NNGT Documentation

Release 1.3.0

Tanguy Fardet

Mar 09, 2020

User Documentation

CHAPTER 1

Overview

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

The library has two main targets:

- people looking for a unifying interface for the three main graph library, 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 being aimed at the graph-theoretical community, the third and fourth are more for the neuroscience community:

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

SpatialGraph a Graph embedded in space (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 connectivity: connectivity between the nodes can be chosen from various well-known graph models

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

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

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 2

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 (very basic) backend, installing one of the following libraries is highly recommended to do some proper network analysis:

- graph_tool (> 2.22 recommended)
- or igraph
- or networkx (>= 2.0)

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

```
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 2.7* since it is what people are used to but I recommend using version *3.5* or higher (replace all 27/2.7 by 35/3.5).

Homebrew

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

brew install gcc-7 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-gcc7 && sudo port install gsl +gcc7
sudo port install autoconf automake libtool
sudo port install python27 pip
sudo port select python python27
sudo port install py27-cython
sudo port select cython cython27
sudo port install py27-numpy py27-scipy py27-matplotlib py27-ipython
sudo port select ipython ipython-2.7
sudo port install py-graph-tool gtk3
```

Once the installation is done, you can just install:

```
export CC=gcc-7
export CXX=gcc-7
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 anyway...

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


#---------------------
## Try to load NEST? ---------------------------------------------------------
#---------------------

load_nest = True


#----------------------
## 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 = 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
```

(continues on next page)

(continued from previous page)

```
db_name     = main

# 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
```

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.

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 (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 of neurons
g1 = nngt.NeuralGroup(500)  # neurons 0 to 499
g2 = nngt.NeuralGroup(500)  # neurons 500 to 999
```

(continues on next page)

(continued from previous page)

```
# make population (without NEST models)
pop = nngt.NeuralPop.from_groups(
    (g1, g2), ("left", "right"), with_models=False)

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


'''
Connect the groups
'''

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

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

ng.connect_neural_groups(net, "left", "left", "newman_watts", **prop_nw)
ng.connect_neural_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
}
ng.connect_neural_types(net, 1, 1, "newman_watts", **prop_nw)

# inhib -> exc (Random scale-free)
prop_rsf = {
    "in_exp": 2.1,
```

(continues on next page)

(continued from previous page)

```
    "out_exp": 2.6,
    "density": 0.2
}
ng.connect_neural_types(net, -1, 1, "random_scale_free", **prop_rsf)

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

Send the network to 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
    iconn  = nest.GetConnections(
        source=list(net.population["inhibitory"].nest_gids),
        target=list(net.population.nest_gids))
    w_nest = set(nest.GetStatus(iconn, "weight"))
    # in NNGT, inhibitory weights are positive to work with graph analysis
    # methods; they are automatically converted to negative weights in NEST
    print("NNGT weights:", w_nngt, "versus NEST weights", w_nest)
```

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": 0.1, "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)
    ns.set_noise(excs, 10., 2.)

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

(continues on next page)

(continued from previous page)

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

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

Go to other tutorials:

- *Intro & user manual*
- *Properties of graph components*
- *Multithreading*
- *Neural groups and 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:

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 `nodes_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

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)
```

(continues on next page)

(continued from previous page)

```
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.nodes_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 *Neural groups and 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 *Neural groups and 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:

```python
# 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)
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
```

(continues on next page)

(continued from previous page)

```
# 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
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

- 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.

Go to other tutorials:

Multithreading

Principle

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

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

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

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]})
```

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.

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Neural groups and populations*
- *Interacting with the NEST simulator*
- *Activity analysis*
- *Properties of graph components*

Neural groups and populations

One of the key features of NNGT is to enable users to group nodes (neurons) into groups sharing common properties in order to facilitate the generation of 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

Neural groups can be created easily through calls to `nngt.NeuralGroup`.

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

creates a single empty group (nothing very interesting).

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

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

```
group1 = NeuralGroup(500)  # a group with 500 neurons
```

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

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

More realistic 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:

```
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:

```
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()`:

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

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': {"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_neural_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)

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

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*
- *Multithreading*
- *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 grap 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 with regard to NEST 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 *Neural groups and 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
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': -57., '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.})
# bursters
params2.update({'b': 30., 'V_reset': -50., 'tau_w': 500.})

oscill = nngt.NeuralGroup(
    nodes=400, neuron_model='aeif_psc_alpha', neuron_param=params1)
burst = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', neuron_param=params2)
adapt = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', 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=250., 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*
- *Multithreading*
- *Neural groups and populations*
- *Activity analysis*
- *Properties of graph components*

Activity analysis

Principle

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

Sorted rasters

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

```
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

Main functions to send `Network` instances to NEST, as well as helper functions to excite or record the network activity.

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.

`data`
Returns the (N, 2) array of (senders, spike times).

`phases`

- "bursting" for periods of high activity where a large fraction of the network is recruited.
- "quiescent" for periods of low activity
- "mixed" for firing rate in between "quiescent" and "bursting".
- "localized" for periods of high activity but where only a small fraction of the network is recruited.

Note: See *parameters* for details on the conditions used to differenciate these phases.

Type Return the phases detected

`properties`
Returns the properties of the activity. Contains the following entries:

- "firing_rate": average value in Hz for 1 neuron in the network.
- "bursting": True if there were bursts of activity detected.
- "burst_duration", "IBI", "ISI", and "period" in ms, if "bursting" is True.
- "SpB" (Spikes per Burst): average number of spikes per neuron during a burst.

`simplify``()`

`nngt.simulation.`**`activity_types`**`(`*spike_detector*, *limits*, *network=None*, *phase_coeff=(0.5, 10.0)*, *mbis=0.5*, *mfb=0.2*, *mflb=0.05*, *skip_bursts=0*, *simplify=False*, *fignums=[]*, *show=False*`)`
Analyze the spiking pattern of a neural network.

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

Parameters

- **spike_detector** (*NEST node(s) (tuple or list of tuples)*) – The recording device that monitored the network's spikes.
- **limits** (*tuple of floats*) – Time limits of the simulation 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_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_detector" 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*, *use_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.
- **use_weights** (*bool, optional (default: True)*) – Whether to use the network weights or default ones (value: 10.).

Returns **gids** (*tuple (nodes in NEST)*) – GIDs of the neurons in the network.

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

Monitoring the activity of nodes in the network.

Parameters

- **group_name** (*list of strings*) – Names of the groups that should be recorded.
- **network** (`Network` or subclass) – Network which population will be used to differentiate groups.
- **nest_recorder** (*strings or list, optional (default: "spike_detector"0)*) – Device(s) to monitor the network.
- **params** (dict or list of, optional (default: *{}*)) – Dictionarie(s) containing the parameters for each recorder (see NEST documentation for details).

Returns

- **recorders** (*tuple*) – Tuple of the recorders' gids
- **recordables** (*tuple*) – Tuple of the recordables' names.

`nngt.simulation.`**`monitor_nodes`**(*gids*, *nest_recorder=None*, *params=None*, *network=None*)

Monitoring the activity of nodes in the network.

Parameters

- **gids** (*tuple of ints or list of tuples*) – GIDs of the neurons in the NEST subnetwork; either one list per recorder if they should monitor different neurons or a unique list which will be monitored by all devices.
- **nest_recorder** (*strings or list, optional (default: "spike_detector")*) – Device(s) to monitor the network.
- **params** (dict or list of, optional (default: *{}*)) – Dictionarie(s) containing the parameters for each recorder (see NEST documentation for details).
- **network** (`Network` or subclass, optional (default: None)) – Network which population will be used to differentiate groups.

Returns

- **recorders** (*tuple*) – Tuple of the recorders' gids
- **recordables** (*tuple*) – Tuple of the recordables' names.

`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.`**`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_detector"s 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.

Changed in version 1.1: Removed optional *weight_fraction* in favour of a compulsory *weight* to avoid problems when synaptic weights need to change. Because of this, *weight_normalization* is no longer necessary and has been removed.

Changed in version 1.0: Added *syn_type*, separating the excitatory and inhibitory degrees and weights.

Changed in version 0.8: Added *nodes*, removed *syn_model* and *syn_params*. Added *weight_normalization* to avoid issues with plastic synapses.

