BSB Documentation

Release 3.10.2

Robin De Schepper

May 31, 2022

CONTENTS:

CHAPTER

ONE

INSTALLATION GUIDE

The scaffold framework can be installed using Pip for Python 3

```
pip install bsb
```

You can verify that the installation works with

```
bsb make-config
bsb -v=3 compile -x=50 -z=50 -p
```

This should generate a template config and an HDF5 file in your current directory and open a plot of the generated network, it should contain a column of `base_type` cells. If no errors occur you are ready to *get started*.

1.1 Installing for NEURON

The BSB's installation will install NEURON from PyPI if no `NEURON` installation is detected by `pip`. This means that any custom installations that rely on `PYTHONPATH` to be detected at runtime but aren't registered as an installed package to pip will be overwritten. Because it is quite common for NEURON to be incorrectly installed from pip's point of view, you have to explicitly ask the BSB installation to install it:

```
pip install bsb[neuron]
```

After installation of the dependencies you will have to describe your cell models using Arborize's `NeuronModel` template and import your Arborize cell models module into a `MorphologyRepository`:

```
$ bsb
> open mr morphologies.hdf5 --create
<repo 'morphologies.hdf5'> arborize my_models
numprocs=1
Importing MyCell1
Importing MyCell2
...
<repo 'morphologies.hdf5'> exit
```

This should allow you to use `morphologies.hdf5` and the morphologies contained within as the *morphology_repository* of the *output* node in your config:

```
{
    "name": "Example config",
    "output": {
    "format": "bsb.output.HDF5Formatter",
```

(continues on next page)

(continued from previous page)

```
    "file": "my_network.hdf5",
    "morphology_repository": "morphologies.hdf5"
    }
}
```

1.2 Installing NEST

The BSB currently runs a fork of NEST 2.18, to install it, follow the instructions below. The instructions assume you are using pyenv for virtual environments.

```
sudo apt-get update && apt-get install -y openmpi-bin libopenmpi-dev
git clone git@github.com:dbbs-lab/nest-simulator
cd nest-simulator
mkdir build && cd build
export PYTHON_CONFIGURE_OPTS="--enable-shared"
# Any Python 3.8+ version built with `--enable-shared` will do
PYVER_M=3.9
PYVER=$PYVER_M.0
VENV=nest-218
pyenv install $PYVER
pyenv virtualenv $PYVER $VENV
pyenv local nest-218
cmake .. \
  -DCMAKE_INSTALL_PREFIX=$(pyenv root)/versions/$VENV \
  -Dwith-mpi=ON \
  -Dwith-python=3 \
  -DPYTHON_LIBRARY=$(pyenv root)/versions/$PYVER/lib/libpython$PYVER_M.so \
  -DPYTHON_INCLUDE_DIR=$(pyenv root)/versions/$PYVER/include/python$PYVER_M
make install -j8
```

Confirm your installation with:

```
python -c "import nest; nest.test()"
```

Note: There might be a few failed tests related to `NEST_DATA_PATH` but this is OK.

CHAPTER

TWO

GETTING STARTED

2.1 First steps

The scaffold provides a simple command line interface (CLI) to compile network architectures and run simulations.

To start, let's create ourselves a project directory and a template configuration:

```
mkdir my_brain
cd my_brain
bsb make-config
```

See *Command Line Interface* for a full list of CLI commands.

The `make-config` command makes a template configuration file:

```
{
  "name": "Empty template",
  "network_architecture": {
    "simulation_volume_x": 400.0,
    "simulation_volume_z": 400.0
  },
  "output": {
    "format": "bsb.output.HDF5Formatter"
  },
  "layers": {
    "base_layer": {
      "thickness": 100
    }
  },
  "cell_types": {
    "base_type": {
      "placement": {
        "class": "bsb.placement.ParticlePlacement",
        "layer": "base_layer",
        "soma_radius": 2.5,
        "density": 3.9e-4
      },
      "morphology": {
        "class": "bsb.morphologies.NoGeometry"
      },
      "plotting": {
        "display_label": "Template cell",
```

(continues on next page)

(continued from previous page)

```
        "color": "#E62314",
        "opacity": 0.5
      }
    }
  },
  "after_placement": {

  },
  "connection_types": {

  },
  "after_connectivity": {

  },
  "simulations": {

  }
}
```

The configuration is laid out to be as self explanatory as possible. For a full walkthrough of all parts see the *Configuration reference*.

To convert the abstract description in the configuration file into a concrete network file with cell positions and connections run the `compile` command:

```
bsb -c network_configuration.json compile -p
```

Note: You can leave off the `-c` (or `--config`) flag in this case as `network_configuration.json` is the default config that `bsb compile` will look for. The `-p` (or `--plot`) flag will plot your network afterwards

2.2 First script

The BSB is also a library that can be imported into Python scripts. You can load configurations and adapt the loaded object before constructing a network with it to programmatically alter the network structure.

Let's go over an example first script that creates 5 networks with different densities of `base_type`.

To use the scaffold in your script you should import the `bsb.core.Scaffold` and construct a new instance by passing it a `bsb.config.ScaffoldConfig`. The only provided configuration is the `bsb.config.JSONConfig`. To load a configuration file, construct a JSONConfig object providing the *file* keyword argument with a path to the configuration file:

```
from bsb.core import Scaffold
from bsb.config import JSONConfig
from bsb.reporting import set_verbosity

config = JSONConfig(file="network_configuration.json")
set_verbosity(3) # This way we can follow what's going on.
scaffold = Scaffold(config)
```

Note: The verbosity is 1 by default, which only displays errors. You could also add a `verbosity` attribute to the root node of the `network_configuration.json` file to set the verbosity.

Let's find the `base_type` cell configuration:

```
base_type = scaffold.get_cell_type("base_type")
```

The next step is to adapt the `base_type` cell density each iteration. The location of the attributes on the Python objects mostly corresponds to their location in the configuration file. This means that:

```
"base_type": {
  "placement": {
    "density": 3.9e-4,
    ...
  },
  ...
}
```

will be stored in the Python `CellType` object under `base_type.placement.density`:

```
max_density = base_type.placement.density
for i in range(5):
  base_type.placement.density = i * 20 / 100 * max_density
  scaffold.compile_network()

  scaffold.plot_network_cache()

  scaffold.reset_network_cache()
```

Warning: If you don't use `reset_network_cache()` between `compile_network()` calls, the new cells will just be appended to the previous ones. This might lead to confusing results.

2.2.1 Full code example

```
from bsb.core import Scaffold
from bsb.config import JSONConfig
from bsb.reporting import set_verbosity

config = JSONConfig(file="network_configuration.json")
set_verbosity(3) # This way we can follow what's going on.
scaffold = Scaffold(config)

base_type = scaffold.get_cell_type("base_type_cell")
max_density = base_type.placement.density

for i in range(5):
  base_type.placement.density = i * 20 / 100 * max_density
  scaffold.compile_network()
```

(continues on next page)

(continued from previous page)

```
  scaffold.plot_network_cache()

  scaffold.reset_network_cache()
```

2.3 Network compilation

`compilation` is the process of creating an output containing the constructed network with cells placed according to the specified placement strategies and connected to each other according to the specified connection strategies:

```
from bsb.core import Scaffold
from bsb.config import JSONConfig
      import os

config = JSONConfig(file="network_configuration.json")

# The configuration provided in the file can be overwritten here.
# For example:
config.cell_types["some_cell"].placement.some_parameter = 50
config.cell_types["some_cell"].plotting.color = os.getenv("ENV_PLOTTING_COLOR", "black")

scaffold = Scaffold(config)
scaffold.compile_network()
```

The configuration object can be freely modified before compilation, although values that depend on eachother - i.e. layers in a stack - will not update each other.

2.4 Network simulation

Simulations can be executed from configuration in a managed way using:

```
scaffold.run_simulation(name)
```

This will load the simulation configuration associated with `name` and create an adapter for the simulator. An adapter translates the scaffold configuration into commands for the simulator. In this way scaffold adapters are able to prepare simulations in external simulators such as NEST or NEURON for you. After the simulator is prepared the simulation is ran.

For more control over the interface with the simulator, or finer control of the configuration, the process can be split into parts. The adapter to the interface of the simulator can be ejected and its configuration can be modified:

```
adapter = scaffold.create_adapter(name)
adapter.devices["input_stimulation"].parameters["rate"] = 40
```

You can then use this adapter to prepare the simulator for the configured simulation:

```
simulator = adapter.prepare()
```

After preparation the simulator is primed, but can still be modified directly accessing the interface of the simulator itself. For example to create 5 extra cells in a NEST simulation on top of the prepared configuration one could:

```
cells = simulator.Create("iaf_cond_alpha", 5)
print(cells)
```

You'll notice that the IDs of those cells won't start at 1 as would be the case for an empty simulation, because the `prepare` statement has already created cells in the simulator.

After custom interfacing with the simulator, the adapter can be used to run the simulation:

```
adapter.simulate()
```

2.4.1 Full code example

```
adapter = scaffold.create_adapter(name)
adapter.devices["input_stimulation"].parameters["rate"] = 40
simulator = adapter.prepare()
cells = simulator.Create("iaf_cond_alpha", 5)
print(cells)
adapter.simulate()
```

2.5 Using Cell Types

Cell types are obtained by name using *bsb.get_cell_type(name)*. And the associated cells either currently in the network cache or in persistent storage can be fetched with *bsb.get_cells_by_type(name)*. The columns of such a set are the scaffold id of the cell, followed by the type id and the xyz position.

A collection of all cell types can be retrieved with *bsb.get_cell_types()*:

```
for cell_type in scaffold.get_cell_types():
  cells = scaffold.get_cells_by_type(cell_type.name)
  for cell in cells:
    print("Cell id {} of type {} at position {}.".format(cell[0], cell[1], cell[2:5]))
```

CHAPTER

THREE

COMMAND LINE INTERFACE

There are 2 entry points in the command line interface:

- **A command**: Can be written in a command line prompt such as the Terminal on Linux or CMD on Windows.
- **The shell**: Can be opened by giving typing `bsb` into a command line prompt.

3.1 Scaffold shell

The scaffold shell is an interactive environment where commands can be given. Unlike with the command line your state is maintained in between commands.

3.1.1 Opening the shell

Open your favorite command line prompt and if the scaffold package is succesfully installed the `bsb` command should be available.

You can close the shell by typing `exit`.

3.1.2 The base state

After opening the shell it will be in the base (default) state. In this state you have access to several commands like opening morphology repositories or hdf5 files.

List of base commands

- `open mr <filename>`: Open a morphology repository. See *List of mr commands*
- `open hdf5 <filename>`: Open an HDF5 file. See *List of hdf5 commands:*

3.1.3 The morphology repository state

In this state you can modify the morphology repository. After you've opened a repository the shell will display a prefix:

```
repo <filename>:
```

List of mr commands

- `list all`: Show a list of all morphologies available in the repository.
- `list voxelized`: Show a list of all morphologies with voxel cloud information available.
- `import repo <filename>`: Import all morphologies from another repository. `-f`/`--overwrite`: Overwrite existing morphologies.
- `import swc <file> <name>`: Import an SWC morphology and store it under the given name.
- `arborize <class> <name>`: Import an Arborize model.
- `remove <name>`: Remove a morphology from the repository.
- `voxelize <name> [<n=130>]`: Generate a voxel cloud of `n` (optional, default=130) voxels for the morphology.
- `plot <name>`: Plot the morphology.
- `close`: Exit the mr state.

3.1.4 The HDF5 state

In this state you can view the structure of HDF5 files.

List of hdf5 commands:

- `view`: Create a hierarchical print of the HDF5 file, groups, datasets, and attributes.
- `plot`: Display a plot of the HDF5 network.

3.2 List of command line commands

Note: Parameters included between square brackets are optional, the brackets need not be included in the actual command.

3.2.1 compile

```
bsb [-v=1 -c=mouse_cerebellum] compile [-p -o]
```

Compiles a network architecture: Places cells in a simulated volume and connects them to eachother. All this information is then stored in a single HDF5 file.

- `-v`, `--verbosity`: Sets the verbosity of the scaffold. The higher the verbosity the more console output will be generated.
- `-c`, `--configuration`: Sets the configuration file that will be used.

- `-p`: Plot the created network.
- `-o=<file>`, `--output=<file>`: Output the result to a specific file.

3.2.2 simulate

`bsb [-v=1] simulate <name> [-rc=<config>] --hdf5=<file>`

Run a simulation from a compiled network architecture.

- `-v`, `--verbosity`: Sets the verbosity of the scaffold. The higher the verbosity the more console output will be generated.
- `-c`, `--configuration`: Sets the configuration file that will be used.
- `name`: Name of the simulation.
- `--hdf5`: Path to the compiled network architecture.
- `-rc`, `--reconfigure`: The path to a new configuration file for the HDF5 file.

3.2.3 run

`bsb [-v=1 -c=mouse_cerebellum] run <name> [-p]`

Run a simulation creating a new network architecture.

