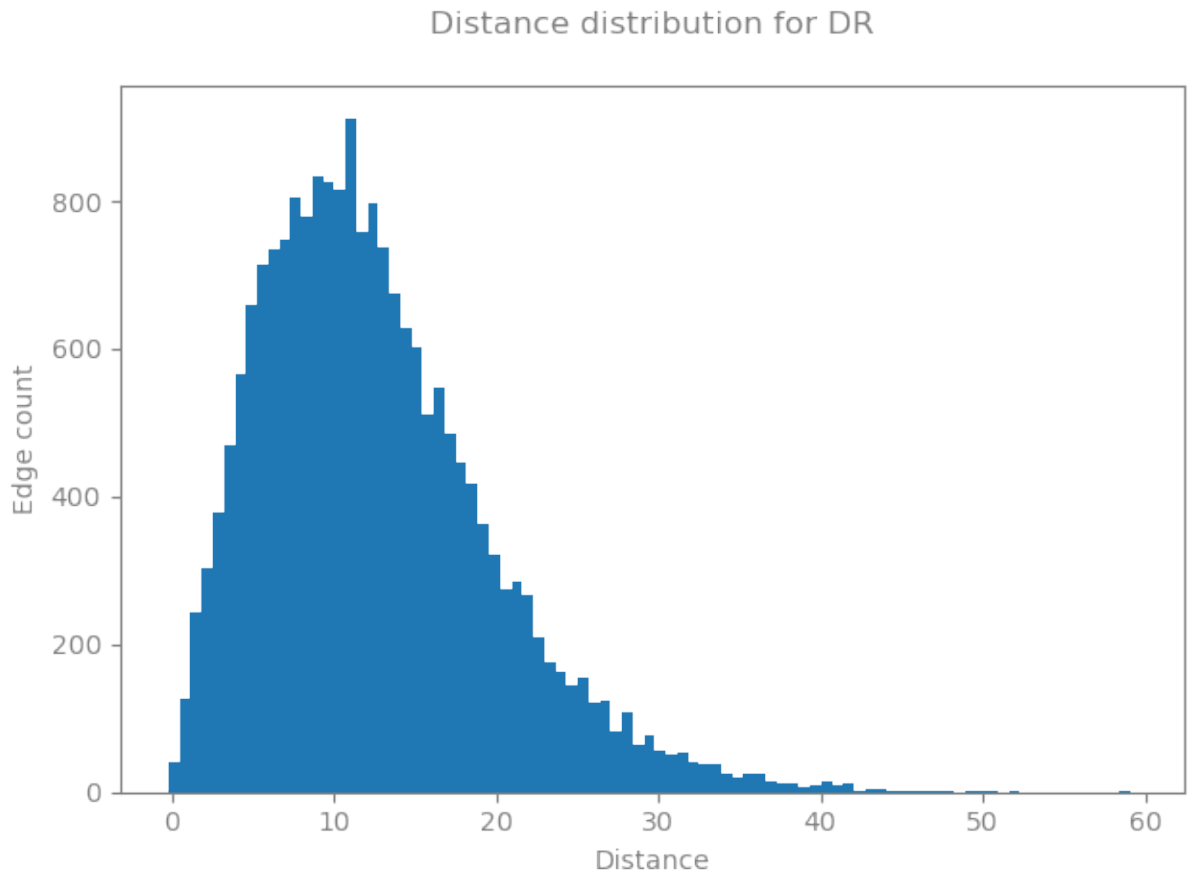
```
    "figure.facecolor": (0, 0, 0, 0), "axes.facecolor": (0, 0, 0, 0),
    "axes.labelcolor": "grey", "text.color": "grey"
})


nngt.seed(0)
```

Let's start by making a random exponential graph

```
shape = Shape.disk(100)

g = nngt.generation.distance_rule(5, shape=shape, nodes=1000, avg_deg=20)
```

Let's plot the distances

```
nplt.edge_attributes_distribution(g, "distance", show=True)
```