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*)

Submit neurons to a Poissonian rate of spikes.

Changed in version 0.9: Added *syn_spec* parameter.

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.

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

`nngt.simulation.`**`set_step_currents`**(*gids*, *times*, *currents*)

Set step-current excitations

Parameters

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

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

`nngt.simulation.`**`plot_activity`**(*gid_recorder=None*, *record=None*, *network=None*, *gids=None*, *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*)

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_detector"s are used.
- **record** (*tuple or list, optional (default: None)*) – List of the monitored variables for each device. If *gid_recorder* is None, record can also be None and only spikes are considered.
- **network** (`Network` or subclass, optional (default: None)) – Network which activity will be monitored.
- **gids** (*tuple, optional (default: None)*) – NEST gids of the neurons which should be monitored.
- **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.

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.`**`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.

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Multithreading*
- *Neural groups and 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 inherit from the main class of the chosen graph library (`gt.Graph`, `igraph.Graph` or `networkx.DiGraph`):

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

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

Warning: This object should never be directly modified through the initial library's methods but always using those of NNGT. If, for some reason, you should directly use the methods from the graph library on the object, make sure they do not modify its structure; any modification performed from a method other than those of *`Graph`* subclasses will lead to undefined behaviour!

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

Graph-theoretical models

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

Main module

Overview

For more details regarding the main classes, see:

Graph classes

`nngt.Graph`([nodes, name, weighted, . . .])	The basic graph class, which inherits from a library class such as `graph_tool.Graph`, `networkx.DiGraph`, or `igraph.Graph`.
`nngt.SpatialGraph`([nodes, name, weighted, . . .])	The detailed class that inherits from `Graph` and implements additional properties to describe spatial graphs (i.e.
`nngt.Network`([name, weighted, directed, . . .])	The detailed class that inherits from `Graph` and implements additional properties to describe various biological functions and interact with the NEST simulator.
`nngt.SpatialNetwork`(population[, name, . . .])	Class that inherits from `Network` and `SpatialGraph` to provide a detailed description of a real neural network in space, i.e.

Details

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

Parameters

- **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.
- **name** (*string, optional (default: "Graph")*) – The name of this `Graph` instance.
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.
- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
- **from_graph** (`GraphObject`, optional) – An optional `GraphObject` to serve as base.
- **kwargs** (*optional keywords arguments*) – Optional arguments that can be passed to the graph, e.g. a dict containing information on the synaptic weights (`weights={"distribution": "constant", "value": 2.3}` which is equivalent to `weights=2.3`), the synaptic *delays*, or a `type` information.

Returns **self** (`Graph`)

adjacency_matrix(*types=True*, *weights=True*)

Return the graph adjacency matrix. NB : source nodes are represented by the rows, targets by the corresponding columns.

Parameters

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

Returns **mat** (`scipy.sparse.csr_matrix` 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

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

edges_array

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

edges_attributes

Access edge attributes

New in version 0.7.

static from_file(*filename*, *fmt='auto'*, *separator=' '*, *secondary=';'*, *attributes=None*, *notifier='@'*, *ignore='#'*, *from_string=False*)

Import a saved graph from a file. @todo: implement gml, dot, xml, gt

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. This argument can also be used to load only a subset of the saved attributes.
- **notifier** (*str, optional (default: "@")*) – Symbol specifying the following as meaningfull information. Relevant information is formatted `@info_name=info_value`, where `info_name` is in ("attributes", "directed", "name", "size") and associated `info_value` are of type (`list`, `bool`, `str`, `int`). Additional notifiers are `@type=SpatialGraph/Network/ SpatialNetwork`, which must be followed by the relevant notifiers among `@shape`, `@population`, and `@graph`.
- **from_string** (*bool, optional (default: False)*) – Load from a string instead of a file.

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

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

Parameters

- **matrix** (`scipy.sparse` matrix or `numpy.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.

Returns *`Graph`*

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

Changed in version 1.0: Added *attribute_class* parameter.

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'*, *use_weights=False*)
Betweenness centrality sequence of all nodes and edges.

@todo add node/edge list

Parameters

- **btype** (str, optional (default: `"both"`)) – Type of betweenness to return (`"edge"`, `"node"`-betweenness, or `"both"`).
- **use_weights** (*bool, optional (default: False)*) – Whether to use weighted (True) or simple degrees (False).

Returns

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

get_degrees(*deg_type='total'*, *node_list=None*, *use_weights=False*, *syn_type='all'*)

Degree sequence of all the nodes.

Changed in version 0.9: Added *syn_type* keyword.

Parameters

- **deg_type** (*string, optional (default: "total")*) – Degree type (among 'in', 'out' or 'total').
- **node_list** (*list, optional (default: None)*) – List of the nodes which degree should be returned
- **use_weights** (*bool, optional (default: False)*) – Whether to use weighted (True) or simple degrees (False).
- **syn_type** (*int or str, optional (default: all)*) – Restrict to a given synaptic type ("excitatory", 1, or "inhibitory", -1).

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

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.

Changed in version 1.0: Returns the full dict of edges attributes if called without arguments.

New in version 0.8.

Parameters

- **edge** (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.

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 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.

New in version 1.2.

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*.

See also:

`get_nodes()`, `edges_attributes`

`get_graph_type`()

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

`get_name`()

Get the name of the graph

`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.

New in version 1.2.

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()`, `nodes_attributes`

`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*

`graph_id`
Unique `int` identifying the instance.

`is_directed`()
Whether the graph is directed or not

`is_multigraph`()
Return True if graph is a multigraph, False otherwise.

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

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

`is_weighted`()
Whether the edges have weights

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

Parameters

- **graph** (`Graph` or `SpatialGraph`) – Graph to convert
- **neural_pop** (`NeuralPop`) – Population to associate to the new `Network`
- **copy** (bool, optional (default: `False`)) – Whether the operation should be made in-place on the object or if a new object should be returned.

Notes

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

static make_spatial(*graph*, *shape=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.

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** (*tuple*) – The neighbours of *node*.

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

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, otherwise raises an error.

Returns *The new connection.*

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

New in version 0.7.

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_edges=True*)

Add a list of edges to the graph.

Changed in version 1.0: new_edges checks for duplicate edges and self-loops

Warning: This function currently does not check for duplicate edges between the existing edges and the added ones, but only inside *edge_list*!

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_edges** (*bool, optional (default: True)*) – Check for duplicate edges and self-loops.
- **@todo** (*add example*)

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)

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.

New in version 0.7.

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()`

`node_nb``()`
Number of nodes in the graph

`nodes_attributes`
Access node attributes

New in version 0.7.

See also:

`edge_attributes`, `get_node_attributes`, `new_node_attribute`, `set_node_attribute`,

`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=''*)
set graph name

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

New in version 0.9.

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

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

Parameters

- **syn_type** (*int or string*) – Type of the connection among 'excitatory' (also *1*) or 'inhibitory' (also *-1*).
- **nodes** (int, float or list, optional (default: *None*)) – If *nodes* is an int, number of nodes of the required type that will be created in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of *syn_type* nodes in the graph; if it is a list, ids of the *syn_type* nodes.
- **fraction** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as *syn_type* (if *nodes* is not *None*, it is the fraction of the specified nodes' edges, otherwise it is the fraction of all edges in the graph).

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

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

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.

to_directed_class()
Returns the class to use for empty directed copies.

If you subclass the base classes, use this to designate what directed class to use for *to_directed()* copies.

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

See also:

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

to_undirected_class()
Returns the class to use for empty undirected copies.

If you subclass the base classes, use this to designate what directed class to use for *to_directed()* copies.

type
Type of the graph.

update(*edges=None*, *nodes=None*)
Update the graph using nodes/edges/graphs as input.

Like dict.update, this method takes a graph as input, adding the graph's noes and edges to this graph. It can also take two inputs: edges and nodes. Finally it can take either edges or nodes. To specify only nodes the keyword *nodes* must be used.

The collections of edges and nodes are treated similarly to the add_edges_from/add_nodes_from methods. When iterated, they should yield 2-tuples (u, v) or 3-tuples (u, v, datadict).