- `-v`, `--verbosity`: Sets the verbosity of the scaffold. The higher the verbosity the more console output will be generated.
- `-c`, `--configuration`: Sets the configuration file that will be used.
- `-p`: Plot the created network.

3.2.4 plot

`bsb plot <file>`

Create a plot of the network in an HDF5 file.

CHAPTER

FOUR

GUIDES

4.1 Layers

Layers are partitions of the simulation volume that most placement strategies use as a reference to place cells in.

4.1.1 Configuration

In the root node of the configuration file the `layers` dictionary configures all the layers. The key in the dictionary will become the layer name. A layer configuration requires only to describe its origin and dimensions. In its simplest form this can be achieved by providing a `position` and `thickness`. In that case the layer will scale along with the simulation volume `X` and `Z`.

Basic usage

Configure the following attributes:

- `position`: XYZ coordinates of the bottom-left corner, unless `xz_center` is set.
- `thickness`: Height of the layer

Example

```
{
  "layer": {
    "granular_layer": {
      "position": [0.0, 600.0, 0.0],
      "thickness": 150.0
    }
  }
}
```

Stacking layers

Placing layers manually can be sufficient, but when you have layers with dynamic sizes it can be usefull to automatically rearrange other layers. To do so you can group layers together in a vertical stack. To stack layers together you need to configure *stack* dictionaries in both with the same *stack_id* and different *position_in_stack*. Each stack requires exactly one definition of its *position*, which can be supplied in any of the layers it consists of:

```
"layers": {
  "layer_a": {
    "thickness": 150.0,
    "stack": {
      "stack_id": 0,
      "position_in_stack": 0,
      "position": [10, 0, 100]
    }
  },
  "layer_b": {
    "thickness": 150.0,
    "stack": {
      "stack_id": 0,
      "position_in_stack": 1
    }
  }
}
```

This will result in a stack of Layer A and B with Layer B on top. Both layers will have an X and Z origin of `10` and `100`, but the Y of Layer B will be raised from `0` with the thickness of Layer A, to `150`, ending up on top of it. Both Layer A and B will have X and Z dimensions equal to the simulation volume X and Z. This can be altered by specifying *xz_scale*.

Scaling layers

Layers by default scale with the simulation volume X and Z. You can change the default one-to-one ratio by specifying *xz_scale*:

```
"layer_a": {
  "xz_scale": 0.5
}
```

When the XZ size is `[100, 100]` layer A will be `[50, 50]` instead. You can also use a list to scale different on the X than on the Z axis:

```
"layer_a": {
  "xz_scale": [0.5, 2.0]
}
```

Volumetric scaling

Layers can also scale relative to the volume of other layers. To do so set a *volume_scale* ratio which will determine how many times larger the volume of this layer will be than its reference layers. The reference layers can be specified with *scale_from_layers*. The shape of the layer will be cubic, unless the *volume_dimension_ratio* is specified:

```
"some_layer": {
  "volume_scale": 10.0,
  "scale_from_layers": ["other_layer"],
  # Cube (default):
  "volume_dimension_ratio": [1., 1., 1.],
  # High pole:
  "volume_dimension_ratio": [1., 20., 1.], # Becomes [0.05, 1., 0.05]
  # Flat bed:
  "volume_dimension_ratio": [20., 1., 20.]
}
```

Note: The `volume_dimension_ratio` is normalized to the Y value.

4.1.2 Scripting

The value of layers in scripting is usually limited because they only contain spatial information.

Retrieving layers

Layers can be retrieved from a `ScaffoldConfig`:

```
from bsb.config import JSONConfig

config = JSONConfig("mouse_cerebellum")
layer = config.get_layer(name="granular_layer")
```

A `Scaffold` also stores its configuration:

```
layer = scaffold.configuration.get_layer(name="granular_layer")
```

All `Layered` placement strategies store a reference to their layer instance:

```
placement = scaffold.get_cell_type("granule_cell").placement
layer_name = placement.layer
layer = placement.layer_instance
```

Note: The instance of a placement strategy's layer is added only after initialisation of the placement strategy, which occurs only after the scaffold is bootstrapped (so after `scaffold = Scaffold(config)`)

4.2 Cell types

Cell types are the main component of the scaffold. They will be placed into the simulation volume and connected to eachother.

4.2.1 Configuration

In the root node of the configuration file the `cell_types` dictionary configures all the cell types. The key in the dictionary will become the cell type name. Each entry should contain a correct configuration for a `placement.PlacementStrategy` and *`morphologies.Morphology`* under the `placement` and `morphology` attributes respectively.

Optionally a `plotting` dictionary can be provided when the scaffold's plotting functions are used.

Basic usage

1. Configure the following attributes in `placement`:

- `class`: the importable name of the placement strategy class. 3 built-in implementations of the placement strategy are available: `ParticlePlacement`, `ParallelArrayPlacement` and `Satellite`
- `layer`: The topological layer in which this cell type appears.
- `soma_radius`: Radius of the cell soma in µm.
- `density`: Cell density, see *Cell count* for more possibilities.

2. Select one of the morphologies that suits your cell type and configure its required attributes. Inside of the morphology attribute, a `detailed_morphologies` attribute can be specified to select detailed morphologies from the morphology repository.

3. The cell type will now be placed whenever the scaffold is compiled, but you'll need to configure connection types to connect it to other cells.

Example

```
{
  "name": "My Test configuration",
  "output": {
    "format": "bsb.output.HDF5Formatter"
  },
  "network_architecture": {
    "simulation_volume_x": 400.0,
    "simulation_volume_z": 400.0
  },
  "layers": {
    "granular_layer": {
      "origin": [0.0, 0.0, 0.0],
      "thickness": 150
    }
  },
  "cell_types": {
    "granule_cell": {
```

(continues on next page)

(continued from previous page)

```
      "placement": {
        "class": "bsb.placement.ParticlePlacement",
        "layer": "granular_layer",
        "soma_radius": 2.5,
        "density": 3.9e-3
      },
      "morphology": {
        "class": "bsb.morphologies.GranuleCellGeometry",
        "pf_height": 180,
        "pf_height_sd": 20,
        "pf_length": 3000,
        "pf_radius": 0.5,
        "dendrite_length": 40,
        "detailed_morphologies": ["GranuleCell"]
      },
      "plotting": {
        "display_name": "granule cell",
        "color": "#E62214"
      }
    }
  },
  "connection_types": {},
  "simulations": {}
}
```

Use `bsb -c=my-config.json compile` to test your configuration file.

4.3 Connection types

Connection types connect cell types together after they've been placed into the simulation volume. They are defined in the configuration under `connection_types`:

```
{
  "connection_types": {
    "cell_A_to_cell_B": {
      "class": "bsb.connectivity.VoxelIntersection",
      "from_cell_types": [
        {
          "type": "cell_A",
          "compartments": ["axon"]
        }
      ],
      "to_cell_types": [
        {
          "type": "cell_B",
          "compartments": ["dendrites", "soma"]
        }
      ]
    }
  }
}
```

The *class* specifies which `ConnectionStrategy` to load for this conenction type. The *from_cell_types* and *to_cell_types* specify which pre- and postsynaptic cell types to use respectively. The cell type definitions in those lists have to contain a *type* that links to an existing cell type and can optionally contain hints to which *compartments* of the morphology to use.

4.3.1 Creating your own

In order to create your own connection type, create an importable module (refer to the Python documentation) with inside a class inheriting from *connectivity.ConnectionStrategy*. Let's start by deconstructing a full code example that connects cells that are near each other between a `min` and `max` distance:

```
from bsb.connectivity import ConnectionStrategy
from bsb.exceptions import ConfigurationError
import scipy.spatial.distance as dist

class ConnectBetween(ConnectionStrategy):
  # Casts given configuration values to a certain type
  casts = {
    "min": float,
    "max": float,
  }
  # Default values for the configuration attributes
  defaults = {
    "min": 0.,
  }
  # Configuration attributes that the user must give or an error is thrown.
  required = ["max"]

  # The function to check whether the given values are all correct
  def validate(self):
    if self.max < self.min:
      raise ConfigurationError("Max distance should be larger than min distance.")

  # The function to determine which cell pairs should be connected
  def connect(self):
    for ft in self.from_cell_types:
      ps_from = self.scaffold.get_placement_set(ft)
      fpos = ps_from.positions
      for tt in self.to_cell_types:
        ps_to = self.scaffold.get_placement_set(tt)
        tpos = ps_to.positions
        pairw_dist = dist.cdist(fpos, tpos)
        pairs = ((pairw_dist <= max) & (pairw_dist >= min)).nonzero()
        # More code to convert `pairs` into a Nx2 matrix of pre & post synaptic pair IDs
        # ...
        self.scaffold.connect_cells(f"{ft.name}_to_{tt.name}", pairs)
```

An example using this strategy, assuming it is importable as the `my_module` module:

```
{
  "connection_types": {
    "cell_A_to_cell_B": {
      "class": "my_module.ConnectBetween",
```

(continues on next page)

(continued from previous page)

```
      "min": 10,
      "max": 15.5,
      "from_cell_types": [
        {
          "type": "cell_A"
        }
      ],
      "to_cell_types": [
        {
          "type": "cell_B"
        }
      ]
    }
  }
}
```

Configuration attributes

All keys present on the connection type in the configuration will be available on the connection strategy under `self.<key>` (e.g. *min* will become `self.min`). Additionally the scaffold object is available under `self.scaffold`.

Configuration attributes will by default have the data type they have in JSON, which can be any of `int`, `float`, `str`, `list` or `dict`. This data type can be overridden by using the class attribute `casts`. Any key present in this dictionary will use the value as a conversion function if the configuration attribute is encountered.

In this example both *min* and *max* will be converted to `float`. You can also provide your own functions or lambdas as long as they take the configuration value as only argument:

```
casts = {"cake_or_pie": lambda x: "pie" if x < 10 else "cake"}
```

You can provide default values for configuration attributes giving the `defaults` class variable dictionary. You can also specify that certain attributes are `required` to be provided. If they occur in the `defaults` dictionary the default value will be used when no value is provided in the configuration.

Validation handling

The given configuration attributes can be further validated using the `validate` method. From inside the `validate` method a `ConfigurationError` can be thrown when the user given values aren't valid. This method is required, if no validation is required a noop function should be given:

```
def validate(self):
  pass
```

Connection handling

Inside of the `connect` function the from and to cell types will be available. You can access their placement data using `self.scaffold.get_placement_set(type)`. The properties of a `PlacementSet` are expensive IO operations, cache them:

```
# WRONG! Will read the data from file 200 times
for i in range(100):
  ps1.positions - ps2.positions

# Correct! Will read the data from file only 2 times
pos1 = ps1.positions
pos2 = ps2.Positions
for i in range(100):
  pos1 - pos2
```

Finally you should call `self.scaffold.connect_cells(tag, matrix)` to connect the cells. The tag is free to choose, the matrix should be rows of pre to post cell ID pairs.

4.3.2 Connection types and labels

When defining a connection type under `connection_types` in the configuration file, it is possible to select specific subpopulations inside the attributes `from_cell_types` and/or `to_cell_types`. By including the attribute `with_label` in the `connection_types` configuration, you can define the subpopulation label:

```
{
  "connection_types": {
    "cell_A_to_cell_B": {
      "class": "my_module.ConnectBetween",
      "from_cell_types": [
        {
          "type": "cell_A",
          "with_label": "cell_A_type_1"
        }
      ],
      "to_cell_types": [
        {
          "type": "cell_B",
          "with_label": "cell_B_type_3"
        }
      ]
    }
  }
}
```

Note: The labels used in the configuration file must correspond to the labels assigned during cell placement.

Using more than one label

If under `connection_types` more than one label has been specified, it is possible to choose whether the labels must be used serially or in a mixed way, by including a new attribute `mix_labels`. For instance:

```
{
  "connection_types": {
    "cell_A_to_cell_B": {
      "class": "my_module.ConnectBetween",
      "from_cell_types": [
        {
          "type": "cell_A","with_label": ["cell_A_type_2","cell_A_type_1"]
        }
      ],
      "to_cell_types": [
        {
          "type": "cell_B","with_label": ["cell_B_type_3","cell_B_type_2"]
        }
      ]
    }
  }
}
```

Using the above configuration file, the established connections are:

- From `cell_A_type_2` to `cell_B_type_3`
- From `cell_A_type_1` to `cell_B_type_2`

Here there is another example of configuration setting:

```
{
  "connection_types": {
    "cell_A_to_cell_B": {
      "class": "my_module.ConnectBetween",
      "from_cell_types": [
        {
          "type": "cell_A","with_label": ["cell_A_type_2","cell_A_type_1"]
        }
      ],
      "to_cell_types": [
        {
          "type": "cell_B","with_label": ["cell_B_type_3","cell_B_type_2"]
        }
      ],
      "mix_labels": true,
    }
  }
}
```

In this case, thanks to the `mix_labels` attribute,the established connections are:

- From `cell_A_type_2` to `cell_B_type_3`
- From `cell_A_type_2` to `cell_B_type_2`
- From `cell_A_type_1` to `cell_B_type_3`

- From `cell_A_type_1` to `cell_B_type_2`

4.4 Output

4.5 Output Formats

4.5.1 Nearly-continuous list

This format is used to store lists that are almost always just a sequence of continuous numbers. It will always contain pairs that describe a continuous chain of numbers as a start and length.