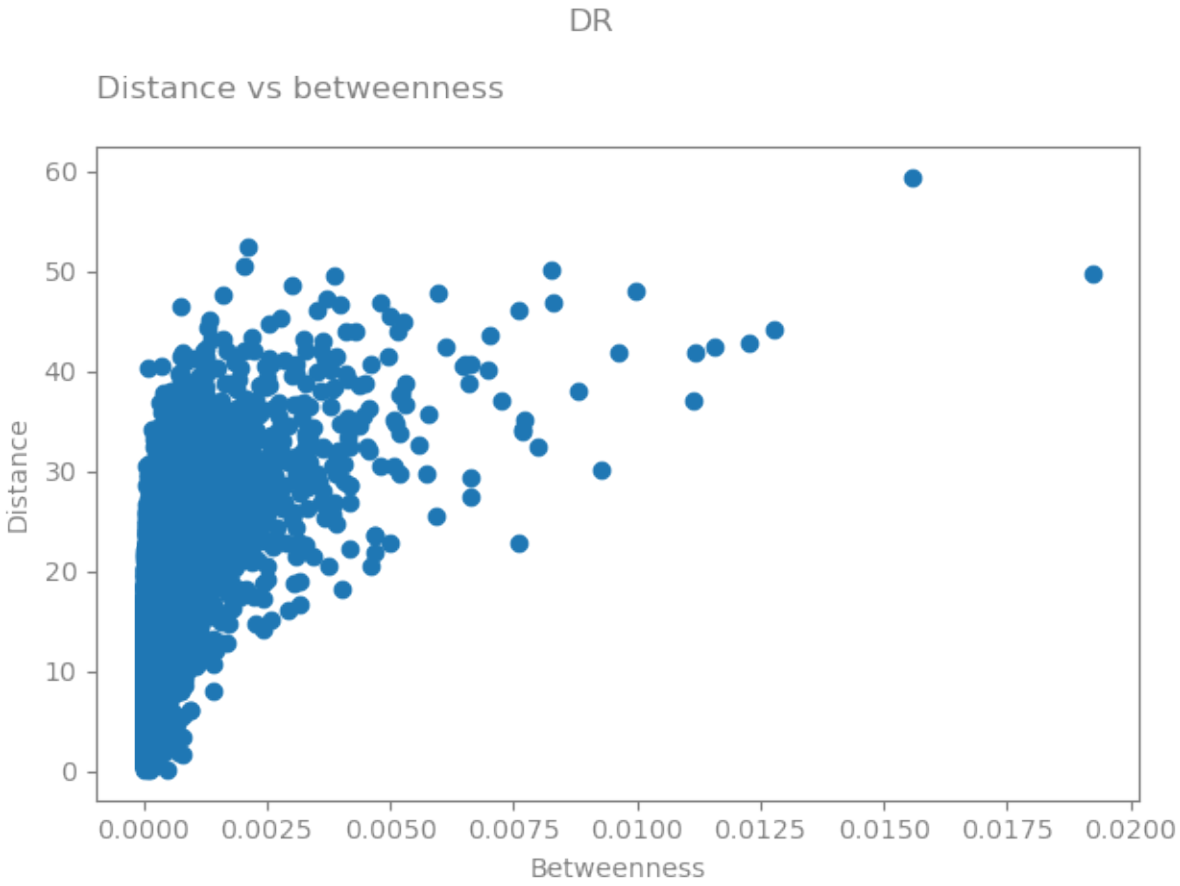
We then compute the betweenness and see how it correlates with the distance

```
nbetw, ebetw = nngt.analysis.betweenness(g)

g.new_edge_attribute("betweenness", "float", values=ebetw)
```

(continues on next page)