Parameters

- **edges** (*Graph object, collection of edges, or None*) – The first parameter can be a graph or some edges. If it has attributes *nodes* and *edges*, then it is taken to be a Graph-like object and those attributes are used as collections of nodes and edges to be added to the graph. If the first parameter does not have those attributes, it is treated as a collection of edges and added to the graph. If the first argument is None, no edges are added.
- **nodes** (*collection of nodes, or None*) – The second parameter is treated as a collection of nodes to be added to the graph unless it is None. If *edges is None* and *nodes is None* an exception is raised. If the first parameter is a Graph, then *nodes* is ignored.

Examples

```
>>> G = nx.path_graph(5)
>>> G.update(nx.complete_graph(range(4,10)))
>>> from itertools import combinations
>>> edges = ((u, v, {'power': u * v})
...          for u, v in combinations(range(10, 20), 2)
...          if u * v < 225)
>>> nodes = [1000]  # for singleton, use a container
>>> G.update(edges, nodes)
```

Notes

It you want to update the graph using an adjacency structure it is straightforward to obtain the edges/nodes from adjacency. The following examples provide common cases, your adjacency may be slightly different and require tweaks of these examples.

```
>>> # dict-of-set/list/tuple
>>> adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
>>> e = [(u, v) for u, nbrs in adj.items() for v in  nbrs]
>>> G.update(edges=e, nodes=adj)
```

```
>>> DG = nx.DiGraph()
>>> # dict-of-dict-of-attribute
>>> adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
>>> e = [(u, v, {'weight': d}) for u, nbrs in adj.items()
...      for v, d in nbrs.items()]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # dict-of-dict-of-dict
>>> adj = {1: {2: {'weight': 1.3}, 3: {'color': 0.7, 'weight':1.2}}}
>>> e = [(u, v, {'weight': d}) for u, nbrs in adj.items()
...      for v, d in nbrs.items()]
>>> DG.update(edges=e, nodes=adj)
```

```
>>> # predecessor adjacency (dict-of-set)
>>> pred = {1: {2, 3}, 2: {3}, 3: {3}}
>>> e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
```

```
>>> # MultiGraph dict-of-dict-of-dict-of-attribute
>>> MDG = nx.MultiDiGraph()
```

(continues on next page)

(continued from previous page)

```
>>> adj = {1: {2: {0: {'weight': 1.3}, 1: {'weight': 1.2}}},
...         3: {2: {0: {'weight': 0.7}}}}
>>> e = [(u, v, ekey, d) for u, nbrs in adj.items()
...       for v, keydict in nbrs.items()
...       for ekey, d in keydict.items()]
>>> MDG.update(edges=e)
```

See also:

add_edges_from() add multiple edges to a graph

add_nodes_from() add multiple nodes to a graph

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

The detailed class that inherits from `Graph` and implements additional properties to describe spatial graphs (i.e. graph where the structure is embedded in space.

Initialize SpatialClass instance.

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(*neurons=None*)

Returns the neurons' positions as a (N, 2) array.

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

shape

class nngt.**Network**(*name='Network'*, *weighted=True*, *directed=True*, *from_graph=None*, *population=None*, *inh_weight_factor=1.0*, ***kwargs*)

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

Initializes `Network` instance.

Parameters

- **nodes** (*int, optional (default: 0)*) – Number of nodes in the graph.
- **name** (*string, optional (default: "Graph")*) – The name of this `Graph` instance.
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weight properties.

- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or undirected.
- **from_graph** (`GraphObject`, optional (default: None)) – An optional `GraphObject` to serve as base.
- **population** (`nngt.NeuralPop`, (default: None)) – An object containing the neural groups and their properties: model(s) to use in NEST to simulate the neurons as well as their parameters.
- **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 ei_network(**args*, ***kwargs*)

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.

New in version 1.0.

Changed in version 0.8: Removed *es_{model, param}* and *is_{model, param}* in favour of *syn_spec* parameter. Renamed *ei_ratio* to *iratio* to match `exc_and_inhib()`.

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()

Return the type 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 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.

nest_gid

nest_gids

neuron_properties(*idx_neuron*)

Properties of a neuron in the graph.

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

Returns *dict of the neuron's properties.*

classmethod num_networks()

Returns the number of alive instances.

population

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

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

Set the synaptic/connection types.

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

Parameters

- **syn_type** (*int or string*) – Type of the connection among 'excitatory' (also *1*) or 'inhibitory' (also *-1*).
- **nodes** (int, float or list, optional (default: *None*)) – If *nodes* is an int, number of nodes of the required type that will be created in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of *syn_type* nodes in the graph; if it is a list, ids of the *syn_type* nodes.
- **fraction** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as *syn_type* (if *nodes* is not *None*, it is the fraction of the specified nodes' edges, otherwise it is the fraction of all edges in the graph).

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

to_nest(*send_only=None*, *use_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.

New in version 1.0.

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.

classmethod uniform_network(**args, **kwargs*)

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

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

Initialize 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*)

Set the synaptic/connection types.

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

Parameters

- **syn_type** (*int or string*) – Type of the connection among 'excitatory' (also *1*) or 'inhibitory' (also *-1*).
- **nodes** (int, float or list, optional (default: *None*)) – If *nodes* is an int, number of nodes of the required type that will be created in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of *syn_type* nodes in the graph; if it is a list, ids of the *syn_type* nodes.
- **fraction** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as *syn_type* (if *nodes* is not *None*, it is the fraction of the specified nodes' edges, otherwise it is the fraction of all edges in the graph).

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

Main functions

`nngt.generate`(di_instructions, **kwargs)	Generate a `Graph` or one of its subclasses from a `dict` containing all the relevant informations.

Continued on next page

Table 3 – continued from previous page

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*, *notifier='@'*, *ignore='#'*)

Load a Graph from a file.

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.
- **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`.
- **ignore** (*str, optional (default: "#")*) – Ignore lines starting with the *ignore* string.

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.

Changed in version 0.7: Added support to write position and Shape when saving *`SpatialGraph`*. Note that saving Shape requires shapely.

@todo: implement gml, dot, xml, 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`, `@population`, 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`).

..versionchanged:: 0.8 Renamed *seed* to *msd*, added *seeds* for multithreading.

Parameters

- **msd** (*int, optional*) – Master seed for numpy *RandomState*. Must be convertible to 32-bit unsigned integers.
- **seeds** (*array of ints, optional*) – Seeds for for *RandomState*. Must be convertible to 32-bit unsigned integers.

`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_module* (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

`nngt.GroupProperty`(size[, constraints, ...])	Class defining the properties needed to create groups of neurons from an existing `GraphClass` or one of its subclasses.
`nngt.MetaGroup`([nodes, name])	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.

Details

class nngt.**GroupProperty**(*size*, *constraints={}*, *neuron_model=None*, *neuron_param={}*, *syn_model=None*, *syn_param={}*)

Class defining the properties needed to create groups of neurons from an existing `GraphClass` or one of its subclasses.

Variables

- ***size*** – `int` Size of the group.
- **constraints** – `dict`, optional (default: {}) Constraints to respect when building the `NeuralGroup`.
- ***neuron_model*** – `string`, optional (default: None) name of the model to use when simulating the activity of this group.
- ***neuron_param*** – `dict`, optional (default: {}) the parameters to use (if they differ from the model's defaults)

Create a new instance of GroupProperties.

Notes

The constraints can be chosen among:

- "avg_deg", "min_deg", "max_deg" (`int`) to constrain the total degree of the nodes
- "avg/min/max_in_deg", "avg/min/max_out_deg", to work with the in/out-degrees
- "avg/min/max_betw" (`double`) to constrain the betweenness centrality
- "in_shape" (`nngt.geometry.Shape`) to chose neurons inside a given spatial region

Examples

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

class `nngt.`**`MetaGroup`**(*nodes=None*, *name=None*, ***kwargs*)

Class defining a meta-group of neurons.

New in version 1.3.

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 *`MetaGroup`* object.

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

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

class `nngt.`**`NeuralGroup`**(*nodes=None*, *neuron_type=1*, *neuron_model=None*, *neuron_param=None*, *name=None*)

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*** – `string`, optional (default: None) the name of the model to use when simulating the activity of this group
- ***neuron_param*** – `dict`, optional (default: {}) the parameters to use (if they differ from the model's defaults)
- ***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.