For example this sequence:

```
[15, 3, 30, 4]
```

Describes 3 numbers starting from 15 and 4 numbers starting from 30:

```
[15, 16, 17, 30, 31, 32, 33]
```

See *helpers.continuity_list()* for the implementation.

Note: The scaffold generates continuous IDs, but this assumption does not hold true in many edge cases like manually placing cells, using custom placement strategies or after postprocessing the placed cells.

4.6 Simulations

After building the scaffold models, simulations can be run using NEST or NEURON.

Simulations can be configured in the `simulations` dictionary of the root node of the configuration file, specifying each simulation with its name, e.g. “first_simulation”, “second_simulation”:

```
{
  "simulations": {
    "first_simulation": {

    },
    "second_simulation": {

    }
  }
}
```

4.6.1 NEST

NEST is mainly used for simulations of Spiking Neural Networks, with point neuron models.

4.6.2 Configuration

NEST simulations in the scaffold can be configured setting the attribute `simulator` to `nest`. The basic NEST simulation properties can be set through the attributes:

- `default_neuron_model`: default model used for all `cell_models`, unless differently indicated in the `neuron_model` attribute of a specific cell model.
- `default_synapse_model`: default model used for all `connection_models` (e.g. `static_synapse`), unless differently indicated in the `synapse_model` attribute of a specific connection model.
- `duration`: simulation duration in [ms].
- `modules`: list of NEST extension modules to be installed.

Then, the dictionaries `cell_models`, `connection_models`, `devices`, `entities` specify the properties of each element of the simulation.

```
{
  "simulations": {
    "first_simulation": {
      "simulator": "nest",
      "default_neuron_model": "iaf_cond_alpha",
      "default_synapse_model": "static_synapse",
      "duration": 1000,
      "modules": ["cerebmodule"],

      "cell_models": {

      },
      "connection_models": {

      },
      "devices": {

      },
      "entities": {

      }

    },
    "second_simulation": {

    }
  }
}
```

Cells

In the `cell_models` attribute, it is possible to specify simulation-specific properties for each cell type:

- `cell_model`: NEST neuron model, if not using the `default_neuron_model`. Currently supported models are `iaf_cond_alpha` and `eglif_cond_alpha_multisyn`. Other available models can be found in the NEST documentation
- `parameters`: neuron model parameters that are common to the NEST neuron models that could be used, including:
  - `t_ref`: refractory period duration [ms]
  - `C_m`: membrane capacitance [pF]
  - `V_th`: threshold potential [mV]
  - `V_reset`: reset potential [mV]
  - `E_L`: leakage potential [mV]

Then, neuron model specific parameters can be indicated in the attributes corresponding to the model names:

- `iaf_cond_alpha`:
  - `I_e`: endogenous current [pA]
  - `tau_syn_ex`: time constant of excitatory synaptic inputs [ms]
  - `tau_syn_in`: time constant of inhibitory synaptic inputs [ms]
  - `g_L`: leaky conductance [nS]
- `eglif_cond_alpha_multisyn`:
  - `Vmin`: minimum membrane potential [mV]
  - `Vinit`: initial membrane potential [mV]
  - `lambda_0`: escape rate parameter
  - `tau_V`: escape rate parameter
  - `tau_m`: membrane time constant [ms]
  - `I_e`: endogenous current [pA]
  - `kadap`: adaptive current coupling constant
  - `k1`: spike-triggered current decay
  - `k2`: adaptive current decay
  - `A1`: spike-triggered current update [pA]
  - `A2`: adaptive current update [pA]
  - `tau_syn1`, `tau_syn2`, `tau_syn3`: time constants of synaptic inputs at the 3 receptors [ms]
  - `E_rev1`, `E_rev2`, `E_rev3`: reversal potential for the 3 synaptic receptors (usually set to 0mV for excitatory and -80mV for inhibitory synapses) [mV]
  - `receptors`: dictionary specifying the receptor number for each input cell to the current neuron

Example

Configuration example for a cerebellar Golgi cell. In the `eglif_cond_alpha_multisyn` neuron model, the 3 receptors are associated to synapses from glomeruli, Golgi cells and Granule cells, respectively.

```
{
  "cell_models": {
    "golgi_cell": {
      "parameters": {
        "t_ref": 2.0,
        "C_m": 145.0,
        "V_th": -55.0,
        "V_reset": -75.0,
        "E_L": -62.0
      },
      "iaf_cond_alpha": {
        "I_e": 36.75,
        "tau_syn_ex": 0.23,
        "tau_syn_in": 10.0,
        "g_L": 3.3
      },
      "eglif_cond_alpha_multisyn": {
        "Vmin": -150.0,
        "Vinit": -62.0,
        "lambda_0": 1.0,
        "tau_V":0.4,
        "tau_m": 44.0,
        "I_e": 16.214,
        "kadap": 0.217,
        "k1": 0.031,
        "k2": 0.023,
        "A1": 259.988,
        "A2":178.01,
        "tau_syn1":0.23,
        "tau_syn2": 10.0,
        "tau_syn3": 0.5,
        "E_rev1": 0.0,
        "E_rev2": -80.0,
        "E_rev3": 0.0,
        "receptors": {
          "glomerulus": 1,
          "golgi_cell": 2,
          "granule_cell": 3
        }
      }
    }
  }
}
```

Connections

Simulations with plasticity

The default synapse model for connection models is usually set to `static_synapse`.

For plastic synapses, it is possible to choose between:

1. homosynaptic plasticity models (e.g. `stdp_synapse`) where weight changes depend on pre- and postsynaptic spike times
2. heterosynaptic plasticity models (e.g. `stdp_synapse_sinexp`), where spikes of an external teaching population trigger the weight change. In this case, a device called "volume transmitter" is created for each postsynaptic neuron, collecting the spikes from the teaching neurons.

For a full set of available synapse models, see the NEST documentation

For the plastic connections, specify the attributes as follows:

- `plastic`: set to `true`.
- `hetero`: set to `true` if using an heterosynaptic plasticity model.
- `teaching`: Connection model name of the teaching connection for heterosynaptic plasticity models.
- `synapse_model`: the name of the NEST synapse model to be used. By default, it is the model specified in the `default_synapse_model` attribute of the current simulation.
- `synapse`: specify the parameters for each one of the synapse models that could be used for that connection.

Note: If the `synapse_model` attribute is not specified, the `default_synapse_model` will be used (`static`). Using synapse models without plasticity - such as `static` - while setting the `plastic` attribute to `true` will lead to errors.

Example

```
{
  "connection_models": {
    "parallel_fiber_to_purkinje": {
      "plastic": true,
      "hetero": true,
      "teaching": "io_to_purkinje",
      "synapse_model": "stdp_synapse_sinexp",
      "connection": {
        "weight": 0.007,
        "delay": 5.0
      },
      "synapse": {
        "static_synapse": {},
        "stdp_synapse_sinexp": {
          "A_minus": 0.5,
          "A_plus": 0.05,
          "Wmin": 0.0,
          "Wmax": 100.0
        }
      }
```

(continues on next page)

(continued from previous page)

```
    },

    "purkinje_to_dcn": {
      "plastic": true,
      "synapse_model": "stdp_synapse",
      "connection": {
        "weight":-0.4,
        "delay": 4.0
      },
      "synapse": {
        "static_synapse": {},
        "stdp_synapse": {
          "tau_plus":30.0,
          "alpha": 0.5,
          "lambda": 0.1,
          "mu_plus": 0.0,
          "mu_minus": 0.0,
          "Wmax": 100.0
        }
      }
    }
  }
}
```

Devices

Entities

4.7 List of placement strategies

4.7.1 PlacementStrategy

Configuration

- `layer`: The layer in which to place the cells.
- `soma_radius`: The radius in µm of the cell body.
- `count`: Determines cell count absolutely.
- `density`: Determines cell count by multiplying it by the placement volume.
- `planar_density`: Determines cell count by multiplying it by the placement surface.
- `placement_relative_to`: The cell type to relate this placement count to.
- `density_ratio`: A ratio that can be specified along with `placement_relative_to` to multiply another cell type's density with.
- `placement_count_ratio`: A ratio that can be specified along with `placement_relative_to` to multiply another cell type's placement count with.

4.7.2 ParallelArrayPlacement

Class: `placement.ParallelArrayPlacement`

4.7.3 FixedPositions

Class: `placement.FixedPositions`

This class places the cells in fixed positions specified in the attribute `positions`.

Configuration

- `positions`: a list of 3D points where the neurons should be placed. For example:

```
{
  "cell_types": {
    "golgi_cell": {
      "placement": {
        "class": "bsb.placement.FixedPositions",
        "layer": "granular_layer",
        "count": 1,
        "positions": [[40.0,0.0,-50.0]]
        }
    },
  }
}
```

4.8 List of connection strategies

Connection strategies starting whose name start with `Connectome` are made for a specific connection between 2 cell types, those that do not can be used for connections between any cell type.

4.8.1 Shared configuration attributes

- `class`: A string that specifies which connection strategy to apply to the connection type.
- `from_cell_types`: An array of objects with a `type` key indicating presynaptic cell types and optionally a `compartments` key for an array of compartment types:

```
"from_cell_types": [
  {"type": "basket_cell", "compartments": ["axon"]},
  {"type": "stellate_cell", "compartments": ["axon"]}
]
```

- `to_cell_types`: Same as `from_cell_types` but for the postsynaptic cell type.

VoxelIntersection

This strategy voxelizes morphologies into collections of cubes, thereby reducing the spatial specificity of the provided traced morphologies by grouping multiple compartments into larger cubic voxels. Intersections are found not between the seperate compartments but between the voxels and random compartments of matching voxels are connected to eachother. This means that the connections that are made are less specific to the exact morphology and can be very useful when only 1 or a few morphologies are available to represent each cell type.

- `affinity`: A fraction between 1 and 0 which indicates the tendency of cells to form connections with other cells with whom their voxels intersect. This can be used to downregulate the amount of cells that any cell connects with.
- `contacts`: A number or distribution determining the amount of synaptic contacts one cell will form on another after they have selected eachother as connection partners.

Note: The affinity only affects the number of cells that are contacted, not the number of synaptic contacts formed with each cell.

FiberIntersection

This strategy is a special case of *VoxelIntersection* that can be applied to morphologies with long straight compartments that would yield incorrect results when approximated with cubic voxels like in VoxelIntersection (e.g. Ascending Axons or Parallel Fibers in Granule Cells). The fiber, organized into hierarchical branches, is split into segments, based on original compartments length and configured resolution. Then, each branch is voxelized into parallelepipeds: each one is built as the minimal volume with sides parallel to the main reference frame axes, surrounding each segment. Intersections with postsynaptic voxelized morphologies are then obtained applying the same method as in *VoxelIntersection*.

- `resolution`: the maximum length [um] of a fiber segment to be used in the fiber voxelization. If the resolution is lower than a compartment length, the compartment is interpolated into smaller segments, to achieve the desired resolution. This property impacts on voxelization of fibers not parallel to the main reference frame axes. Default value is 20.0 um, i.e. the length of each compartment in Granule cell Parallel fibers.
- `affinity`: A fraction between 1 and 0 which indicates the tendency of cells to form connections with other cells with whom their voxels intersect. This can be used to downregulate the amount of cells that any cell connects with. Default value is 1.
- `to_plot`: a list of cell fiber numbers (e.g. 0 for the first cell of the presynaptic type) that will be plotted during connection creation using *plot_fiber_morphology*.
- `transform`: A set of attributes defining the transformation class for fibers that should be rotated or bended. Specifically, the *QuiverTransform* allows to bend fiber segments based on a vector field in a voxelized volume. The attributes to be set are:
  - `quivers`: the vector field array, of shape e.g. `(3, 500, 400, 200)` for a volume with 500, 400 and 200 voxels in x, y and z directions, respectively.
  - `vol_res`: the size [um] of voxels in the volume where the quiver field is defined. Default value is 25.0, i.e. the voxel size in the Allen Brain Atlas.
  - `vol_start`: the origin of the quiver field volume in the reconstructed volume reference frame.
  - `shared`: if the same transformation should be applied to all fibers or not

TouchingConvergenceDivergence

- `divergence`: Preferred amount of connections starting from 1 from_cell
- `convergence`: Preferred amount of connections ending on 1 to_cell

ConnectomeGlomerulusGranule

Inherits from TouchingConvergenceDivergence. No additional configuration. Uses the dendrite length configured in the granule cell morphology.