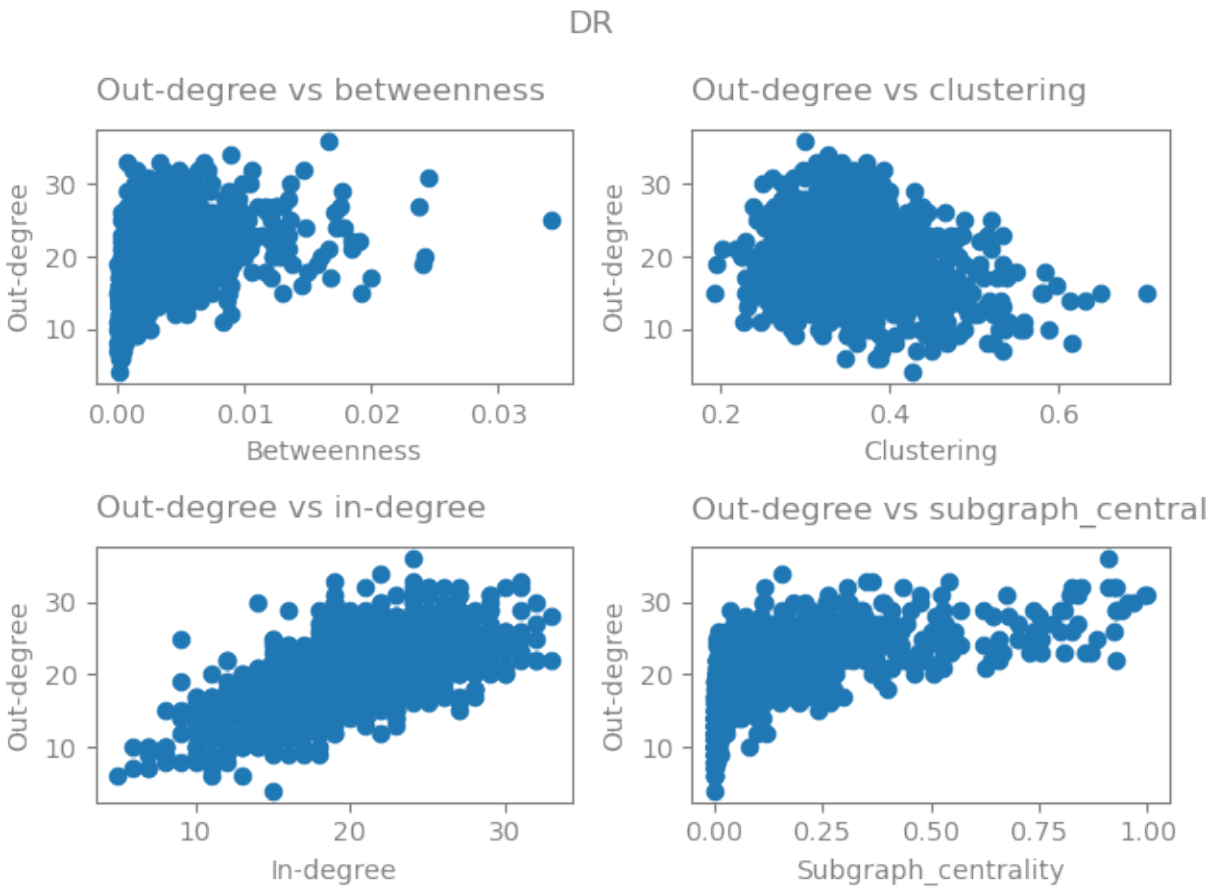
(continued from previous page)

```
nplt.correlation_to_attribute(g, "distance", "betweenness",
                              attribute_type="edge", show=True)
```



Let's check the correlations between various node properties and their degree

```
g.new_node_attribute("betweenness", "float", values=nbetw)

attr = ["betweenness", "clustering", "in-degree", "subgraph_centrality"]

nplt.correlation_to_attribute(g, "out-degree", attr, show=True)
```



Total running time of the script: (0 minutes 50.907 seconds)

2.4.3 Visualizing graph structures

The following examples show how to use NNGT to draw graphs in ways that make their structural properties stand out.

2.4.4 Visualizing graph properties

2.5 Contributing to NNGT

2.5.1 Signaling issues and bugs

If you encounter something that you think is an error, please let me know either via the user mailing list or directly on the issue tracker.

Warning: When signaling a bug, please **always** include a python script containing a minimal working example (MWE) that reproduces the issue.

2.5.2 Preparing a contribution

To prepare a contribution to NNGT, you should follow these successive steps:

1. start from the `main` branch: `git checkout main`,
2. create a new branch from `main`: `git checkout -b name-of-your-choice`,
3. make the changes you want to and commit them,
4. check them locally using: `pytest testing` (you'll need to install pytest via `pip install pytest`)

2.5.3 Sending a patch to SourceHut

To contribute on SourceHut, you don't need an account (though you can also make a patch using the website if you have an account there).