Changed in version 0.8: Removed *syn_model* and *syn_param*.

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.

`has_model`

`ids`

`is_metagroup`

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

New in version 1.0.

Type Whether the group can be used in a population

`name`

`nest_gids`

`neuron_model`

`neuron_param`

`neuron_type`

`parent`
Return the parent `NeuralPop` of the group

`properties`

`size`

class `nngt.`**`NeuralPop`**(*size=None*, *parent=None*, *meta_groups=None*, *with_models=True*, **args*, ***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.

Changed in version 1.2: Added *meta_groups* parameter.

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_meta_group`(*group*, *name=None*, *replace=False*)
Add an existing meta group to the population.

New in version 1.2.

Parameters

- **group** (`NeuralGroup`) – 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 neurons to a specific group.

Parameters

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

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

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

Changed in version 0.8: Removed *syn_model* and *syn_param*.

Changed in version 1.0: *neurons* can be an int to signify a desired size for the group without actually setting the indices.

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.

New in version 1.2.

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.

Changed in version 0.8: Added *syn_spec* parameter.

Changed in version 1.2: Added *meta_groups* parameter

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()`

`excitatory`
Return the ids of all excitatory nodes inside the population.

New in version 1.3.

`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).

Changed in version 0.8: Added *syn_spec* parameter.

Changed in version 1.2: Added *meta_groups* parameter

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.},
    ...
```

(continues on next page)

(continued from previous page)

```
}
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_group(*neurons*, *numbers=False*)
Return the group of the neurons.

Parameters

- **neurons** (*int or array-like*) – IDs of the neurons for which the group should be returned.
- **numbers** (*bool, optional (default: False)*) – Whether the group identifier should be returned as a number; if `False`, the group names are returned.

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

Parameters

- **groups** (`str`, `int` or array-like, optional (default: `None`)) – Names or numbers of the groups for which the neural properties should be returned.
- **neurons** (int or array-like, optional (default: `None`)) – IDs of the neurons for which parameters should be returned.
- **element** (`list` of `str`, optional (default: `"neuron"`)) – Element for which the parameters should be returned (either `"neuron"` or `"synapse"`).

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

has_models

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

New in version 1.2.

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

New in version 1.3.

is_valid
Whether the population can be used to create a NEST network.

meta_groups

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

New in version 1.3.

parent
Parent `Network`, if it exists, otherwise `None`.

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.

size
Number of neurons in this population.

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.)}
    }
}
```

New in version 0.8.

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.

Changed in version 1.2: Added *neuron_type* and *meta_groups* parameters

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.

NNGT

Package aimed at facilitating the analysis of Neural Networks Growth and Topology.

The library mainly provides algorithms for

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

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 and graph_tool.

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`([nodes, name, weighted, ...])	The basic graph class, which inherits from a library class such as `graph_tool.Graph`, `networkx.DiGraph`, or `igraph.Graph`.
`nngt.GroupProperty`(size[, constraints, ...])	Class defining the properties needed to create groups of neurons from an existing `GraphClass` or one of its subclasses.
`nngt.MetaGroup`([nodes, name])	Class defining a meta-group of neurons.
`nngt.Network`([name, weighted, directed, ...])	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`([nodes, name, weighted, ...])	The detailed class that inherits from `Graph` and implements additional properties to describe spatial graphs (i.e.
`nngt.SpatialNetwork`(population[, name, ...])	Class that inherits from `Network` and `SpatialGraph` to provide a detailed description of a real neural network in space, i.e.

Continued on next page

Table 5 – continued from previous page

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.

Analysis module

Tools to analyze neuronal networks, using either their topological properties, their activity, or more importantly, taking both into account.

Content

nngt.analysis.adjacency_matrix(graph[, ...])	Adjacency matrix of the graph.
nngt.analysis.assortativity(graph[, deg_type])	Assortativity of the graph.
nngt.analysis.bayesian_blocks(t[, x, sigma, ...])	Bayesian Blocks Implementation
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(graph[, nodes, ...])	Return the closeness centrality for each node in *nodes*.
nngt.analysis.clustering(graph)	Global clustering coefficient of the graph.
nngt.analysis.degree_distrib(graph[, ...])	Degree distribution of a graph.
nngt.analysis.diameter(graph)	Pseudo-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.local_clustering(graph[, nodes])	Local clustering coefficient of the 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.num_scc(graph[, listing])	Returns the number of strongly connected components (SCCs).
nngt.analysis.num_wcc(graph[, listing])	Connected components if the directivity of the edges is ignored.

Continued on next page

Table 6 – continued from previous page

nngt.analysis.reciprocity(graph)	Graph reciprocity, defined as $E^{\leftrightarrow}/E$, where $E^{\leftrightarrow}$ and E are, respectively, the number of bidirectional edges and the total number of edges in the graph.
nngt.analysis.spectral_radius(graph[, . . .])	Spectral radius of the graph, defined as the eigenvalue of greatest module.
nngt.analysis.subgraph_centrality(graph[, . . .])	Subgraph centrality, accordign to [Estrada2005], 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(graph)	Same as *nngt.analysis.clustering()* (for networkx users)

Details

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.
  - **'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)
```

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

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

Adjacency matrix of the graph.

Parameters

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

Returns a `csr_matrix`.

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

Assortativity of the graph.

Parameters

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

Returns *a float quantifying the graph assortativity.*

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

Betweenness distribution of a graph.

Changed in version 0.7.

Inclusion of automatic binning.

Parameters

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

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

Parameters

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

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

Global clustering coefficient of the graph. Defined as:

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

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

Degree distribution of a graph.

Changed in version 0.7.

Inclusion of automatic binning.

Parameters

- **graph** (`Graph` or subclass) – the graph to analyze.
- **deg_type** (*string, optional (default: "total")*) – type of degree to consider ("in", "out", or "total").
- **node_list** (*list or numpy.array of ints, optional (default: None)*) – Restrict the distribution to a set of nodes (default: all nodes).
- **use_weights** (*bool, optional (default: False)*) – use weighted degrees (do not take the sign into account: all weights are positive).
- **log** (*bool, optional (default: False)*) – use log-spaced bins.
- **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`**(*graph*)

Pseudo-diameter of the graph

@todo: weighted diameter

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

Local clustering coefficient of the nodes. Defined as

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

Parameters

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

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

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

Returns the number of strongly connected components (SCCs). SCC are ensembles where all contained nodes can reach any other node in the ensemble using the directed edges.

See also:

`num_wcc()`

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

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

See also:

`num_scc()`

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

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

Returns *a float quantifying the reciprocity.*

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

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

Parameters

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

Returns *the spectral radius as a float.*

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

Subgraph centrality, accordign to [Estrada2005], for each node in the graph.

[Estrada2005]: Ernesto Estrada and Juan A. Rodríguez-Velázquez, Subgraph centrality in complex networks, PHYSICAL REVIEW E 71, 056103 (2005), available on ArXiv.

Parameters

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

- **normalize** (*str, optional (default: "max_centrality")*) – Whether the centrality should be normalized. Accepted normalizations are "max_eigenvalue" and "max_centrality"; the first rescales the adjacency matrix by the its largest eigenvalue before taking the exponential, the second sets the maximum centrality to one.

Returns **centralities** (`numpy.ndarray`) – The subgraph centrality of each node.

`nngt.analysis.`**`transitivity`**(*graph*)
Same as `nngt.analysis.clustering()` (for networkx users)

`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.**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.

Core module

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

Content

class `nngt.core.`**`Connections`**

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

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

Compute the delays of the neuronal connections.

Parameters

- **graph** (class:~*nngt.Graph* or subclass) – Graph the nodes belong to.
- **dlist** (class:*numpy.array*, optional (default: None)) – List of user-defined delays).
- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges which value should be updated.
- **distribution** (class:*string*, optional (default: "constant")) – Type of distribution (choose among "constant", "uniform", "lognormal", "gaussian", "user_def", "lin_corr", "log_corr").
- **parameters** (class:*dict*, optional (default: {})) – Dictionary containing the distribution parameters.
- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

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

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

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

Parameters

- **graph** (class:~*nngt.Graph* or subclass) – Graph the nodes belong to.
- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges.
- **pos** (class:*numpy.array*, optional (default: None)) – Positions of the nodes; note that if *graph* has a "position" attribute, *pos* will not be taken into account.