ConnectomeGlomerulusGolgi

Inherits from TouchingConvergenceDivergence. No additional configuration. Uses the dendrite radius configured in the Golgi cell morphology.

ConnectomeGolgiGlomerulus

Inherits from TouchingConvergenceDivergence. No additional configuration. Uses the `axon_x`, `axon_y`, `axon_z` from the Golgi cell morphology to intersect a parallelopipid Golgi axonal region with the glomeruli.

ConnectomeGranuleGolgi

Creates 2 connectivity sets by default *ascending_axon_to_golgi* and *parallel_fiber_to_golgi* but these can be overwritten by providing `tag_aa` and/or `tag_pf` respectively.

Calculates the distance in the XZ plane between granule cells and Golgi cells and uses the Golgi cell morphology's dendrite radius to decide on the intersection.

Also creates an ascending axon height for each granule cell.

- `aa_convergence`: Preferred amount of ascending axon synapses on 1 Golgi cell.
- `pf_convergence`: Preferred amount of parallel fiber synapses on 1 Golgi cell.

ConnectomeGolgiGranule

No configuration, it connects each Golgi to each granule cell that it shares a connected glomerules with.

ConnectomeAscAxonPurkinje

Intersects the rectangular extension of the Purkinje dendritic tree with the granule cells in the XZ plane, uses the Purkinje cell's placement attributes `extension_x` and `extension_z`.

- `extension_x`: Extension of the dendritic tree in the X plane
- `extension_z`: Extension of the dendritic tree in the Z plane

ConnectomePFPurkinje

No configuration. Uses the Purkinje cell's placement attribute `extension_x`. Intersects Purkinje cell dendritic tree extension along the x axis with the x position of the granule cells, as the length of a parallel fiber far exceeds the simulation volume.

4.9 Placement sets

PlacementSets are constructed from the *Output* and can be used to retrieve lists of identifiers, positions, rotations and additional datasets. It can also be used to construct a list of `Cells` that combines that information into objects.

Note: Loading these datasets from storage is an expensive operation. Store a local reference to the data you retrieve:

```
data = placement_set.identifiers # Store a local variable
cell0 = data[0] # NOT: placement_set.identifiers[0]
cell1 = data[1] # NOT: placement_set.identifiers[1]
```

4.9.1 Retrieving a PlacementSet

The output formatter of the scaffold is responsible for retrieving the dataset from the output storage. The scaffold itself has a method `get_placement_set` that takes a name of a cell type as input which will defer to the output formatter and returns a PlacementSet. If the placement set does not exist, an `DatesetNotFoundError` is thrown.

```
ps = scaffold.get_placement_set("granule_cell")
```

4.9.2 Identifiers

The identifiers of the cells of a cell type can be retrieved using the `identifiers` property. Identifiers are stored in a *Nearly-continuous list*.

```
for n, cell_id in enumerate(ps.identifiers):
  print("I am", ps.tag, "number", n, "with ID", cell_id)
```

4.9.3 Positions

The positions of the cells can be retrieved using the `positions` property. This dataset is not present on entity types:

```
for n, cell_id, position in zip(range(len(ps)), ps.identifiers, ps.positions):
  print("I am", ps.tag, "number", n, "with ID", cell_id)
  print("My position is", position)
```

4.9.4 Rotations

Some placement strategies or external data sources might also provide rotational information for each cell. The `rotations` property works analogous to the `positions` property.

4.9.5 Additional datasets

Not implemented yet.

4.10 Plotting Tools

The scaffold package provides tools to plot network topology (either point and detailed networks) and morphologies in the `bsb.plotting` module.

To plot a network saved in a *bsb* instance, you can use:

- `plot_network_cache(scaffold)`: to plot the network saved in the memory cache after having compiled it
- `plot_network(scaffold)`: to plot a network, adding the keyword argument `from_memory=False` if you want to plot a network saved in a previously compiled HDF5 file. The default value is `from_memory=True`, which plots the version saved in your cache (you should have compiled the network in the current session).
- `plot_network_detailed(scaffold)`: Plots cells represented by their fully detailed morphologies. These plots are usually not able to render more than a 30-50 cells at the same time depending on the complexity of their morphology.

You can also plot morphologies:

- `plot_morphology(m)`: Plots a *Morphology*
- `plot_fiber_morphology(fm)`: Plots a `FiberMorphology`
- `plot_voxel_cloud(m.cloud)`: Plots a `VoxelCloud`

All of the above functions take a `fig` keyword argument of type `plotly.graph_objects.Figure` in case you want to modify the figure, or combine multiple plotting functions on the same figure, such as plotting a morphology and the voxel cloud of its axon.

4.11 Blender

The BSB features a blender module capable of creating the network inside of Blender and animating the network activity. On top of that it completely prepares the scene including camera and lighting and contains rendering and sequencing pipelines so that videos of the network can be produced from start to finish with the BSB framework.

This guide assumes familiarity with Blender but can probably be succesfully reproduced by a panicking PhD student with a deadline tomorrow aswell.

4.11.1 Blender mixin module

To use a `network` in the Blender context invoke the blender mixins using the `for_blender()` function. This will load all the blender functions onto the network object:

```
import bpy, bsb.core

network = bsb.core.from_hdf5("mynetwork.hdf5")
network.for_blender()
# `network` now holds a reference to each BSB blender mixin/blendin function
```

4.11.2 Blending

Some of the functions in the blender module set the scene state state-independently. This means that whatever state your blender scene used to be in before calling the function, afterwards some aspect of the scene state will always be the same. The function calls … blend in. A concrete example would be the `network.load_population` function: If the current scene does not contain the population being loaded it will be created, anywhere in the script after the function call you can safely assume the population exists in the scene. Since the function does nothing if the population exists you can put it anywhere.

These blending functions are useful because you're likely to want to change some colors or sizes or positions of large amounts of objects and the easiest way to do that is by changing the declarative value and repeating your script. This would not be possible if the `load_population` function were to always recreate the population each time the script was called.

The primary blending function is the `network.blend(name, scene)` function that blends your network into the scene under the given name, blending in a root collection, cells collection, camera and light for it. If there's nothing peculiar about any of the cell types in your network fire up the `load_populations` blendin and your network will pop up in the scene. From here on out you are either free to do with the blender objects what you want or you can continue to use some of the BSB blendins:

```
import bpy, bsb.core, h5py, itertools

network = bsb.core.from_hdf5("mynetwork.hdf5")
# Blend the network into the current scene under the name `scaffold`
network.for_blender().blend(bpy.context.scene, "scaffold")
# Load all cell types into the blender scene
populations = network.get_populations()
cells = itertools.chain(*(p.cells for p in populations.values()))
# Use the 'pulsar' animation to animate all cells with the simulation results
with h5py.File("my_results.hdf5", "r") as f:
  # Animate the simulation's spikes
  network.animate.pulsar(f["recorders/soma_spikes"], cells)
```

Note: While `load_populations` simply checks the existence, `get_populations` returns a BlenderPopulation object that holds references to each cell, and its Blender object. Some work goes into looking up the blender object for each cell so if you don't use the cells in every run of the script it might be better to open up with a `load_populations` and call `get_population(name)` later when you need a specific population.

Warning: It's easy to overload Blender with cell objects. It becomes quite difficult to use Blender around 20,000 cells. If you have significantly more cells be sure to save unpopulated versions of your Blender files, run the blendin

script, save as another file, render it and make the required changes to the unpopulated version, repeating the process. Optimizations are likely to be added in the future.

4.11.3 Blender HPC workflow

The `devops/blender-pipe` folder contains scripts to facilitate the rendering and sequencing of BSB blendfiles on HPC systems. Copy them together to a directory on the HPC system and make sure that the `blender` command opens Blender. The pipeline contains 2 steps, `rendering` each frame in parallel and `sequencing` the rendered images into a video.

jrender.slurm

The render jobscript uses `render.py` to invoke Blender. Each Blender process will be tasked with rendering a certain proportion of the frames. `jrender.slurm` takes 2 arguments, the blendfile and the output image folder:

```
sbatch jrender.slurm my_file.blend my_file_imgs
```

jsequence.slurm

The sequencing jobscript stitches together the rendered frames into a video. This has to be done in serial on a single node. It takes the blendfile and image folder as arguments:

```
sbatch jsequence.slurm my_file.blend my_file_imgs
```

CHAPTER

FIVE

CELL PLACEMENT

Cell placement is handled by the *placement module*. This module will place the cell types in the layers based on a certain *Placement Strategy*.

Placement occurs as the first step during network architecture compilation.

The placement order starts from cell type with the lowest cell count first unless specified otherwise in the cell type's placement configuration.

See the *List of placement strategies*

Contents

5.1 Configuration

5.1.1 Cell count

Specifying cell count can be done with `count`, `density` (µm^-3), `planar_density` (µm^-2) or a ratio to another cell with `placement_relative_to` (other cell type) and either `density_ratio` to place with their density multiplied by the given ratio or `placement_count_ratio` to place with their count multiplied by the given ratio

5.1.2 Placement order

By default the cell types are placed sorted from least to most cells per type. This default order can be influenced by specifying an `after` attribute in the cell type's placement configuration. This is an array of cell type names which need to be placed before this cell type:

```
{
  "cell_types": {
    "later_cell_type": {
      "...": "...",
      "after": ["first_cell_type"]
    },
    "first_cell_type": { "...": "..." },
  }
}
```

5.2 Placement Strategy

Each cell type has to specify a placement strategy that determines the algorithm used to place cells. The placement strategy is an interface whose `place` method is called when placement occurs.

5.2.1 Placing cells

Call the scaffold instance's `core.Scaffold.place_cells()` function to place cells in the simulation volume.

5.3 Labels

CHAPTER

SIX

MORPHOLOGIES

Morphologies are the 3D representation of a cell. In the BSB they consist of branches, pieces of cable described as vectors of the properties of points. Consider the following branch with 4 points `p0, p1, p2, p3`:

```
branch0 = [x, y, z, r]
x = [x0, x1, x2, x3]
y = [y0, y1, y2, y3]
z = [z0, z1, z2, z3]
r = [r0, r1, r2, r3]
```

The points on the branch can also be described as individual `Compartments`:

```
branch0 = [c0, c1, c2]
c0 = Comp(start=[x0, y0, z0], end=[x1, y1, z1], radius=r1)
c1 = Comp(start=[x1, y1, z1], end=[x2, y2, z2], radius=r2)
c2 = Comp(start=[x2, y2, z2], end=[x3, y3, z3], radius=r3)
```

Branches also specify which other branches they are connected to and in this way the entire network of neuronal processes can be described. Those branches that do not have a parent branch are called `roots`. A morphology can have as many roots as it likes; usually in the case of 1 root it represents the soma; in the case of many roots they each represent the start of a process such as an axon on dendrite around an imaginary soma.

In the end a morphology can be summed up in pseudo-code as:

```
m = Morphology(roots)
m.roots = <all roots>
m.branches = <all branches, depth first starting from the roots>
```

The `branches` attribute is the result of a depth-first iteration of the roots list. Any kind of iteration over roots or branches will always follow this same depth-first order.

The data of these morphologies are stored in `MorphologyRepositories` as groups of branches following the first vector-based branch description. If you want to use `compartments` you'll have to call `branch.to_compartments()` or `morphology.to_compartments()`. For a root branch this will yield `n - 1` compartments formed as line segments between pairs of points on the branch. For non-root branches an extra compartment is prepended between the last point of the parent branch and the first point of the child branch. Compartments are individuals so branches are no longer used to describe the network of points, instead each compartment lists their own parent compartment.

6.1 Using morphologies

For this introduction we're going to assume that you have a `MorphologyRepository` with morphologies already present in them. To learn how to create your own morphologies stored in `MorphologyRepositories` see morphologies/repository.

Let's start with loading a morphology and inspecting its root *Branch*:

```
from bsb.core import from_hdf5
from bsb.output import MorphologyRepository

mr = MorphologyRepository("path/to/mr.hdf5")
# Alternatively if you have your MR inside of a compiled network:
network = from_hdf5("network.hdf5")
mr = network.morphology_repository
morfo = mr.get_morphology("my_morphology")

# Use a local reference to the properties if you're not going to manipulate the
# morphology, as they require a full search of the morphology to be determined every
# time the property is accessed.
roots = morfo.roots
branches = morfo.branches
print("Loaded a morphology with", len(roots), "roots, and", len(branches), "branches")
# In most morphologies there will be a single root, representing the soma.
soma_branch = roots[0]

# Use the vectors of the branch (this is the most performant option)
print("A branch can be represented by the following vectors:")
print("x:", soma_branch.x)
print("y:", soma_branch.y)
print("z:", soma_branch.z)
print("r:", soma_branch.radii)
# Use the points property to retrieve a matrix notation of the branch
# (Stacks the vectors into a 2d matrix)
print("The soma can also be represented by the following matrix:", soma_branch.points)

# There's also an iterator to walk over the points in the vectors
print("The soma is defined as the following points:")
for point in soma_branch.walk():
  print("*", point)
```

As you can see an individual branch contains all the positional data of the individual points in the morphology. The morphology object itself then contains the collection of branches. Normally you'd use the `.branches` but if you want to work with the positional data of the whole morphology in a object you can do this by flattening the morphology:

```
from bsb.core import from_hdf5

network = from_hdf5("network.hdf5")
mr = network.morphology_repository
morfo = mr.get_morphology("my_morphology")

print("All the branches in depth-first order:", morfo.branches)
print("All the points on those branches in depth first order:")
```

(continues on next page)

(continued from previous page)

```
print("- As vectors:", morfo.flatten())
print("- As matrix:", morfo.flatten(matrix=True).shape)
```

CHAPTER

SEVEN

CELL CONNECTIVITY

Cell connections are made as the second step of compilation. Each connection type configures one *connectivity.ConnectionStrategy* and can override the `connect` method to connect cells to eachother. Use the scaffold instance's :func:.core.Scaffold.connect_cells` to connect cells to eachother.