What you need is to use `git send-email`, and you can find how to install and set it up on this page.

Before sending you patch, please squash you commits using:

```
git checkout -b patch-branch
git merge --squash name-of-your-choice
git checkout -a -m "A descriptive message of the changes"
```

First contribution

Once this is done, you can push your patch to the mailing list using:

```
git send-email --annotate --to=~tfardet/nngt-developers@lists.sr.ht -v1 HEAD^
```

you can add further information in the description using annotate.

Warning: Always use `--annotate` because you will need to change the subject from "[PATCH v1]" to "[PATCH NNGT]" or "[PATCH NNGT v1]" (as you prefer as long as the second word is NNGT) so that the patch is automatically tested on SourceHut

Do not hesitate to ask for help on the developer mailing list if you need help on your first contribution.

Post-review changes: later contributions

If changes are requested, apply the changes to the branch `name-of-your-choice`, then reset `patch-branch`

```
git checkout patch-branch
git fetch origin
git reset --hard origin/main
git merge --squash name-of-your-choice
git checkout -a -m "A descriptive message of the changes"
```

then, publish the patch saying it's a new version:

```
git send-email --annotate --to=~tfardet/nngt-developers@lists.sr.ht -v2 HEAD^
```

Or -v3, -v4, etc for later patches.

Warning: As before, use annotate to change the subject to "[PATCH NNGT]" or "[PATCH NNGT v2]" so that the patch is automatically tested on SourceHut

2.5.4 Making a PR on GitHub

If you prefer using GitHub, then you can open a PR on the repo.

2.6 Database module

NNGT provides a database to store NEST simulations. This database requires `peewee>3` to work and can be switched on using:

```
nngt.set_config("use_database", True)
```

The commands are then used by calling `nngt.database` to access the database tools.

2.6.1 Functions

`nngt.database.get_results`(*table*, *column=None*, *value=None*)

Return the entries where the attribute *column* satisfies the required equality.

Parameters

- **table** (*str*) – Name of the table where the search should be performed (among `'simulation'`, `'computer'`, `'neuralnetwork'`, `'activity'`, `'synapse'`, `'neuron'`, or `'connection'`).
- **column** (*str, optional (default: None)*) – Name of the variable of interest (a column on the table). If None, the whole table is returned.

- **value** (*column* corresponding type, optional (default: None)) – Specific value for the variable of interest. If None, the whole column is returned.

Returns `peewee.SelectQuery` with entries matching the request.

`nngt.database.is_clear()`
Check that the logs are clear.

`nngt.database.log_simulation_end(network=None, log_activity=True)`
Record the simulation completion and simulated times, save the data, then reset.

`nngt.database.log_simulation_start(network, simulator, save_network=True)`
Record the simulation start time, all nodes, connections, network, and computer properties, as well as some of simulation's.

Parameters

- **network** (*Network* or subclass) – Network used for the current simulation.
- **simulator** (*str*) – Name of the simulator.
- **save_network** (*bool, optional (default: True)*) – Whether to save the network or not.

`nngt.database.reset()`
Reset log status.

2.6.2 Recording a simulation

```
nngt.database.log_simulation_start(net, "nest-2.14")
nest.Simulate(1000.)
nngt.database.log_simulation_end()
```

2.6.3 Checking results in the database

The database contains the following tables, associated to their respective fields:

- 'activity': `Activity`,
- 'computer': `Computer`,
- 'connection': `Connection`,
- 'neuralnetwork': `NeuralNetwork`,
- 'neuron': `Neuron`,
- 'simulation': `Simulation`,
- 'synapse': `Synapse`.

These tables are the first keyword passed to *get_results()*, you can find the existing columns for each of the tables in the following classes descriptions:

Store results into a database

2.7 Geospatial module

The geospatial module contains functions and objects that enable straightforward network plots together with geospatial data.

It relies on geopandas and cartopy in the background.

See "*Geospatial networks*" for an example.