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

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

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

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

Parameters

- **graph** (`Graph` or subclass) – Graph on which edge types will be created.
- **inhib_nodes** (int, float or list, optional (default: *None*)) – If *inhib_nodes* is an int, number of inhibitory nodes in the graph (all connections from inhibitory nodes are inhibitory); if it is a float, ratio of inhibitory nodes in the graph; if it is a list, ids of the inhibitory nodes.
- **inhib_frac** (float, optional (default: *None*)) – Fraction of the selected edges that will be set as refractory (if *inhib_nodes* is not *None*, it is the fraction of the nodes' edges that will become inhibitory, otherwise it is the fraction of all the edges in the graph).

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

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

Parameters

- **graph** (class:*~nngt.Graph* or subclass) – Graph the nodes belong to.
- **elist** (class:*numpy.array*, optional (default: None)) – List of the edges (for user defined weights).
- **wlist** (class:*numpy.array*, optional (default: None)) – List of the weights (for user defined weights).
- **distribution** (class:*string*, optional (default: "constant")) – Type of distribution (choose among "constant", "uniform", "lognormal", "gaussian", "user_def", "lin_corr", "log_corr").
- **parameters** (class:*dict*, optional (default: {})) – Dictionary containing the distribution parameters.
- **noise_scale** (class:*int*, optional (default: None)) – Scale of the multiplicative Gaussian noise that should be applied on the weights.

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

`nngt.core.`**`GraphObject`**
alias of `nngt.core.nx_graph._NxGraph`

Generation module

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

Content

nngt.generation.all_to_all([nodes, . . .])	Generate a graph where all nodes are connected.
nngt.generation.connect_neural_groups(. . . [, . . .])	Function to connect excitatory and inhibitory population with a given graph model.
nngt.generation.connect_neural_types(. . . [, . . .])	Function to connect excitatory and inhibitory population with a given graph model.
nngt.generation.connect_nodes(network, . . .)	Function to connect nodes with a given graph model.
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.gaussian_degree(avg, std[, . . .])	Generate a random graph with constant in- or out-degree.
nngt.generation.newman_watts(coord_nb, . . .)	Generate a small-world graph using the Newman-Watts algorithm.
nngt.generation.price_scale_free(m[, c, . . .])	@todo make the algorithm.
nngt.generation.random_scale_free(in_exp, . . .)	Generate a free-scale graph of given reciprocity and otherwise devoid of correlations.

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.`**`connect_neural_groups`**(*network*, *source_groups*, *target_groups*, *graph_model*, *density=-1.0*, *edges=-1*, *avg_deg=-1.0*, *unit='um'*, *weighted=True*, *directed=True*, *multigraph=False*, ***kwargs*)

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

Changed in version 1.2.0: Allow to use `NeuralGroup` as *source_groups* and *target_groups* arguments.

Changed in version 0.8: Model-specific arguments are now provided as keywords and not through a dict. It is now possible to provide different weights and delays at each call.

@todo make the modifications for only a set of edges

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").
- **kwargs** (*keyword arguments*) – Specific model parameters. or edge attributes specifiers such as *weights* or *delays*.

`nngt.generation.`**`connect_neural_types`**(*network*, *source_type*, *target_type*, *graph_model*, *density=-1.0*, *edges=-1*, *avg_deg=-1.0*, *unit='um'*, *weighted=True*, *directed=True*, *multigraph=False*, ***kwargs*)

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

Changed in version 0.8: Model-specific arguments are now provided as keywords and not through a dict. It is now possible to provide different weights and delays at each call.

@todo make the modifications for only a set of edges

Parameters

- **network** (`Network` or `SpatialNetwork`) – The network to connect.
- **source_type** (*int*) – The type of source neurons (`1` for excitatory, `-1` for inhibitory neurons).
- **target_type** (*int*) – The type of target neurons.
- **graph_model** (*string*) – The name of the connectivity model (among "erdos_renyi", "random_scale_free", "price_scale_free", and "newman_watts").
- **kwargs** (*keyword arguments*) – Specific model parameters. or edge attributes specifiers such as *weights* or *delays*.

`nngt.generation.`**`connect_nodes`**(*network*, *sources*, *targets*, *graph_model*, *density=-1.0*, *edges=-1*, *avg_deg=-1.0*, *unit='um'*, *weighted=True*, *directed=True*, *multigraph=False*, ***kwargs*)

Function to connect nodes with a given graph model.

New in version 1.0.

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").
- **kwargs** (*keyword arguments*) – Specific model parameters. or edge attributes specifiers such as *weights* or *delays*.

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

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

Parameters

- **scale** (*float*) – Characteristic scale for the distance rule. E.g for linear distance- rule, $P(i,j) \propto (1-d_{ij}/scale))$, whereas for the exponential distance-rule, $P(i,j) \propto e^{-d_{ij}/scale}$.
- **rule** (*string, optional (default: 'exp')*) – Rule that will be apply to draw the connections between neurons. Choose among "exp" (exponential), "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=-1.0*, *nodes=0*, *edges=-1*, *avg_deg=-1.0*, *reciprocity=-1.0*, *weighted=True*, *directed=True*, *multigraph=False*, *name='ER'*, *shape=None*, *positions=None*, *population=None*, *from_graph=None*, ***kwargs*)

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

Parameters

- **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'`.

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

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

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

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

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

Parameters

- **avg** (*float*) – The value of the average degree.
- **std** (*float*) – The standard deviation of the Gaussian distribution.
- **degree_type** (*str, optional (default: 'in')*) – The type of the fixed degree, among 'in', 'out' or 'total' @todo: Implement 'total' degree
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **reciprocity** (*double, optional (default: -1 to let it free)*) – @todo: not implemented yet. Fraction of edges that are bidirectional (only for directed graphs – undirected graphs have a reciprocity of 1 by definition)
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – @todo: only for directed graphs for now. Whether the graph is directed or not.

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

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

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

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

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

Parameters

- **in_exp** (*float*) – Absolute value of the in-degree exponent γ_i, such that $p(k_i) \propto k_i^{-\gamma_i}$
- **out_exp** (*float*) – Absolute value of the out-degree exponent γ_o, such that $p(k_o) \propto k_o^{-\gamma_o}$
- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **density** (*double, optional (default: 0.1)*) – Structural density given by *edges / (nodes*nodes)*.
- **edges** (*int (optional)*) – The number of edges between the nodes
- **avg_deg** (*double, optional*) – Average degree of the neurons given by *edges / nodes*.
- **weighted** (*bool, optional (default: True)*) – Whether the graph edges have weights.
- **directed** (*bool, optional (default: True)*) – Whether the graph is directed or not.
- **multigraph** (*bool, optional (default: False)*) – Whether the graph can contain multiple edges between two nodes. can contain multiple edges between two
- **name** (*string, optional (default: "ER")*) – Name of the created graph.
- **shape** (`Shape`, optional (default: None)) – Shape of the neurons' environment.
- **positions** (`numpy.ndarray`, optional (default: None)) – A 2D or 3D array containing the positions of the neurons in space.
- **population** (`NeuralPop`, optional (default: None)) – Population of neurons defining their biological properties (to create a `Network`)
- **from_graph** (`Graph` or subclass, optional (default: None)) – Initial graph whose nodes are to be connected.

Returns **graph_fs** (`Graph`)

Note: As reciprocity increases, requested values of *in_exp* and *out_exp* will be less and less respected as the distribution will converge to a common exponent $\gamma = (\gamma_i + \gamma_o)/2$. Parameter *nodes* is required unless *from_graph* or *population* is provided.

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

@todo make the algorithm.

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

Parameters

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

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

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

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

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

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

Parameters

- **coord_nb** (*int*) – The number of neighbours for each node on the initial topological lattice.
- **proba_shortcut** (*double*) – Probability of adding a new random (shortcut) edge for each existing edge on the initial lattice.