See the *List of connection strategies*.

7.1 Configuration

Each ConnectionStrategy is a ConfigurableClass, meaning that the attributes from the configuration files will be copied and validated onto the connection object.

7.2 Connecting cells

The connection matrices use a 2 column, 2 dimensional ndarray where the columns are the from and to id respectively. For morphologically detailed connections additional identifiers can be passed into the function to denote the specific compartments and morphologies that were used.

CHAPTER

EIGHT

SIMULATING NETWORKS WITH THE BSB

The BSB manages simulations by deferring as soon as possible to the simulation backends. Each simulator has good reasons to make their design choices, fitting to their simulation paradigm. These choices lead to divergence in how simulations are described, and each simulator has their own niche functions. This means that if you are already familiar with a simulator, writing simulation config should feel familiar, on top of that the BSB is able to offer you access to each simulator's full set of features. The downside is that you're required to write a separate simulation config block per backend.

Now, let's get started.

8.1 Conceptual overview

Each simulation config block needs to specify which *simulator* they use. Valid values are `arbor`, `nest` or `neuron`. Also included in the top level block are the *duration*, *resolution* and *temperature* attributes:

```
{
  "simulations": {
    "my_arbor_sim": {
      "simulator": "arbor",
      "duration": 2000,
      "resolution": 0.025,
      "temperature": 32,
      "cell_models": {

      },
      "connection_models": {

      },
      "devices": {

      }
    }
  }
}
```

The *cell_models* are the simulator specific representations of the network's *cell types*, the *connection_models* of the network's *connectivity types* and the *devices* define the experimental setup (such as input stimuli and recorders). All of the above is simulation backend specific and are covered in detail below.

8.2 Arbor

8.2.1 Cell models

The keys given in the *cell_models* should correspond to a `cell type` in the network. If a certain `cell type` does not have a corresponding `cell model` then no cells of that type will be instantiated in the network. Cell models in Arbor should refer to importable `arborize` cell models. The Arborize model's `.cable_cell` factory will be called to produce cell instances of the model:

```
{
  "cell_models": {
    "cell_type_A": {
      "model": "my.models.ModelA"
    },
    "afferent_to_A": {
      "relay": true
    }
  }
}
```

Note: *Relays* will be represented as `spike_source_cells` which can, through the connectome relay signals of other relays or devices. `spike_source_cells` cannot be the target of connections in Arbor, and the framework targets the targets of a relay instead, until only `cable_cells` are targeted.

8.2.2 Connection models

todo: doc

```
{
  "connection_models": {
    "aff_to_A": {
      "weight": 0.1,
      "delay": 0.1
    }
  }
}
```

8.2.3 Devices

`spike_generator` and `probes`:

```
{
  "devices": {
    "input_stimulus": {
      "device": "spike_generator",
      "explicit_schedule": {
        "times": [1,2,3]
      },
      "targetting": "cell_type",
```

(continues on next page)

(continued from previous page)

```
      "cell_types": ["mossy_fibers"]
    },
    "all_cell_recorder": {
      "targetting": "representatives",
      "device": "probe",
      "probe_type": "membrane_voltage",
      "where": "(uniform (all) 0 9 0)"
    }
  }
}
```

todo: doc & link to targetting

8.3 NEST

8.4 NEURON

CHAPTER

NINE

INDICES AND TABLES

9.1 Configuration reference

Note: The key of a configuration object in its parent will be stored as its *name* property and is used throughout the package. Some of these values are hardcoded into the package and the names of the standard configuration objects should not be changed.

9.1.1 Root attributes

The root node accepts the following attributes:

- *name*: *Unused*, a name for the configuration file. Is stored in the output files so it can be used for reference.
- *output*: Configuration object for the output *output.HDF5Formatter*.
- *network_architecture*: Configuration object for general simulation properties.
- *layers*: A dictionary containing the *models.Layer* configurations.
- *cell_types*: A dictionary containing the *models.CellType* configurations.
- *connection_types*: A dictionary containing the *connectivity.ConnectionStrategy* configurations.
- *simulations*: A dictionary containing the `simulation.SimulationAdapter` configurations.

```
{
  "name": "...",
  "output": {

  },
  "network_architecture": {

  },
  "layers": {
    "some_layer": {

    },
    "another_layer": {

    }
  },
```

(continues on next page)

(continued from previous page)

```
  "cell_types": {

  },
  "connection_types": {

  },
  "simulations": {

  }
}
```

9.1.2 Output attributes

Format

This attribute is a string that refers to the implementation of the OutputFormatter that should be used:

```
{
  "output": {
    "format": "bsb.output.HDF5Formatter"
  }
}
```

If you write your own implementation the string should be discoverable by Python. Here is an example for `MyOutputFormatter` in a package called `my_package`:

```
{
  "output": {
    "format": "my_package.MyOutputFormatter"
  }
}
```

Your own implementations must inherit from *output.OutputFormatter*.

File

Determines the path and filename of the output file produced by the output formatter. This path is relative to Python's current working directory.

```
{
  "output": {
    "file": "my_file.hdf5"
  }
}
```

9.1.3 Network architecture attributes

simulation_volume_x

The size of the X dimension of the simulation volume.

simulation_volume_z

The size of the Z dimension of the simulation volume.

```
{
  "network_architecture": {
    "simulation_volume_x": 150.0,
    "simulation_volume_z": 150.0
  }
}
```

Note: The Y can not be set directly as it is a result of stacking/placing the layers. It's possible to place cells outside of the simulation volume, and even to place layers outside of the volume, but it is not recommended behavior. The X and Z size are merely the base/anchor and a good indicator for the scale of the simulation, but they aren't absolute restrictions.

Warning: Do not modify these values directly on the configuration object: It will not rescale your layers. Use `resize` instead.

9.1.4 Layer attributes

position

(Optional) The XYZ coordinates of the bottom-left corner of the layer. Is overwritten if this layer is part of a *stack*.

```
"some_layer": {
  position: [100.0, 0.0, 100.0]
}
```

thickness

A fixed value of Y units.

Required unless the layer is scaled to other layers.

```
"some_layer": {
  "thickness": 600.0
}
```

xz_scale

(Optional) The scaling of this layer compared to the simulation volume. By default a layer's X and Z scaling are `[1.0, 1.0]` and so are equal to the simulation volume.

```
"some_layer": {
  "xz_scale": [0.5, 2.0]
}
```

xz_center

(Optional) Should this layer be aligned to the corner or the center of the simulation volume? Defaults to `False`.

stack

(Optional) Layers can be stacked on top of eachother if you define this attribute and give their stack configurations the same *stack_id*. The *position_in_stack* will determine in which order they are stacked, with the lower values placed on the bottom, receiving the lower Y coordinates. Exactly one layer per stack should define a *position* attribute in their stack configuration to pinpoint the bottom-left corner of the start of the stack.

stack_id

Unique identifier of the stack. All layers with the same stack id are grouped together.

position_in_stack

Unique identifier for the layer in the stack. Layers with larger positions will be placed on top of layers with lower ids.

position

This attribute needs to be specified in exactly one layer's *stack* dictionary and determines the starting (bottom-corner) position of the stack.

Example

This example defines 2 layers in the same stack:

```
{
  "layers": {
    "top_layer": {
    "thickness": 300,
      "stack": {
        "stack_id": 0,
        "position_in_stack": 1,
        "position": [0., 0., 0.]
      }
    },
    "bottom_layer": {
```

(continues on next page)

(continued from previous page)

```
      "thickness": 200,
      "stack": {
        "stack_id": 0,
        "position_in_stack": 0
      }
    }
  }
}
```

volume_scale

(Optional) The scaling factor used to scale this layer with respect to other layers. If this attribute is set, the *scale_from_layers* attribute is also required.

```
"some_layer": {
  "volume_scale": 10.0,
  "scale_from_layers": ["other_layer"]
}
```

scale_from_layers

(Optional) A list of layer names whose volume needs to be added up, and this layer's volume needs to be scaled to.

Example

Layer A has a volume of `2000.0`, Layer B has a volume of `3000.0`. Layer C specifies a *volume_scale* of `10.0` and *scale_from_layers* = `["layer_a", "layer_b"]`; this will cause it to become a cube (unless *volume_dimension_ratio* is specified) with a volume of `(2000.0 + 3000.0) * 10.0 = 50000.0`

volume_dimension_ratio

(Optional) Ratio of the rescaled dimensions. All given numbers are normalized to the Y dimension:

```
"some_layer": {
  "volume_scale": 10.0,
  "scale_from_layers": ["other_layer"],
  # Cube (default):
  "volume_dimension_ratio": [1., 1., 1.],
  # High pole:
  "volume_dimension_ratio": [1., 20., 1.], # Becomes [0.05, 1., 0.05]
  # Flat bed:
  "volume_dimension_ratio": [20., 1., 20.]
}
```

9.1.5 Cell Type Attributes

entity

If a cell type is marked as an entity with `"entity": true`, it will not receive a position in the simulation volume, but it will still be assigned an ID during placement that can be used for the connectivity step. This is for example useful for afferent fibers.

If *entity* is `true` no *morphology* or *plotting* needs to be specified.

relay

If a cell type is a *relay* it immediately relays all of its inputs to its target cells. Also known as a parrot neuron.

placement

Configuration node of the placement of this cell type. See *Placement Attributes*.

morphology

Configuration node of the morphologies of this cell type. This is still an experimental API, expect changes. See *Morphology attributes*.

plotting

Configuration node of the plotting attributes of this cell type. See *Plotting attributes*.

Example

```
```

9.1.6 Placement Attributes

Each configuration node needs to specify a `placement.PlacementStrategy` through *class*. Depending on the strategy another specific set of attributes is required. To see how to configure each `placement.PlacementStrategy` see the *List of placement strategies*.

class

A string containing a PlacementStrategy class name, including its module.

```
"class": "bsb.placement.ParticlePlacement"
```

9.1.7 Connectivity Attributes

The connectivity configuration node contains some basic attributes listed below and a set of strategy specific attributes that you can find in *List of connection strategies*.

class

A string containing a ConnectivityStrategy class name, including its module.

```
"class": "bsb.placement.VoxelIntersection"
```

from_types/to_types

A list of pre/postsynaptic selectors. Each selector is made up of a *type* to specify the cell type and a *compartments* list that specify the involved compartments for morphologically detailed connection strategies.

Deprecated since version 4.0: Each connectivity type will only be allowed to have 1 presynaptic and postsynaptic cell type. *from/to_types* will subsequently be renamed to *from/to_type*

```
"from_types": [
  {
    "type": "example_cell",
    "compartments": [
      "axon"
    ]
  }
]
```

9.1.8 Morphology attributes

9.1.9 Plotting attributes

color

The color representation for this cell type in plots. Can be any valid Plotly color string.

```
"color": "black"
"color": "#000000"
```

label

The legend label for this cell type in plots.

```
"label": "My Favourite Cells"
```

9.2 Reference Guide

Full reference guide to the most important parts of the documentation.

9.2.1 Command line interface module

This module contains all classes and functions required to run the scaffold from the command line.

exception bsb.cli.**ParseError**
Thrown when the parsing of a command string fails.

class bsb.cli.**ReplState**
Stores the REPL state and executes each step of the REPL.

add_parser_globals()
Adds subparsers and arguments that should be there in any state.

add_subparser(**args*, ***kwargs*)
Add a top level subparser to the current REPL parser.

clear_prefix()
Clear the REPL prefix.

close_hdf5()
Closes the currently open HDF5 file.

Raises *ParseError* – Raised if there's no open HDF5 file.

Return type None

destroy_globals()
Always called before the REPL exits to clean up open resources.

exit_repl(*args*)
Exit the REPL.

open_hdf5(*args*)
Callback function that handles the `open hdf5` command.

Parameters **args** (*Namespace*) – Result of ArgumentParser.parse_args()

Return type None

open_morphology_repository(*args*)
Callback function that handles the `open mr` command.