2.7.1 Content

The module provides four main tools:

- *draw_map()* to plot a network on a map
- `maps`, a dictionary of `GeoDataFrame` containing data from the NaturalEarth project:
  - `"adaptive"` entry contains 280 entries (in 2021) at the coarsest scale available for each entry,
  - `"110m"` entry contains 177 countries at 110m resolution,
  - `"50m"` entry contains 241 countries at 50m resolution,
  - `"10m"` entry contains 295 map subunits at 10m resolution.
- `code_to_names` is a dictionary converting A3 ISO codes to the associated unit's name (available for all four scales: adaptive and 110/50/10m).
- `cities` is a `GeoDataFrame` containing cities' data from from NaturalEarth "populated places".

Note: This data is automatically downloaded when the module is loaded for the first time. It is stored in `cartopy.config['data_dir']`.

2.7.2 Details

nngt.geospatial.**draw_map**(*graph*, *node_names*, *geodata=None*, *geodata_names=None*, *points=None*, *show_points=False*, *linecolor=None*, *hue=None*, *proj=None*, *all_geodata=True*, *axis=None*, *show=False*, ***kwargs*)

Draw a network on a map.

Parameters

- **graph** (*Graph* or subclass) – Graph to plot.
- **node_names** (*str*) – Node attribute containing the nodes' names or A3 codes. This attribute will be used to place each node on the map. By default (if no *geodata* is provided), the world map is used and each node must therefore be associated to a country name or (better) an A3 ISO code.
- **geodata** (`GeoDataFrame`, optional (default: world map)) – Optional dataframe containing the geospatial information. Predefined geodatas are "110m", "50m", and "10m" for world maps with respectively 110, 50, and 10 meter resolutions, or "adaptive" (default) for a world map with adaptive resolution depending on the country size.
- **geodata_names** (*str, optional (default: "NAME_LONG" or "SU_A3")*) – Column in *geodata* corresponding to the *node_names* (respectively for full country names or A3 codes).

- **points** (*str, optional (default: capitals and representative points)*) – Whether a precise point should be associated to each node. It can be either an entry in *geodata*, the "centroid" of each geometry entry, or a "representative" point. By default, if the world map is used, each country will be associated to its capital (contained in the module's `cities` object); if another *geodata* element is provided, it defaults to "representative".
- **show_points** (*bool, optional (default: False)*) – Wether the points should be displayed.
- **linecolor** (*str, char, float or array, optional (default: current palette)*) – Color of the map lines.
- **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** (*str, char, float or array, optional (default: "k")*) – Edge color. If ecolor="groups", edges color will depend on the source and target groups, i.e. only edges from and toward same groups will have the same 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*.
- **threshold** (*float, optional (default: 0.5)*) – Size under which edges are not plotted.
- **proj** (`cartopy.crs` object, optional (default: cartesian plane)) – Projection that will be used to draw the map.
- **all_geodata** (*bool, optional (default: True)*) – Whether all the data contained in *geodata* should be plotted, even if *graph* contains only a subset of it.
- **axis** (*matplotlib axis, optional (default: a new axis)*) – Axis that will be used to plot the graph.
- ****kwargs** (*dict*) – All possible arguments from *draw_network()*.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Barrat2004] Barrat, Barthelemy, Pastor-Satorras, Vespignani. The Architecture of Complex Weighted Networks. PNAS 2004, 101 (11). DOI: 10.1073/pnas.0400087101.

[Clemente2018] Clemente, Grassi. Directed Clustering in Weighted Networks: A New Perspective. Chaos, Solitons & Fractals 2018, 107, 26–38. DOI: 10.1016/j.chaos.2017.12.007, arXiv: 1706.07322.

[Fagiolo2007] Fagiolo. Clustering in Complex Directed Networks. Phys. Rev. E 2007, 76, (2), 026107. DOI: 10.1103/PhysRevE.76.026107, arXiv: physics/0612169.

[Onnela2005] Onnela, Saramäki, Kertész, Kaski. Intensity and Coherence of Motifs in Weighted Complex Networks. Phys. Rev. E 2005, 71 (6), 065103. DOI: 10.1103/physreve.71.065103, arXiv: cond-mat/0408629.

[Saramaki2007] Saramäki, Kivelä, Onnela, Kaski, Kertész. Generalizations of the Clustering Coefficient to Weighted Complex Networks. Phys. Rev. E 2007, 75 (2), 027105. DOI: 10.1103/PhysRevE.75.027105, arXiv: cond-mat/0608670.