- **nodes** (*int, optional (default: None)*) – The number of nodes in the graph.
- **density** (*double, optional (default: 0.1)*) – Structural density given by *edges* / (*nodes‘*‘nodes*).
- **edges** (*int (optional)*) – The number of edges between the nodes
- **avg_deg** (*double, optional*) – Average degree of the neurons given by *edges* / *nodes*.
- **weighted** (*bool, optional (default: True)*) – 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.Shape`	alias of `nngt.geometry.backup_shape.BackupShape`
`nngt.geometry.culture_from_file`(*args, **kwargs)	
`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.
`nngt.geometry.shapes_from_file`(*args, **kwargs)	

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.

```python
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

```python
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)

```

(continues on next page)

(continued from previous page)

```
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

`nngt.geometry.`**`Shape`**
alias of `nngt.geometry.backup_shape.BackupShape`

`nngt.geometry.`**`culture_from_file`**`(`**args*, ***kwargs*`)`

`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`**`(`**args*, ***kwargs*`)`

`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`)

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

Various tools for random number generation, array searching and type testing.

`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*.

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

Error raised when an argument is invalid.

`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.

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.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.
`nngt.plot.edge_attributes_distribution`(…)	Return node *attributes* for a set of *nodes*.
`nngt.plot.node_attributes_distribution`(…)	Return node *attributes* for a set of *nodes*.
`nngt.plot.palette`([numbers])	

Details

This modules provides the following features:

- plotting the distribution of some attribute over the graph
- basic graph plotting

- animation of some recorded activity

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*)

Bases: `nngt.plot.animations._SpikeAnimator`, `matplotlib.animation.FuncAnimation`

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'.

new_frame_seq()

new_saved_frame_seq()

on_click(*event*)

on_keyboard_press(*kb_event*)

on_keyboard_release(*kb_event*)

save(*filename*, *writer=None*, *fps=None*, *dpi=None*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *extra_anim=None*, *savefig_kwargs=None*)

Saves a movie file by drawing every frame.

Parameters

- **filename** (*str*) – The output filename, e.g., `mymovie.mp4`.

- **writer** (`MovieWriter` or str, optional) – A *MovieWriter* instance to use or a key that identifies a class to use, such as 'ffmpeg'. If `None`, defaults to **:rc:'animation.writer'**.
- **fps** (*number, optional*) – Frames per second in the movie. Defaults to `None`, which will use the animation's specified interval to set the frames per second.
- **dpi** (*number, optional*) – Controls the dots per inch for the movie frames. This combined with the figure's size in inches controls the size of the movie. If `None`, defaults to **:rc:'savefig.dpi'**.
- **codec** (*str, optional*) – The video codec to be used. Not all codecs are supported by a given `MovieWriter`. If `None`, default to **:rc:'animation.codec'**.
- **bitrate** (*number, optional*) – Specifies the number of bits used per second in the compressed movie, in kilobits per second. A higher number means a higher quality movie, but at the cost of increased file size. If `None`, defaults to **:rc:'animation.bitrate'**.
- **extra_args** (*list, optional*) – List of extra string arguments to be passed to the underlying movie utility. If `None`, defaults to **:rc:'animation.extra_args'**.
- **metadata** (*Dict[str, str], optional*) – Dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.
- **extra_anim** (*list, optional*) – Additional *Animation* objects that should be included in the saved movie file. These need to be from the same *matplotlib.figure.Figure* instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.
- **savefig_kwargs** (*dict, optional*) – Is a dictionary containing keyword arguments to be passed on to the *savefig* command which is called repeatedly to save the individual frames.

Notes

fps, codec, bitrate, extra_args, metadata are used to construct a `MovieWriter` instance and can only be passed if *writer* is a string. If they are passed as non-*None* and `writer` is a `MovieWriter`, a *RuntimeError* will be raised.

`save_movie`(*filename*, *fps=30*, *video_encoder='html5'*, *codec='h264'*, *bitrate=-1*, *start=None*, *stop=None*, *interval=None*, *num_frames=None*, *metadata=None*)

Save the animation to a movie file.

Parameters

- **filename** (`str`) – Name of the file where the movie will be saved.
- **fps** (*int, optional (default: 30)*) – Frame per second.
- **video_encoder** (`str`, optional (default 'html5')) – Movie encoding format; either 'ffmpeg', 'html5', or 'imagemagick'.
- **codec** (`str`, optional (default: "h264")) – Codec to use for writing movie; if None, default *animation.codec* from *matplotlib* will be used.
- **bitrate** (*int, optional (default: -1)*) – Controls size/quality tradeoff for movie. Default (-1) lets utility auto-determine.
- **start** (*float, optional (default: initial time)*) – Start time, corresponding to the first spike time that will appear on the video.
- **stop** (*float, optional (default: final time)*) – Stop time, corresponding to the last spike time that will appear on the video.

- **interval** (*int, optional (default: None)*) – Timestep increment for each new frame. Default saves all timesteps (often heavy). E.g. setting *interval* to 10 will make the file 10 times lighter.
- **num_frames** (*int, optional (default: None)*) – Total number of frames that should be saved.
- **metadata** (`dict`, optional (default: None)) – Metadata for the video (e.g. 'title', 'artist', 'comment', 'copyright')

Notes

- `ffmpeg` is required for 'ffmpeg' and 'html5' encoders. To get available formats, type `ffmpeg -formats` in a terminal; type `ffmpeg -codecs | grep EV` for available codecs.
- Imagemagick is required for 'imagemagick' encoder.

`set_axis` (*axis*, *xlabel*, *ylabel*, *lines*, *xdata=None*, *ydata=None*, ***kwargs*)
Setup an axis.

Parameters

- **axis** (`matplotlib.axes.Axes` object)
- **xlabel** (*str*)
- **ylabel** (*str*)
- **lines** (list of `matplotlib.lines.Line2D` objects)
- **xdata** (*1D array-like, optional (default: None)*)
- **ydata** (*1D array-like, optional (default: None)*)
- ****kwargs** (*dict, optional (default: {})*) – Optional arguments ("xlim" or "ylim", 2-tuples; "set_xticks", bool).

`steps = [1, 5, 10, 20, 25, 50, 100, 200, 250, 500, 1000, 2000, 2500, 5000, 10000, 2500`

`to_html5_video` (*embed_limit=None*)
Returns animation as an HTML5 video tag.

This saves the animation as an h264 video, encoded in base64 directly into the HTML5 video tag. This respects the rc parameters for the writer as well as the bitrate. This also makes use of the `interval` to control the speed, and uses the `repeat` parameter to decide whether to loop.

`to_jshtml` (*fps=None*, *embed_frames=True*, *default_mode=None*)
Generate HTML representation of the animation

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*)
Bases: `nngt.plot.animations._SpikeAnimator`, `matplotlib.animation.FuncAnimation`

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()`.

`new_frame_seq`()

`new_saved_frame_seq`()

`on_click`(*event*)

`on_keyboard_press`(*kb_event*)

`on_keyboard_release`(*kb_event*)

`save`(*filename*, *writer=None*, *fps=None*, *dpi=None*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *extra_anim=None*, *savefig_kwargs=None*)

Saves a movie file by drawing every frame.

Parameters

- **filename** (*str*) – The output filename, e.g., `mymovie.mp4`.
- **writer** (`MovieWriter` or str, optional) – A *MovieWriter* instance to use or a key that identifies a class to use, such as 'ffmpeg'. If `None`, defaults to **:rc:‘animation.writer‘**.
- **fps** (*number, optional*) – Frames per second in the movie. Defaults to `None`, which will use the animation's specified interval to set the frames per second.
- **dpi** (*number, optional*) – Controls the dots per inch for the movie frames. This combined with the figure's size in inches controls the size of the movie. If `None`, defaults to **:rc:‘savefig.dpi‘**.
- **codec** (*str, optional*) – The video codec to be used. Not all codecs are supported by a given `MovieWriter`. If `None`, default to **:rc:‘animation.codec‘**.
- **bitrate** (*number, optional*) – Specifies the number of bits used per second in the compressed movie, in kilobits per second. A higher number means a higher quality movie, but at the cost of increased file size. If `None`, defaults to **:rc:‘animation.bitrate‘**.
- **extra_args** (*list, optional*) – List of extra string arguments to be passed to the underlying movie utility. If `None`, defaults to **:rc:‘animation.extra_args‘**.
- **metadata** (*Dict[str, str], optional*) – Dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

- **extra_anim** (*list, optional*) – Additional *Animation* objects that should be included in the saved movie file. These need to be from the same *matplotlib.figure.Figure* instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.
- **savefig_kwargs** (*dict, optional*) – Is a dictionary containing keyword arguments to be passed on to the *savefig* command which is called repeatedly to save the individual frames.