Parameters **args** (*Namespace*) – Result of ArgumentParser.parse_args()

Return type None

repl()
Execute the next repl step.

set_next_state(*state*)
Set the next REPL state.

Parameters **state** (*string*) – The next state. For each state there should be a set_parser_``state``_state function (e.g. *set_parser_base_state()*).

Return type None

set_parser_base_hdf5_state()
Adds the HDF5 state subparsers and arguments to the REPL parser.

set_parser_base_mr_state()
Adds the morphology repository state subparsers and arguments to the REPL parser.

set_parser_base_state()
Adds the initial subparsers and arguments to the REPL parser.

set_reply(*message*)
Set the REPL reply, to be printed to the user at the end of this step.

Parameters **message** (*string*) – The reply to print.

Return type None

update_parser()
Creates a new parser for the next REPL step. Tries to add subparsers and arguments if the method "set_parser_``state``_state" is callable.

bsb.cli.**check_positive_factory**(*name*)
Return a function to report whether a certain value is a positive integer. If it isn't, raise an ArgumentTypeError.

bsb.cli.**repl_plot_morphology**(*morphology_repository*, *args*)
Callback function that handles `plot` command in the *base_mr* state.

bsb.cli.**repl_view_hdf5**(*handle*, *args*)
Callback function that handles `view` command in the *base_hdf5* state.

bsb.cli.**repl_voxelize**(*morphology_repository*, *args*)
Callback function that handles `voxelize` command in the *base_mr* state.

bsb.cli.**scaffold_cli**()
console_scripts entry point for the scaffold package. Will start the CLI handler or REPL handler.

bsb.cli.**start_cli**()
Scaffold package CLI handler

bsb.cli.**start_repl**()
Scaffold package REPL handler. Will parse user commands.

9.2.2 Configuration module

9.2.3 Connectivity module

exception bsb.connectivity.**AdapterError**(**args*, ***kwargs*)

class bsb.connectivity.**AllToAll**
All to all connectivity between two neural populations

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**ArborError**(**args*, ***kwargs*)

exception bsb.connectivity.**AttributeMissingError**(**args*, ***kwargs*)

exception bsb.connectivity.**CastConfigurationError**(**args*, ***kwargs*)

exception bsb.connectivity.**CastError**(**args*, ***kwargs*)

exception bsb.connectivity.**CircularMorphologyError**(**args*, ***kwargs*)

exception bsb.connectivity.**ClassError**(**args*, ***kwargs*)

exception bsb.connectivity.**CompartmentError**(**args*, ***kwargs*)

exception bsb.connectivity.**ConfigurableCastError**(**args*, ***kwargs*)

exception bsb.connectivity.**ConfigurableClassNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**ConfigurationError**(**args*, ***kwargs*)

exception bsb.connectivity.**ConfigurationFormatError**(**args*, ***kwargs*)

exception bsb.connectivity.**ConfigurationWarning**

class bsb.connectivity.**ConnectionStrategy**

exception bsb.connectivity.**ConnectivityError**(**args*, ***kwargs*)

exception bsb.connectivity.**ConnectivityWarning**

class bsb.connectivity.**ConnectomeAscAxonPurkinje**
Legacy implementation for the connections between ascending axons and purkinje cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeBCSCPurkinje**
Legacy implementation for the connections between basket cells,stellate cells and purkinje cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeDcnGlyGolgi**
Implementation for the connections between mossy fibers and glomeruli. The connectivity is somatotopic and

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeDcnGolgi**
Implementation for the connections between mossy fibers and glomeruli. The connectivity is somatotopic and

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeDcnGranule**
Implementation for the connections between mossy fibers and glomeruli. The connectivity is somatotopic and

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeGapJunctions**
Legacy implementation for gap junctions between a cell type.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeGapJunctionsGolgi**
Legacy implementation for Golgi cell gap junctions.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeGlomerulusGolgi
Legacy implementation for the connections between Golgi cells and glomeruli.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeGlomerulusGranule
Legacy implementation for the connections between glomeruli and granule cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeGolgiGlomerulus
Legacy implementation for the connections between glomeruli and Golgi cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeGolgiGranule
Legacy implementation for the connections between Golgi cells and granule cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeGranuleGolgi
Legacy implementation for the connections between Golgi cells and glomeruli.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeIOMolecular
Legacy implementation for the connection between inferior olive and Molecular layer interneurons. As this is a spillover-mediated non-synaptic connection depending on the IO to Purkinje cells, each interneuron connected to a PC which is receving input from one IO, is also receiving input from that IO

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeIOPurkinje
Legacy implementation for the connection between inferior olive and Purkinje cells. Purkinje cells are clustered (number of clusters is the number of IO cells), and each clusters is innervated by 1 IO cell

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.ConnectomeMossyDCN
Implementation for the connection between mossy fibers and DCN cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomeMossyGlomerulus**
Implementation for the connections between mossy fibers and glomeruli. The connectivity is somatotopic and

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomePFInterneuron**
Legacy implementation for the connections between parallel fibers and a molecular layer interneuron cell_type.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomePFPurkinje**
Legacy implementation for the connections between parallel fibers and purkinje cells.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**ConnectomePurkinjeDCN**
Legacy implementation for the connection between purkinje cells and DCN cells. Also rotates the dendritic trees of the DCN.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**ContinuityError**(**args*, ***kwargs*)

class bsb.connectivity.**Convergence**
Implementation of a general convergence connectivity between two populations of cells (this does not work with entities)

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**DataNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**DataNotProvidedError**(**args*, ***kwargs*)

exception bsb.connectivity.**DatasetNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**DeviceConnectionError**(**args*, ***kwargs*)

exception bsb.connectivity.**DynamicClassError**(**args*, ***kwargs*)

class bsb.connectivity.**ExternalConnections**
Load the connection matrix from an external source.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**ExternalSourceError**(**args*, ***kwargs*)

class bsb.connectivity.**FiberIntersection**
FiberIntersection connection strategies voxelize a fiber and find its intersections with postsynaptic cells. It's a specific case of VoxelIntersection.

For each presynaptic cell, the following steps are executed:

1. Extract the FiberMorphology
2. Interpolate points on the fiber until the spatial resolution is respected
3. transform
4. Interpolate points on the fiber until the spatial resolution is respected
5. Voxelize (generates the voxel_tree associated to this morphology)
6. Check intersections of presyn bounding box with all postsyn boxes
7. Check intersections of each candidate postsyn with current presyn voxel_tree

intersect_voxel_tree(*from_voxel_tree*, *to_cloud*, *to_pos*)
Similarly to *intersect_clouds* from *VoxelIntersection*, it finds intersecting voxels between a from_voxel_tree and a to_cloud set of voxels

Parameters

- **from_voxel_tree** – tree built from the voxelization of all branches in the fiber (in absolute coordinates)
- **to_cloud** (*VoxelCloud*) – voxel cloud associated to a to_cell morphology
- **to_pos** (*list*) – 3-D position of to_cell neuron

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**FiberTransform**

exception bsb.connectivity.**IncompleteExternalMapError**(**args*, ***kwargs*)

exception bsb.connectivity.**IncompleteMorphologyError**(**args*, ***kwargs*)

exception bsb.connectivity.**IntersectionDataNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**InvalidDistributionError**(**args*, ***kwargs*)

exception bsb.connectivity.**KernelLockedError**(**args*, ***kwargs*)

exception bsb.connectivity.**KernelWarning**

exception bsb.connectivity.**LayerNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**MissingMorphologyError**(**args*, ***kwargs*)

exception bsb.connectivity.**MissingSourceError**(**args*, ***kwargs*)

exception bsb.connectivity.**MorphologyDataError**(**args*, ***kwargs*)

exception bsb.connectivity.**MorphologyError**(**args*, ***kwargs*)

exception bsb.connectivity.**MorphologyRepositoryError**(**args*, ***kwargs*)

exception bsb.connectivity.**MorphologyWarning**

exception bsb.connectivity.**NestError**(**args*, ***kwargs*)

exception bsb.connectivity.**NestKernelError**(**args*, ***kwargs*)

exception bsb.connectivity.**NestModelError**(**args*, ***kwargs*)

exception bsb.connectivity.**NestModuleError**(**args*, ***kwargs*)

exception bsb.connectivity.**NeuronError**(**args*, ***kwargs*)

exception bsb.connectivity.**OrderError**(**args*, ***kwargs*)

exception bsb.connectivity.**ParallelIntegrityError**(**args*, ***kwargs*)

exception bsb.connectivity.**PlacementError**(**args*, ***kwargs*)

exception bsb.connectivity.**PlacementWarning**

exception bsb.connectivity.**QuiverFieldWarning**

class bsb.connectivity.**QuiverTransform**

QuiverTransform applies transformation to a FiberMorphology, based on an orientation field in a voxelized volume. Used for parallel fibers.

transform_branch(*branch*, *offset*)

Compute bending transformation of a fiber branch (discretized according to original compartments and configured resolution value). The transformation is a rotation of each segment/compartment of each fiber branch to align to the cross product between the orientation vector and the transversal direction vector (i.e. cross product between fiber morphology/parent branch orientation and branch direction): compartment[n+1].start = compartment[n].end cross_prod = orientation_vector X transversal_vector or transversal_vector X orientation_vector compartment[n+1].end = compartment[n+1].start + cross_prod * length_comp

Parameters **branch** (*Branch object*) – a branch of the current fiber to be transformed

Returns a transformed branch

validate()

Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**ReceptorSpecificationError**(**args*, ***kwargs*)

exception bsb.connectivity.**RelayError**(**args*, ***kwargs*)

exception bsb.connectivity.**RepositoryWarning**

exception bsb.connectivity.**ResourceError**(**args*, ***kwargs*)

class bsb.connectivity.**SatelliteCommonPresynaptic**

Connectivity for satellite neurons (homologous to center neurons)

validate()

Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**ScaffoldError**(**args*, ***kwargs*)

exception bsb.connectivity.**ScaffoldWarning**

exception bsb.connectivity.**SimulationNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**SimulationWarning**

exception bsb.connectivity.**SourceQualityError**(**args*, ***kwargs*)

exception bsb.connectivity.**SpatialDimensionError**(**args*, ***kwargs*)

exception bsb.connectivity.**SuffixTakenError**(**args*, ***kwargs*)

class bsb.connectivity.**TouchDetector**

Connectivity based on intersection of detailed morphologies

validate()

Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.connectivity.**TouchingConvergenceDivergence**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**TransmitterError**(**args*, ***kwargs*)

exception bsb.connectivity.**TreeError**(**args*, ***kwargs*)

exception bsb.connectivity.**TypeNotFoundError**(**args*, ***kwargs*)

exception bsb.connectivity.**UnionCastError**(**args*, ***kwargs*)

exception bsb.connectivity.**UnknownDistributionError**(**args*, ***kwargs*)

exception bsb.connectivity.**UnknownGIDError**(**args*, ***kwargs*)

exception bsb.connectivity.**UserUserDeprecationWarning**

class bsb.connectivity.**VoxelIntersection**
This strategy voxelizes morphologies into collections of cubes, thereby reducing the spatial specificity of the provided traced morphologies by grouping multiple compartments into larger cubic voxels. Intersections are found not between the seperate compartments but between the voxels and random compartments of matching voxels are connected to eachother. This means that the connections that are made are less specific to the exact morphology and can be very useful when only 1 or a few morphologies are available to represent each cell type.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

exception bsb.connectivity.**VoxelTransformError**(**args*, ***kwargs*)

exception bsb.connectivity.**VoxelizationError**(**args*, ***kwargs*)

bsb.connectivity.**report**(**message*, *level=2*, *ongoing=False*, *token=None*, *nodes=None*, *all_nodes=False*)
Send a message to the appropriate output channel.

Parameters

- **message** (*string*) – Text message to send.
- **level** (*int*) – Verbosity level of the message.
- **ongoing** – The message is part of an ongoing progress report. This replaces the endline (*n*) character with a carriage return (*r*) character

bsb.connectivity.**warn**(*message*, *category=None*)
Send a warning.

Parameters

- **message** (*str*) – Warning message
- **category** – The class of the warning.