[Zhang2005] Zhang, Horvath. A General Framework for Weighted Gene Co-Expression Network Analysis. Statistical Applications in Genetics and Molecular Biology 2005, 4 (1). DOI: 10.2202/1544-6115.1128, PDF.

[Fardet2021] Fardet, Levina. Weighted directed clustering: interpretations and requirements for heterogeneous, inferred, and measured networks. 2021. arXiv: 2105.06318.

[ig-connected] igraph - is_connected

[gt-adjacency] graph-tool - spectral.adjacency

[nx-adjacency] networkx - convert_matrix.to_scipy_sparse_matrix

[nx-sp] networkx - algorithms.shortest_paths.generic.all_shortest_paths

[nx-assortativity] networkx - algorithms.assortativity.degree_assortativity_coefficient

[nx-sp] networkx - algorithms.shortest_paths.generic.average_shortest_path_length

[nx-ebetw] networkx - algorithms.centrality.edge_betweenness_centrality

[nx-nbetw] networkx - networkx.algorithms.centrality.betweenness_centrality

[nx-harmonic] networkx - algorithms.centrality.harmonic_centrality

[nx-closeness] networkx - algorithms.centrality.closeness_centrality

[nx-ucc] networkx - algorithms.components.connected_components

[nx-scc] networkx - algorithms.components.strongly_connected_components

[nx-wcc] networkx - algorithms.components.weakly_connected_components

[nx-diameter] networkx - algorithms.distance_measures.diameter

[nx-dijkstra] networkx - algorithms.shortest_paths.weighted.all_pairs_dijkstra

[gt-global-clustering] graph-tool - clustering.global_clustering

[ig-global-clustering] igraph - transitivity_undirected

[nx-global-clustering] networkx - algorithms.cluster.transitivity

[Barrat2004] Barrat, Barthelemy, Pastor-Satorras, Vespignani. The Architecture of Complex Weighted Networks. PNAS 2004, 101 (11). DOI: 10.1073/pnas.0400087101.

[Onnela2005] Onnela, Saramäki, Kertész, Kaski. Intensity and Coherence of Motifs in Weighted Complex Networks. Phys. Rev. E 2005, 71 (6), 065103. DOI: 10.1103/physreve.71.065103, arxiv:*cond-mat/0408629*.

[Fagiolo2007] Fagiolo. Clustering in Complex Directed Networks. Phys. Rev. E 2007, 76 (2), 026107. DOI: 10.1103/PhysRevE.76.026107, arXiv: physics/0612169.

[Zhang2005] Zhang, Horvath. A General Framework for Weighted Gene Co-Expression Network Analysis. Statistical Applications in Genetics and Molecular Biology 2005, 4 (1). DOI: 10.2202/1544-6115.1128, PDF.

[Fardet2021] Fardet, Levina. Weighted directed clustering: interpretations and requirements for heterogeneous, inferred, and measured networks. 2021. arXiv: 2105.06318.

[nx-global-clustering] networkx - algorithms.cluster.transitivity

[Yin2019] Yin, Benson, and Leskovec. The Local Closure Coefficient: A New Perspective On Network Clustering. Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining 2019, 303-311. DOI: 10.1145/3289600.3290991, PDF.

[Fardet2021] Fardet, Levina. Weighted directed clustering: interpretations and requirements for heterogeneous, inferred, and measured networks. 2021. arXiv: 2105.06318.

[Barrat2004] Barrat, Barthelemy, Pastor-Satorras, Vespignani. The Architecture of Complex Weighted Networks. PNAS 2004, 101 (11). DOI: 10.1073/pnas.0400087101.

[Clemente2018] Clemente, Grassi. Directed Clustering in Weighted Networks: A New Perspective. Chaos, Solitons & Fractals 2018, 107, 26–38. DOI: 10.1016/j.chaos.2017.12.007, arXiv: 1706.07322.

[Fagiolo2007] Fagiolo. Clustering in Complex Directed Networks. Phys. Rev. E 2007, 76, (2), 026107. DOI: 10.1103/PhysRevE.76.026107, arXiv: physics/0612169.

[Onnela2005] Onnela, Saramäki, Kertész, Kaski. Intensity and Coherence of Motifs in Weighted Complex Networks. Phys. Rev. E 2005, 71 (6), 065103. DOI: 10.1103/physreve.71.065103, arXiv: cond-mat/0408629.