Notes

fps, codec, bitrate, extra_args, metadata are used to construct a `MovieWriter` instance and can only be passed if *writer* is a string. If they are passed as non-*None* and `writer` is a `MovieWriter`, a *RuntimeError* will be raised.

`save_movie`(*filename*, *fps=30*, *video_encoder='html5'*, *codec='h264'*, *bitrate=-1*, *start=None*, *stop=None*, *interval=None*, *num_frames=None*, *metadata=None*)

Save the animation to a movie file.

Parameters

- **filename** (`str`) – Name of the file where the movie will be saved.
- **fps** (*int, optional (default: 30)*) – Frame per second.
- **video_encoder** (`str`, optional (default 'html5')) – Movie encoding format; either 'ffmpeg', 'html5', or 'imagemagick'.
- **codec** (`str`, optional (default: "h264")) – Codec to use for writing movie; if None, default *animation.codec* from *matplotlib* will be used.
- **bitrate** (*int, optional (default: -1)*) – Controls size/quality tradeoff for movie. Default (-1) lets utility auto-determine.
- **start** (*float, optional (default: initial time)*) – Start time, corresponding to the first spike time that will appear on the video.
- **stop** (*float, optional (default: final time)*) – Stop time, corresponding to the last spike time that will appear on the video.
- **interval** (*int, optional (default: None)*) – Timestep increment for each new frame. Default saves all timesteps (often heavy). E.g. setting *interval* to 10 will make the file 10 times lighter.
- **num_frames** (*int, optional (default: None)*) – Total number of frames that should be saved.
- **metadata** (`dict`, optional (default: None)) – Metadata for the video (e.g. 'title', 'artist', 'comment', 'copyright')

Notes

- `ffmpeg` is required for 'ffmpeg' and 'html5' encoders. To get available formats, type `ffmpeg -formats` in a terminal; type `ffmpeg -codecs | grep EV` for available codecs.
- Imagemagick is required for 'imagemagick' encoder.

`set_axis`(*axis*, *xlabel*, *ylabel*, *lines*, *xdata=None*, *ydata=None*, ***kwargs*)

Setup an axis.

Parameters

- **axis** (`matplotlib.axes.Axes` object)
- **xlabel** (*str*)
- **ylabel** (*str*)
- **lines** (list of `matplotlib.lines.Line2D` objects)
- **xdata** (*1D array-like, optional (default: None)*)
- **ydata** (*1D array-like, optional (default: None)*)
- ****kwargs** (*dict, optional (default: {})*) – Optional arguments ("xlim" or "ylim", 2-tuples; "set_xticks", bool).

`steps = [1, 5, 10, 20, 25, 50, 100, 200, 250, 500, 1000, 2000, 2500, 5000, 10000, 2500`

`to_html5_video`(*embed_limit=None*)

Returns animation as an HTML5 video tag.

This saves the animation as an h264 video, encoded in base64 directly into the HTML5 video tag. This respects the rc parameters for the writer as well as the bitrate. This also makes use of the `interval` to control the speed, and uses the `repeat` parameter to decide whether to loop.

`to_jshtml`(*fps=None*, *embed_frames=True*, *default_mode=None*)

Generate HTML representation of the animation

`nngt.plot.`**`draw_network`**(*network*, *nsize='total-degree'*, *ncolor='group'*, *nshape='o'*, *nborder_color='k'*, *nborder_width=0.5*, *esize=1.0*, *ecolor='k'*, *ealpha=0.5*, *max_nsize=5.0*, *max_esize=2.0*, *curved_edges=False*, *threshold=0.5*, *decimate_connections=None*, *spatial=True*, *restrict_sources=None*, *restrict_targets=None*, *restrict_nodes=None*, *show_environment=True*, *fast=False*, *size=(600, 600)*, *xlims=None*, *ylims=None*, *dpi=75*, *axis=None*, *colorbar=False*, *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", or "betweenness".
- **ncolor** (*float, array of floats or string, optional (default: 0.5)*) – Color of the nodes; if a float in [0, 1], position of the color in the current palette, otherwise a string that correlates the color to a node attribute among "in/out/total-degree", "betweenness" or "group".
- **nshape** (*char, 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.

- **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.
- **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.
- **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. This method leads to less pretty plots and zooming on the graph will make the edges start or ending in places that will differ more or less strongly from the actual node positions.
- **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.
- **show** (*bool, optional (default: True)*) – Display the plot immediately.
- **axis** (*matplotlib axis, optional (default: create new axis)*) – Axis on which the network will be plotted.
- ****kwargs** (*dict*) – Optional keyword arguments including *node_cmap* to set the nodes colormap (default is "magma" for continuous variables and "Set1" for groups) and the boolean *simple_nodes* to make node plotting faster.

`nngt.plot.`**`palette`**(*numbers=None*)

`nngt.plot.`**`degree_distribution`**(*network, deg_type='total', nodes=None, num_bins='doane', use_weights=False, logx=False, logy=False, axis=None, axis_num=None, colors=None, norm=False, show=True, title=None, **kwargs*)

Plotting the degree distribution of a graph.

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** (*int or N-tuple, optional (default: 'auto'):*) – Number of bins used to sample the distribution. Defaults to unsupervised Bayesian blocks method.

- **use_weights** (*bool, optional (default: False)*) – Use weighted degrees (do not take the sign into account : only the magnitude of the weights is considered).
- **logx** (*bool, optional (default: False)*) – Use log-spaced bins.
- **logy** (*bool, optional (default: False)*) – Use logscale for the degree count.
- **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.
- **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.`**`betweenness_distribution`**(*network*, *btype='both'*, *use_weights=True*, *nodes=None*, *logx=False*, *logy=False*, *num_nbins='auto'*, *num_ebins=None*, *axes=None*, *colors=None*, *norm=False*, *show=True*, ***kwargs*)

Plotting the betweenness distribution of a graph.

Parameters

- **graph** (`Graph` or subclass) – the graph to analyze.
- **btype** (*string, optional (default: "both")*) – type of betweenness to display ("node", "edge" or "both")
- **use_weights** (*bool, optional (default: True)*) – use weighted degrees (do not take the sign into account : all weights are positive).
- **nodes** (*list or numpy.array of ints, optional (default: all nodes)*) – Restrict the distribution to a set of nodes (taken into account only for the node attribute).
- **logx** (*bool, optional (default: False)*) – use log-spaced bins.
- **logy** (*bool, optional (default: False)*) – use logscale for the degree count.
- **num_nbins** (*int or 'auto', optional (default: 'auto'):*) – Number of bins used to sample the node distribution. Defaults to unsupervised Bayesian blocks method.
- **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.
- **show** (*bool, optional (default: True)*) – Show the Figure right away if True, else keep it warm for later use.

`nngt.plot.`**`edge_attributes_distribution`**(*network*, *attributes*, *edges=None*, *num_bins='auto'*, *logx=False*, *logy=False*, *norm=False*, *title=None*, *colors=None*, *show=True*, ***kwargs*)

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

New in version 1.0.3.

Parameters

- **network** (`Graph`) – The graph where the *nodes* belong.
- **attributes** (*str or list*) – Attributes which should be returned (e.g. "betweenness", "delay", "weights").
- **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.

`nngt.plot.`**`node_attributes_distribution`**(*network*, *attributes*, *nodes=None*, *num_bins='auto'*, *logx=False*, *logy=False*, *norm=False*, *title=None*, *colors=None*, *show=True*, ***kwargs*)

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" * "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.