9.2.4 Exceptions module

exception bsb.exceptions.**AdapterError**(**args*, ***kwargs*)

exception bsb.exceptions.**ArborError**(**args*, ***kwargs*)

exception bsb.exceptions.**AttributeMissingError**(**args*, ***kwargs*)

exception bsb.exceptions.**CastConfigurationError**(**args*, ***kwargs*)

exception bsb.exceptions.**CastError**(**args*, ***kwargs*)

exception bsb.exceptions.**CircularMorphologyError**(**args*, ***kwargs*)

exception bsb.exceptions.**ClassError**(**args*, ***kwargs*)

exception bsb.exceptions.**CompartmentError**(**args*, ***kwargs*)

exception bsb.exceptions.**ConfigurableCastError**(**args*, ***kwargs*)

exception bsb.exceptions.**ConfigurableClassNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**ConfigurationError**(**args*, ***kwargs*)

exception bsb.exceptions.**ConfigurationFormatError**(**args*, ***kwargs*)

exception bsb.exceptions.**ConfigurationWarning**

exception bsb.exceptions.**ConnectivityError**(**args*, ***kwargs*)

exception bsb.exceptions.**ConnectivityWarning**

exception bsb.exceptions.**ContinuityError**(**args*, ***kwargs*)

exception bsb.exceptions.**DataNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**DataNotProvidedError**(**args*, ***kwargs*)

exception bsb.exceptions.**DatasetNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**DeviceConnectionError**(**args*, ***kwargs*)

exception bsb.exceptions.**DynamicClassError**(**args*, ***kwargs*)

exception bsb.exceptions.**ExternalSourceError**(**args*, ***kwargs*)

exception bsb.exceptions.**IncompleteExternalMapError**(**args*, ***kwargs*)

exception bsb.exceptions.**IncompleteMorphologyError**(**args*, ***kwargs*)

exception bsb.exceptions.**IntersectionDataNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**InvalidDistributionError**(**args*, ***kwargs*)

exception bsb.exceptions.**KernelLockedError**(**args*, ***kwargs*)

exception bsb.exceptions.**KernelWarning**

exception bsb.exceptions.**LayerNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**MissingMorphologyError**(**args*, ***kwargs*)

exception bsb.exceptions.**MissingSourceError**(**args*, ***kwargs*)

exception bsb.exceptions.**MorphologyDataError**(**args*, ***kwargs*)

exception bsb.exceptions.**MorphologyError**(**args*, ***kwargs*)

exception bsb.exceptions.**MorphologyRepositoryError**(**args*, ***kwargs*)

exception bsb.exceptions.**MorphologyWarning**

exception bsb.exceptions.**NestError**(**args*, ***kwargs*)

exception bsb.exceptions.**NestKernelError**(**args*, ***kwargs*)

exception bsb.exceptions.**NestModelError**(**args*, ***kwargs*)

exception bsb.exceptions.**NestModuleError**(**args*, ***kwargs*)

exception bsb.exceptions.**NeuronError**(**args*, ***kwargs*)

exception bsb.exceptions.**OrderError**(**args*, ***kwargs*)

exception bsb.exceptions.**ParallelIntegrityError**(**args*, ***kwargs*)

exception bsb.exceptions.**PlacementError**(**args*, ***kwargs*)

exception bsb.exceptions.**PlacementWarning**

exception bsb.exceptions.**QuiverFieldWarning**

exception bsb.exceptions.**ReceptorSpecificationError**(**args*, ***kwargs*)

exception bsb.exceptions.**RelayError**(**args*, ***kwargs*)

exception bsb.exceptions.**RepositoryWarning**

exception bsb.exceptions.**ResourceError**(**args*, ***kwargs*)

exception bsb.exceptions.**ScaffoldError**(**args*, ***kwargs*)

exception bsb.exceptions.**ScaffoldWarning**

exception bsb.exceptions.**SimulationNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**SimulationWarning**

exception bsb.exceptions.**SourceQualityError**(**args*, ***kwargs*)

exception bsb.exceptions.**SpatialDimensionError**(**args*, ***kwargs*)

exception bsb.exceptions.**SuffixTakenError**(**args*, ***kwargs*)

exception bsb.exceptions.**TransmitterError**(**args*, ***kwargs*)

exception bsb.exceptions.**TreeError**(**args*, ***kwargs*)

exception bsb.exceptions.**TypeNotFoundError**(**args*, ***kwargs*)

exception bsb.exceptions.**UnionCastError**(**args*, ***kwargs*)

exception bsb.exceptions.**UnknownDistributionError**(**args*, ***kwargs*)

exception bsb.exceptions.**UnknownGIDError**(**args*, ***kwargs*)

exception bsb.exceptions.**UserUserDeprecationWarning**

exception bsb.exceptions.**VoxelTransformError**(**args*, ***kwargs*)

exception bsb.exceptions.**VoxelizationError**(**args*, ***kwargs*)

9.2.5 Functions module

Contains all the mathematical helper functions used throughout the scaffold. Differs from helpers.py only categorically. Helpers.py contains functions, classes and general logic that supports the scaffold, while functions.py contains a collection of mathematical functions.

bsb.functions.**add_y_axis**(*points*, *min*, *max*)
Add random values to the 2nd column of a matrix of 2D points.

bsb.functions.**apply_2d_bounds**(*possible_points*, *cell_bounds*)
Compare a 2xN matrix of XZ coordinates to a matrix 2x3 with a minimum column and maximum column of XYZ coordinates.

bsb.functions.**compute_circle**(*center*, *radius*, *n_samples=50*)
Create *n_samples* points on a circle based on given *center* and *radius*.

Parameters

- **center** (*array-like*) – XYZ vector of the circle center
- **radius** (*scalar value*) – Radius of the circle
- **n_samples** (*int*) – Amount of points on the circle.

bsb.functions.**compute_intersection_slice**(*l1*, *l2*)
Returns the indices of elements in l1 that intersect with l2.

bsb.functions.**exclude_index**(*arr*, *index*)
Return a new list with the element at *index* removed.

bsb.functions.**get_candidate_points**(*center*, *radius*, *bounds*, *min_*, *max_*, *return_=False*)
Returns a list of points that are suited next candidates in a random walk.

Computes a circle of points between $2r +$ distance away from the center and removes any points that lie outside of the given bounds.

Parameters

- **center** (*list*) – 2D position of the starting point.
- **radius** (*float*) – Unit distance radius of the particle at the center point.
- **bounds** (*ndarray*) – A 2x3 matrix where the first column are the minimum XYZ and the last column the maximum XYZ.
- **min_** (*float*) – Lower bound of epsilon used to calculate random distance.
- **max_** (*float*) – Upper bound of epsilon used to calculate random distance.
- **return_** – If *True* the candidates and used to calculate them will be returned as a tuple.

bsb.functions.**get_distances**(*candidates*, *point*)
Return the distances of a list of points to a common point

bsb.functions.**poisson_train**(*frequency*, *duration*, *start_time=0*, *seed=None*)
Generator function for a Homogeneous Poisson train.

Parameters

- **frequency** – The mean spiking frequency.
- **duration** – Maximum duration.
- **start_time** – Timestamp.

- **seed** – Seed for the random number generator. If None, this will be decided by numpy, which chooses the system time.

Returns A relative spike time from t=start_time, in seconds (not ms).

EXAMPLE:

```
# Make a list of spikes at 20 Hz for 3 seconds
spikes = [i for i in poisson_train(20, 3)]
```

9.2.6 Helpers module

class bsb.helpers.**CastableConfigurableClass**

class bsb.helpers.**ConfigurableClass**

A class that can be configured.

cast_config()

Casts/validates values imported onto this object from configuration files to their final form. The *casts* dictionary should contain the key of the attribute and a function that takes a value as only argument. This dictionary will be used to cast the attributes when cast_config is called.

abstract validate()

Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.helpers.**DistributionConfiguration**

Cast a configuration node into a *scipy.stats* distribution.

fallback

alias of `float`

validate()

Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.helpers.**EvalConfiguration**

validate()

Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.helpers.**FloatEvalConfiguration**

fallback

alias of `float`

class bsb.helpers.**ListEvalConfiguration**

fallback

alias of `list`

class bsb.helpers.**OptionallyCastable**

bsb.helpers.**assert_attr_array**(*section*, *attr*, *section_name*)

Asserts that an attribute exists on a dictionary or object, and that it is an array.

Parameters

- **section** (*dict, object*) – Dictionary or object that needs to contain the attribute.
- **attr** (*string*) – Attribute name.
- **section_name** (*string*) – Name of the section to print out the location of the missing attribute.

bsb.helpers.**assert_attr_in**(*section*, *attr*, *values*, *section_name*)
Assert that the attribute is present in the section dictionary and that its value is included in the given array.

bsb.helpers.**continuity_hop**(*iterator*)
Hop over a continuity list in steps of 2, returning the start & count pairs.

bsb.helpers.**continuity_list**(*iterable*, *step=1*)
Return a compacted notation of a list of nearly continuous numbers.

The `iterable` will be iterated and chains of continuous numbers will be determined. Each chain will then be added to the output format as a starting number and count.

Example: `[4,5,6,7,8,9,12] ==> [4,6,12,1]`

Parameters

- **iterable** (*iter*) – The collection of elements to be compacted.
- **step** – `iterable[i]` needs to be equal to `iterable[i - 1] + step` for them to considered continuous.

bsb.helpers.**expand_continuity_list**(*iterable*, *step=1*)
Return the full set of items associated with the continuity list, as formatted by *helpers.continuity_list()*.

bsb.helpers.**iterate_continuity_list**(*iterable*, *step=1*)
Generate the continuity list

bsb.helpers.**listify_input**(*value*)
Turn any non-list values into a list containing the value. Sequences will be converted to a list using *list()*, *None* will be replaced by an empty list.

9.2.7 Models module

class bsb.models.**CellType**(*name*, *placement=None*)
A CellType represents a population of cells.

list_all_morphologies()
Return a list of all the morphology identifiers that can represent this cell type in the simulation volume.

place()
Place this cell type.

set_morphology(*morphology*)
Set the Morphology class for this cell type.

Parameters **morphology** (*Instance of a subclass of scaffold.morphologies.Morphology*) – Defines the geometrical constraints for the axon and dendrites of the cell type.

set_placement(*placement*)
Set the placement strategy for this cell type.

validate()
Check whether this CellType is valid to be used in the simulation.

class bsb.models.**ConnectivitySet**(*handler*, *tag*)
Connectivity sets store connections.

property connection_types
Return all the ConnectionStrategies that contributed to the creation of this connectivity set.

property connections
Return a list of `Intersections`. Connections contain pre- & postsynaptic identifiers.

property from_identifiers
Return a list with the presynaptic identifier of each connection.

get_postsynaptic_types()
Return a list of the postsynaptic cell types found in this set.

get_presynaptic_types()
Return a list of the presynaptic cell types found in this set.

has_compartment_data()
Check if compartment data exists for this connectivity set.

property intersections
Return a list of `Intersections`. Intersections contain pre- & postsynaptic identifiers and the intersecting compartments.

property meta
Retrieve the metadata associated with this connectivity set. Returns `None` if the connectivity set does not exist.

Returns Metadata

Return type dict

property to_identifiers
Return a list with the postsynaptic identifier of each connection.

class bsb.models.**Layer**(*name*, *origin*, *dimensions*, *scaling=True*)
A Layer represents a compartment of the topology of the simulation volume that slices the volume in horizontally stacked portions.

scale_to_reference()
Compute scaled layer volume

To compute layer thickness, we scale the current layer to the combined volume of the reference layers. A ratio between the dimension can be specified to alter the shape of the layer. By default equal ratios are used and a cubic layer is obtained (given by *dimension_ratios*).

The volume of the current layer (= X*Y*Z) is scaled with respect to the volume of reference layers by a factor *volume_scale*, so:

X*Y*Z = volume_reference_layers / volume_scale [A]

Supposing that the current layer dimensions (X,Y,Z) are each one depending on the dimension Y according to *dimension_ratios*, we obtain:

X*Y*Z = (Y*dimension_ratios[0] * Y * (Y*dimension_ratios[2]) [B] X*Y*Z = (Y^3) * prod(dimension_ratios) [C]

Therefore putting together [A] and [C]: (Y^3) * prod(dimension_ratios) = volume_reference_layers / volume_scale

from which we derive the normalized_size Y, according to the following formula:

Y = cubic_root((volume_reference_layers * volume_scale) / prod(dimension_ratios))

class bsb.models.**PlacementSet**(*handler*, *cell_type*)

Fetches placement data from storage. You can either access the parallel-array datasets `.identifiers`, `.positions` and `.rotations` individually or create a collection of `Cells` that each contain their own identifier, position and rotation.

Note: Use `core.get_placement_set()` to correctly obtain a PlacementSet.

---

property cells

Reorganize the available datasets into a collection of `Cells`

property identifiers

Return a list of cell identifiers.

property positions

Return a dataset of cell positions.

property rotations

Return a dataset of cell rotations.

Raises DatasetNotFoundError when there is no rotation information for this cell type.

9.2.8 Morphologies module

class bsb.morphologies.**Branch**(**args*, *labels=None*)

A vector based representation of a series of point in space. Can be a root or connected to a parent branch. Can be a terminal branch or have multiple children.

attach_child(*branch*)

Attach a branch as a child to this branch.

Parameters **branch** (*Branch*) – Child branch

property children

Collection of the child branches of this branch.

Returns list of *Branches*

Return type list

detach_child(*branch*)

Remove a branch as a child from this branch.

Parameters **branch** (*Branch*) – Child branch

label(**labels*)

Add labels to every point on the branch. See `label_points` to label individual points.

Parameters **labels** (*str*) – Label(s) for the branch.

label_points(*label*, *mask*)

Add labels to specific points on the branch. See `label` to label the entire branch.