[Saramaki2007] Saramäki, Kivelä, Onnela, Kaski, Kertész. Generalizations of the Clustering Coefficient to Weighted Complex Networks. Phys. Rev. E 2007, 75 (2), 027105. DOI: 10.1103/PhysRevE.75.027105, arXiv: cond-mat/0608670.

[Zhang2005] Zhang, Horvath. A General Framework for Weighted Gene Co-Expression Network Analysis. Statistical Applications in Genetics and Molecular Biology 2005, 4 (1). DOI: 10.2202/1544-6115.1128, PDF.

[Fardet2021] Fardet, Levina. Weighted directed clustering: interpretations and requirements for heterogeneous, inferred, and measured networks. 2021. arXiv: 2105.06318.

[nx-local-clustering] networkx - algorithms.cluster.clustering

[nx-reciprocity] networkx - algorithms.reciprocity.overall_reciprocity

[nx-sp] networkx - algorithms.shortest_paths.weighted.multi_source_dijkstra

[nx-sp] networkx - algorithms.shortest_paths.generic.shortest_path

[Muldoon2016] Muldoon, Bridgeford, Bassett. Small-World Propensity and Weighted Brain Networks. Sci Rep 2016, 6 (1), 22057. DOI: 10.1038/srep22057, arXiv: 1505.02194.

[Barrat2004] Barrat, Barthelemy, Pastor-Satorras, Vespignani. The Architecture of Complex Weighted Networks. PNAS 2004, 101 (11). DOI: 10.1073/pnas.0400087101.

[Onnela2005] Onnela, Saramäki, Kertész, Kaski. Intensity and Coherence of Motifs in Weighted Complex Networks. Phys. Rev. E 2005, 71 (6), 065103. DOI: 10.1103/physreve.71.065103, arxiv:*cond-mat/0408629*.

[Estrada2005] Ernesto Estrada and Juan A. Rodríguez-Velázquez, Subgraph centrality in complex networks, PHYSICAL REVIEW E 71, 056103 (2005), DOI: 10.1103/PhysRevE.71.056103, arXiv: cond-mat/0504730.

[Barrat2004] Barrat, Barthelemy, Pastor-Satorras, Vespignani. The Architecture of Complex Weighted Networks. PNAS 2004, 101 (11). DOI: 10.1073/pnas.0400087101.

[Fagiolo2007] Fagiolo. Clustering in Complex Directed Networks. Phys. Rev. E 2007, 76, (2), 026107. DOI: 10.1103/PhysRevE.76.026107, arXiv: physics/0612169.

[Onnela2005] Onnela, Saramäki, Kertész, Kaski. Intensity and Coherence of Motifs in Weighted Complex Networks. Phys. Rev. E 2005, 71 (6), 065103. DOI: 10.1103/physreve.71.065103, arXiv: cond-mat/0408629.

[Zhang2005] Zhang, Horvath. A General Framework for Weighted Gene Co-Expression Network Analysis. Statistical Applications in Genetics and Molecular Biology 2005, 4 (1). DOI: 10.2202/1544-6115.1128, PDF.

[Barrat2004] Barrat, Barthelemy, Pastor-Satorras, Vespignani. The Architecture of Complex Weighted Networks. PNAS 2004, 101 (11). DOI: 10.1073/pnas.0400087101.

[Fagiolo2007] Fagiolo. Clustering in Complex Directed Networks. Phys. Rev. E 2007, 76, (2), 026107. DOI: 10.1103/PhysRevE.76.026107, arXiv: physics/0612169.

[Zhang2005] Zhang, Horvath. A General Framework for Weighted Gene Co-Expression Network Analysis. Statistical Applications in Genetics and Molecular Biology 2005, 4 (1). DOI: 10.2202/1544-6115.1128, PDF.

[newman-clustered-2003] Newman, M. E. J. Properties of Highly Clustered Networks, Phys. Rev. E 2003 68 (2). DOI: 10.1103/PhysRevE.68.026121, arXiv: cond-mat/0303183.

PYTHON MODULE INDEX

n

INDEX

F

G

H

I

L

M

N

O

P

R

S

T

U

W