`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*, *nodes=None*, *title=None*, *show=True*)

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

Parameters

- **network** (*Graph*) – The graph where the *nodes* belong.
- **reference_attribute** (*str or array-like*) – Attribute which should serve as reference, among:
  - "betweenness"
  - "clustering"
  - "in-degree", "out-degree", "total-degree"
  - "subgraph_centrality"
  - "b2" (requires NEST)
  - "firing_rate" (requires NEST)
  - a custom array of values, in which case one entry per node in *nodes* is required.
- **other_attributes** (*str or list*) – Attributes that will be compared to the reference.
- **nodes** (*list, optional (default: all nodes)*) – Nodes for which the attributes should be returned.

Known bugs

- Graph I/O confirmed not working with *graph_tool <= 2.19* when using edge attributes. Confirmed working with *graph_tool >= 2.22*.
- Plotting `SpatialGraph` with *networkx* does not work.

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*
- *Multithreading*
- *Neural groups and populations*
- *Interacting with the NEST simulator*

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 *Multithreading* 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.nodes_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.nodes_attributes['size'])
print(g2.nodes_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.nodes_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)))
g3.new_edge_attribute('rank', 'int')
g3.set_edge_attribute('rank', val=2, edges=edges[:3, :])
print(g3.edges_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.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.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 Complex populations: `NeuralGroup` and `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_neural_groups()` function.

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

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

# make population (without NEST models)
```

(continues on next page)

(continued from previous page)

```
pop = nngt.NeuralPop.from_groups(
    (g1, g2), ("left", "right"), with_models=False)

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


'''
Connect the groups
'''

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

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

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

For more details, see the full page on *Neural groups and populations*.

2.3.6 Real neuronal culture 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': -57., '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.})
# bursters
params2.update({'b': 30., 'V_reset': -50., 'tau_w': 500.})

oscill = nngt.NeuralGroup(
    nodes=400, neuron_model='aeif_psc_alpha', neuron_param=params1)
burst = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', neuron_param=params2)
```

(continues on next page)

(continued from previous page)

```
adapt = nngt.NeuralGroup(
    nodes=200, neuron_model='aeif_psc_alpha', 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=250., 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*.

2.3.7 Using the graph library of the NNGT object

Warning: This behavior will change in the next major release (NNGT 2.0) where the specific library graph object will be handled through composition instead of direct inheritance. It will then be accessible via a `graph` property of the `Graph` instance.

As mentionned in the installation and introduction, NNGT uses existing graph library objects to store the graph. The library was designed so that most of the functions of the underlying graph library can be used directly on the `Graph` object.

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 using the underlying library.

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, harmonic=True))
>>> gt.draw.graph_draw(g)
>>> nngt.plot.draw_network(g)
>>> plt.show()
```

Example using igraph

```
>>> import igraph as ig
>>> import matplotlib.pyplot as plt
>>> print(g.closeness(mode='out'))
>>> ig.plot(g)
>>> nngt.plot.draw_network(g)
>>> plt.show()
```

Example using networkx

```
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>> print(nx.closeness_centrality(g))
>>> nx.draw(g)
>>> 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 even when they compute it, check how they do it!

Go to other tutorials:

- *Intro & user manual*
- *Graph generation*
- *Multithreading*
- *Neural groups and populations*
- *Interacting with the NEST simulator*
- *Activity analysis*
- *Properties of graph components*

2.4 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.

2.4.1 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.

2.4.2 Recording a simulation

```
nngt.database.log_simulation_start(net, "nest-2.14")
nest.Simulate(1000.)
nngt.database.log_simulation_end()
```

2.4.3 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

class `nngt.database.db_generation.`**`Activity`**(**args*, ***kwargs*)

Class detailing the network's simulated activity.

`DoesNotExist`

alias of `ActivityDoesNotExist`

`id = <IntegerField: Activity.id>`

`raster = <PickledField: Activity.raster>`

Raster of the simulated activity.

`simulations`

class `nngt.database.db_generation.`**`Computer`**(**args*, ***kwargs*)

Class containing informations about the conputer.

`DoesNotExist`

alias of `ComputerDoesNotExist`

`cores = <IntegerField: Computer.cores>`

Number of cores returned by `psutil.cpu_count()` or `-1`

`id = <IntegerField: Computer.id>`

`name = <TextField: Computer.name>`

Name from `platform.node()` or `"unknown"`

`platform = <TextField: Computer.platform>`

System information from `platform.platform()`

`python = <TextField: Computer.python>`

Python version given by `platform.python_version()`

`ram = <IntegerField: Computer.ram>`

Total memory given by `psutil.virtual_memory().total` (long) or `-1`

`simulations`

class `nngt.database.db_generation.`**`Connection`**(**args*, ***kwargs*)

Class detailing the existing connections in the network: a couple of pre- and post-synaptic neurons and a synapse.

DoesNotExist
alias of `ConnectionDoesNotExist`

id = <IntegerField: Connection.id>

post = <ForeignKeyField: Connection.post>

post_id = <ForeignKeyField: Connection.post>

pre = <ForeignKeyField: Connection.pre>

pre_id = <ForeignKeyField: Connection.pre>

simulations

synapse = <ForeignKeyField: Connection.synapse>

synapse_id = <ForeignKeyField: Connection.synapse>

`nngt.database.db_generation.`**migrate**(**operations*, ***kwargs*)

class `nngt.database.db_generation.`**NeuralNetwork**(**args*, ***kwargs*)
Class containing informations about the neural network.

DoesNotExist
alias of `NeuralNetworkDoesNotExist`

compressed_file = <LongCompressedField: NeuralNetwork.compressed_file>
Compressed (bz2) string of the graph from `str(graph)`; once uncompressed, can be loaded using `Graph.from_file(name, from_string=True)`.

directed = <IntegerField: NeuralNetwork.directed>
Whether the graph is directed or not

edges = <IntegerField: NeuralNetwork.edges>
Number of edges.

id = <IntegerField: NeuralNetwork.id>

network_type = <TextField: NeuralNetwork.network_type>
Type of the network from Graph.type

nodes = <IntegerField: NeuralNetwork.nodes>
Number of nodes.

simulations

weight_distribution = <TextField: NeuralNetwork.weight_distribution>
Name of the weight_distribution used.

weighted = <IntegerField: NeuralNetwork.weighted>
Whether the graph is weighted or not.

class `nngt.database.db_generation.`**Neuron**(**args*, ***kwargs*)
Base class that will be modified to contain all the properties of the neurons used during a simulation.

DoesNotExist
alias of `NeuronDoesNotExist`

id = <IntegerField: Neuron.id>

int_connections

out_connections

class `nngt.database.db_generation.`**Simulation**(**args*, ***kwargs*)
Class containing all informations about the simulation properties.

DoesNotExist
alias of `SimulationDoesNotExist`

activity = <ForeignKeyField: Simulation.activity>
Activity table entry where the simulated activity is described.

activity_id = <ForeignKeyField: Simulation.activity>

completion_time = <DateTimeField: Simulation.completion_time>
Date and time at which the simulation ended.

computer = <ForeignKeyField: Simulation.computer>
Computer table entry where the computer used is defined.

computer_id = <ForeignKeyField: Simulation.computer>

connections = <ForeignKeyField: Simulation.connections>
Connection table entry where the connections are described.

connections_id = <ForeignKeyField: Simulation.connections>

grnd_seed = <IntegerField: Simulation.grnd_seed>
Master seed of the simulation.

id = <IntegerField: Simulation.id>

local_seeds = <PickledField: Simulation.local_seeds>
List of the local threads seeds.

network = <ForeignKeyField: Simulation.network>
Network table entry where the simulated network is described.

network_id = <ForeignKeyField: Simulation.network>

pop_sizes = <PickledField: Simulation.pop_sizes>
Pickled list containing the group sizes.

population = <PickledField: Simulation.population>
Pickled list containing the neural group names.

resolution = <FloatField: Simulation.resolution>
Timestep used to simulate the components of the neural network

simulated_time = <FloatField: Simulation.simulated_time>
Virtual time that was simulated for the neural network.

simulator = <TextField: Simulation.simulator>
Name of the neural simulator used (NEST, Brian…)

start_time = <DateTimeField: Simulation.start_time>
Date and time at which the simulation started.

class `nngt.database.db_generation.`**Synapse**(**args*, ***kwargs*)
Base class that will be modified to contain all the properties of the synapses used during a simulation.

DoesNotExist
alias of `SynapseDoesNotExist`

connections

id = <IntegerField: Synapse.id>

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index