Parameters

- **label** (*str*) – Label to apply to the points.
- **mask** (*np.ndarray(dtype=bool, shape=(branch_size,))*) – Boolean mask equal in size to the branch that determines which points get labelled.

label_walk()

Iterate over the labels of each point in the branch.

property points
Return the vectors of this branch as a matrix.

property size
Returns the amount of points on this branch

Returns Number of points on the branch.

Return type int

property terminal
Returns whether this branch is terminal or has children.

Returns True if this branch has no children, False otherwise.

Return type bool

to_compartments(*start_id=0*, *parent=None*)
Convert the branch to compartments.

Deprecated since version 3.6: Use the vectors and points API instead (`.points`, `.walk()`)

walk()
Iterate over the points in the branch.

class bsb.morphologies.**Compartment**(*start*, *end*, *radius*, *id=None*, *labels=None*, *parent=None*, *section_id=None*, *morphology=None*)
Compartments are line segments with a radius. They can be constructed from the points on a *Branch* or by concatenating the results of a depth-first iteration of the branches of a *Morphology*.

classmethod from_template(*template*, ***kwargs*)
Create a compartment based on a template compartment. Accepts any keyword argument to overwrite or add attributes.

class bsb.morphologies.**GolgiCellGeometry**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.morphologies.**GranuleCellGeometry**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.morphologies.**Morphology**(*roots*)
A multicompartmental spatial representation of a cell based on connected 3D compartments.

Todo Uncouple from the MorphologyRepository and merge with TrueMorphology.

property branches
Return a depth-first flattened array of all branches.

flatten(*vectors=None*, *matrix=False*, *labels=None*)
Return the flattened vectors of the morphology

Parameters **vectors** (*list of str*) – List of vectors to return such as ['x', 'y', 'z'] to get the positional vectors.

Returns Tuple of the vectors in the given order, if *matrix* is True a matrix composed of the vectors is returned instead.

Return type tuple of ndarrays (*matrix=False*) or matrix (*matrix=True*)

get_branches(*labels=None*)
Return a depth-first flattened array of all or the selected branches.

Parameters **labels** (*list*) – Names of the labels to select.

Returns List of all branches or all branches with any of the labels when given

Return type list

rotate(*v0*, *v*)
Rotate a morphology to be oriented as vector v, supposing to start from orientation v0. norm(v) = norm(v0) = 1 Rotation matrix R, representing a rotation of angle alpha around vector k

to_compartments()
Return a flattened array of compartments

class bsb.morphologies.**NilCompartment**

class bsb.morphologies.**NoGeometry**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.morphologies.**PurkinjeCellGeometry**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.morphologies.**RadialGeometry**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.morphologies.**Representation**

bsb.morphologies.**branch_iter**(*branch*)
Iterate over a branch and all of its children depth first.

9.2.9 Networks module

bsb.networks.**reduce_branch**(*branch*, *branch_points*)
Reduce a branch (list of points) to only its start and end point and the intersection with a list of known branch points.

9.2.10 Output module

class bsb.output.**HDF5Formatter**
Stores the output of the scaffold as a single HDF5 file. Is also a MorphologyRepository and an HDF5TreeHandler.

exists()
Check if the resource exists.

get_cells_of_type(*name*, *entity=False*)
Return the position matrix for a specific cell type.

get_connectivity_set(*tag*)
Return a connectivity set.

Parameters **tag** (*string*) – Key of the connectivity set in the *connections* group.

Returns The connectivity set.

Return type ConnectivitySet

Raises DatasetNotFoundError

get_connectivity_set_connection_types(*tag*)
Return all the ConnectionStrategies that contributed to the creation of this connectivity set.

get_connectivity_set_meta(*tag*)
Return the metadata associated with this connectivity set.

get_connectivity_sets()
Return all the ConnectivitySets present in the network file.

get_simulator_output_path(*simulator_name*)
Return the path where a simulator can dump preliminary output.

has_cells_of_type(*name*, *entity=False*)
Check whether the position matrix for a certain cell type is present.

init_scaffold()
Initialize the scaffold when it has been loaded from an output file.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.output.**HDF5ResourceHandler**

get_handle(*mode='r'*)
Open an HDF5 resource.

release_handle(*handle*)
Close the MorphologyRepository storage resource.

class bsb.output.**HDF5TreeHandler**
TreeHandler that uses HDF5 as resource storage

class bsb.output.**MorphologyCache**(*morphology_repository*)
Loads and caches `morphologies` so that each morphology is loaded only once and its instance is shared among all cells with that Morphology. Saves a lot on memory, but the Morphology should be treated as read only.

rotate_all_morphologies(*phi_step*, *theta_step=None*)
Extracts all unrotated morphologies from a morphology_repository and creates rotated versions, at sampled orientations in the 3D space

Parameters

- **phi_step** (*int, optional*) – Resolution of azimuth angle sampling, in degrees
- **theta_step** – Resolution of elevation angle sampling, in degrees

class bsb.output.**MorphologyRepository**(*file=None*)

get_handle(*mode='r'*)
Open the HDF5 storage resource and initialise the MorphologyRepository structure.

get_morphology(*name*, *scaffold=None*)
Load a morphology from repository data

import_arbz(*name*, *cls*, *overwrite=False*)
Import an Arborize model as a morphology.

Arborize models make some assumptions about morphologies, inherited from how NEURON deals with it: There is only 1 root, and the soma is at the beginning of this root. This is not necesarily so for morphologies in general in the BSB that can have as many roots as they want.

import_swc(*file*, *name*, *tags=[]*, *overwrite=False*)
Import and store .swc file contents as a morphology in the repository.

list_morphologies(*include_rotations=False*, *only_rotations=False*, *cell_type=None*)
Return a list of morphologies in a morphology repository, filtered by rotation and/or cell type.

Parameters

- **include_rotations** (*bool*) – Include each cached rotation of each morphology.
- **only_rotations** (*bool*) – Get only the rotated caches of the morphologies.
- **cell_type** – Specify the cell type for which you want to extract the morphologies.
- **cell_type** – CellType

Returns List of morphology names

Return type list

class bsb.output.**OutputFormatter**

abstract exists()
Check if the resource exists.

abstract get_cells_of_type(*name*)
Return the position matrix for a specific cell type.

abstract get_connectivity_set(*tag*)
Return a connectivity set.

Parameters **tag** (*string*) – Key of the connectivity set in the *connections* group.

Returns The connectivity set.

Return type ConnectivitySet

Raises DatasetNotFoundError

abstract get_connectivity_set_connection_types(*tag*)
Return the connection types that contributed to this connectivity set.

abstract get_connectivity_set_meta(*tag*)
Return the meta dictionary of this connectivity set.

abstract get_connectivity_sets()
Return all connectivity sets.

Returns List of connectivity sets.

Return type ConnectivitySet

abstract get_simulator_output_path(*simulator_name*)
Return the path where a simulator can dump preliminary output.

abstract has_cells_of_type(*name*)
Check whether the position matrix for a certain cell type is present.

abstract init_scaffold()
Initialize the scaffold when it has been loaded from an output file.

class bsb.output.**ResourceHandler**

abstract get_handle(*mode=None*)
Open the output resource and return a handle.

abstract release_handle(*handle*)
Close the open output resource and release the handle.

class bsb.output.**TreeHandler**
Interface that allows a ResourceHandler to handle storage of TreeCollections.

9.2.11 Placement module

9.2.12 Plotting module

bsb.plotting.**hdf5_gdf_plot_spike_raster**(*spike_recorders*, *input_region=None*, *fig=None*, *show=True*)
Create a spike raster plot from an HDF5 group of spike recorders saved from NEST gdf files. Each HDF5 dataset includes the spike timings of the recorded cell populations, with spike times in the first row and neuron IDs in the second row.

bsb.plotting.**hdf5_plot_spike_raster**(*spike_recorders*, *input_region=None*, *show=True*, *cutoff=0*, *cell_type_sort=None*, *cell_sort=None*)
Create a spike raster plot from an HDF5 group of spike recorders.

Parameters

- **input_region** (*2-element list-like*) – Specifies an interval [min, max] on the x axis to highlight as active input to the simulation.
- **show** (*bool*) – Immediately plot the result
- **cutoff** (*float*) – Amount of ms initial simulation to ignore.
- **cell_type_sort** (*function-like*) – A function to sort the cell types. Must take 2 dictionaries as arguments, being the raster plot's x values per label and y values per label. Must return an array labels matching those of the x and y values to order them.
- **cell_sort** (*function-like*) – A function that takes the cell type label and set of ids and returns a map to sort them.

bsb.plotting.**plot_network**(*network*, *fig=None*, *cubic=True*, *swapaxes=True*, *show=True*, *legend=True*, *from_memory=True*)
Plot a network, either from the current cache or the storage.

bsb.plotting.**set_morphology_scene_range**(*scene*, *offset_morphologies*)
Set the range on a scene containing multiple morphologies.

Parameters

- **scene** – A scene of the figure. If the figure itself is given, `figure.layout.scene` will be used.
- **offset_morphologies** – A list of tuples where the first element is offset and the 2nd is the `Morphology`

9.2.13 Postprocessing module

class bsb.postprocessing.**AscendingAxonLengths**

class bsb.postprocessing.**BidirectionalContact**

class bsb.postprocessing.**CerebellumLabels**

class bsb.postprocessing.**DCNRotations**
Create a matrix of planes tilted between -45° and 45°, storing id and the planar coefficients a, b, c and d for each DCN cell

class bsb.postprocessing.**DCN_large_differentiation**
Extract from the overall DCN glutamate large cells (GADnL) 2 subpopulations that are involved in the construction of the NucleoCortical pathways

class bsb.postprocessing.**LabelMicrozones**

class bsb.postprocessing.**MissingAxon**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.postprocessing.**PostProcessingHook**

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.postprocessing.**SpoofDetails**
Create fake morphological intersections between already connected non-detailed connection types.

9.2.14 Scaffold class

from_hdf5

Bootstrap a scaffold instance from an HDF5 file.

9.2.15 Simulation module

9.2.16 Simulators

NEST module

class bsb.simulators.nest.**NestAdapter**
Interface between the scaffold model and the NEST simulator.

broadcast(*data*, *root=0*)
Broadcast data over MPI

collect_output(*simulator*)
Collect the output of a simulation that completed

connect_neurons()
Connect the cells in NEST according to the connection model configurations

create_devices()
Create the configured NEST devices in the simulator

create_model(*cell_model*)
Create a NEST cell model in the simulator based on a cell model configuration.

create_neurons()
Create a population of nodes in the NEST simulator based on the cell model configurations.

create_synapse_model(*connection_model*)
Create a NEST synapse model in the simulator based on a synapse model configuration.

get_rank()
Return the rank of the current node.

get_size()
Return the size of the collection of all distributed nodes.

prepare()
This method turns a stored HDF5 network architecture and returns a runnable simulator.

Returns A simulator prepared to run a simulation according to the given configuration.

simulate(*simulator*)
Start a simulation given a simulator object.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.simulators.nest.**NestCell**(*adapter*)

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.simulators.nest.**NestConnection**(*adapter*)

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.simulators.nest.**NestDevice**(*adapter*)

get_nest_targets()
Return the targets of the stimulation to pass into the nest.Connect call.

validate()
Must be implemented by child classes. Raise exceptions when invalid configuration parameters are received.

class bsb.simulators.nest.**NestEntity**(*adapter*)

NEURON module

9.2.17 Trees module

class bsb.trees.**TreeCollection**(*name*, *handler*)
Keeps track of a collection of KDTrees in cooperation with a TreeHandler.

bsb.trees.**is_valid_tree_name**(*name*)
Validate whether a given string is fit to be the name of a tree in a TreeCollection. Must not contain any plus signs, parentheses or colons.

9.2.18 Voxels module

class bsb.voxels.**HitDetector**(*detector*)
Wrapper class for commonly used hit detectors in the voxelization process.

classmethod for_rtree(*tree*)
Factory function that creates a hit detector for the given morphology.

Parameters **morphology** (TrueMorphology) – A morphology.

Returns A hit detector

Return type *HitDetector*

bsb.voxels.**detect_box_compartments**(*tree*, *box_origin*, *box_size*)
Given a tree of compartment locations and a box, it will return the ids of all compartments in the outer sphere of the box

Parameters **box_origin** – The lowermost corner of the box.

9.3 Index

9.4 Module Index

CHAPTER

TEN

DEVELOPER GUIDES

10.1 Developer Installation

To install:

```
git clone git@github.com:dbbs-lab/bsb
cd bsb
pip install -e .[dev]
pre-commit install
```

Test your install with:

```
python -m unittest discover -s tests
```

10.2 Documentation

To build the documentation run:

```
cd docs
make html
```

10.2.1 Conventions

- Values are marked as `5` or `"hello"` using double backticks (`` ``).
- Configuration attributes are marked as *attribute* using the guilabel directive (`:guilabel:`attribute``)

PYTHON MODULE INDEX

b

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W