Brain Scaffold Builder

Release 4.0.0a4

Robin De Schepper

Feb 24, 2022

GETTING STARTED

The BSB is a framework for reconstructing and simulating multi-paradigm neuronal network models. It removes much of the repetitive work associated with writing the required code and lets you focus on the parts that matter. It helps write organized, well-parametrized and explicit code understandable and reusable by your peers.

Get started Get started with your first project!

Components Learn how to write your own components to e.g. place or connect cells.

Simulations Learn how to simulate your network models

Examples View examples explained step by step

Plugins Learn to package your code for others to use!

Contributing Help out the project by contributing code.

CHAPTER

ONE

INSTALLATION GUIDE

1.1 Preamble

Warning: Your mileage with the framework will vary based on your adherence to Python best practices.

1.1.1 Which Python to use?

Linux distributions come bundled with Python installations and many parts of the distro depend on these installations, making them hard to update and installing packages into the system-wide environment can have surprising side effects.

Instead to stay up to date with the newest Python releases use a tool like pyenv to manage different Python versions at the same time. Windows users can install a newer binary from the Python website. You're also most likely to make a big bloated mess out of these environments and will run into myriads of strange environment errors.

1.1.2 Why is everyone telling me to use a virtual env?

Python's package system is flawed, it can only install packages in a "global" fashion. You can't install multiple versions of the same package for different projects so eventually packages will start clashing with each other. On top of that scanning the installed packages for metadata, like plugins, becomes slower the more packages you have installed.

To fix these problems Python relies on "virtual environments". Use either `pyenv` (mentioned above), `venv` (part of Python's stdlib) or if you must `virtualenv` (package). Packages inside a virtual environment do not clash with packages from another environment and let you install your dependencies on a per project basis.

1.2 Instructions

The scaffold framework can be installed using `pip`:

```
pip install bsb>=4.0.0a0
```

You can verify that the installation works with

```
bsb -v=3 compile -x=100 -z=100 -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*.

Another verification method is to import the package in a Python script:

```
from bsb.core import Scaffold

# Create an empty scaffold network with the default configuration.
scaffold = Scaffold()
```

1.3 Simulator backends

If you'd like to install the scaffold builder for point neuron simulations with NEST or multicompartmental neuron simulations with NEURON use:

```
pip install bsb[nest]
# or
pip install bsb[neuron]
# or both
pip install bsb[nest,neuron]
```

Note: This does not install the simulators, just the Python requirements for the framework to handle simulations using these backends.

1.4 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 *storage* node in your config:

```
{
    "name": "Example config",
    "storage": {
        "engine": "hdf5",
        "root": "my_network.hdf5",
```

(continues on next page)

(continued from previous page)

```
        "morphology_repository": "morphologies.hdf5"
    }
}
```

1.5 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:

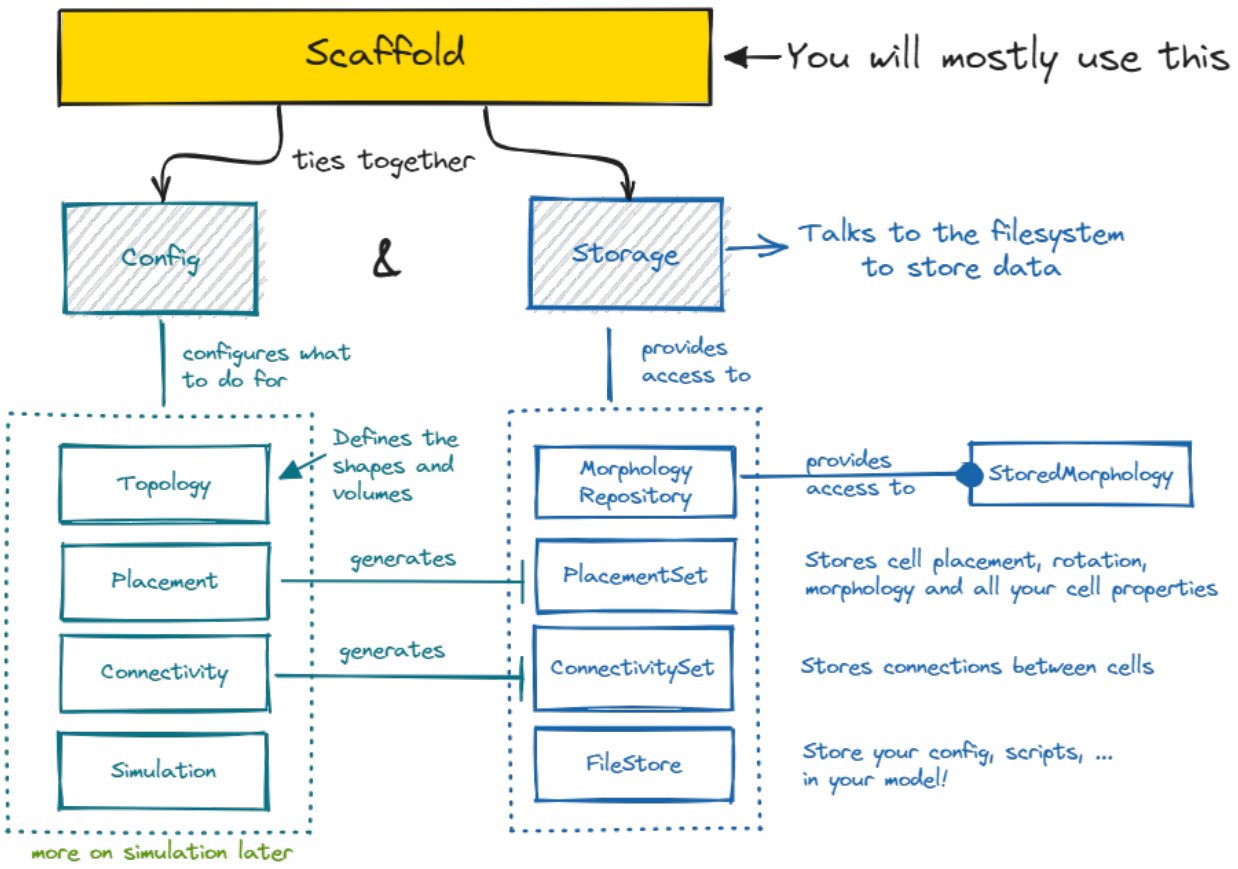
```
python -c "import nest; nest.test()"
```

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

CHAPTER

TWO

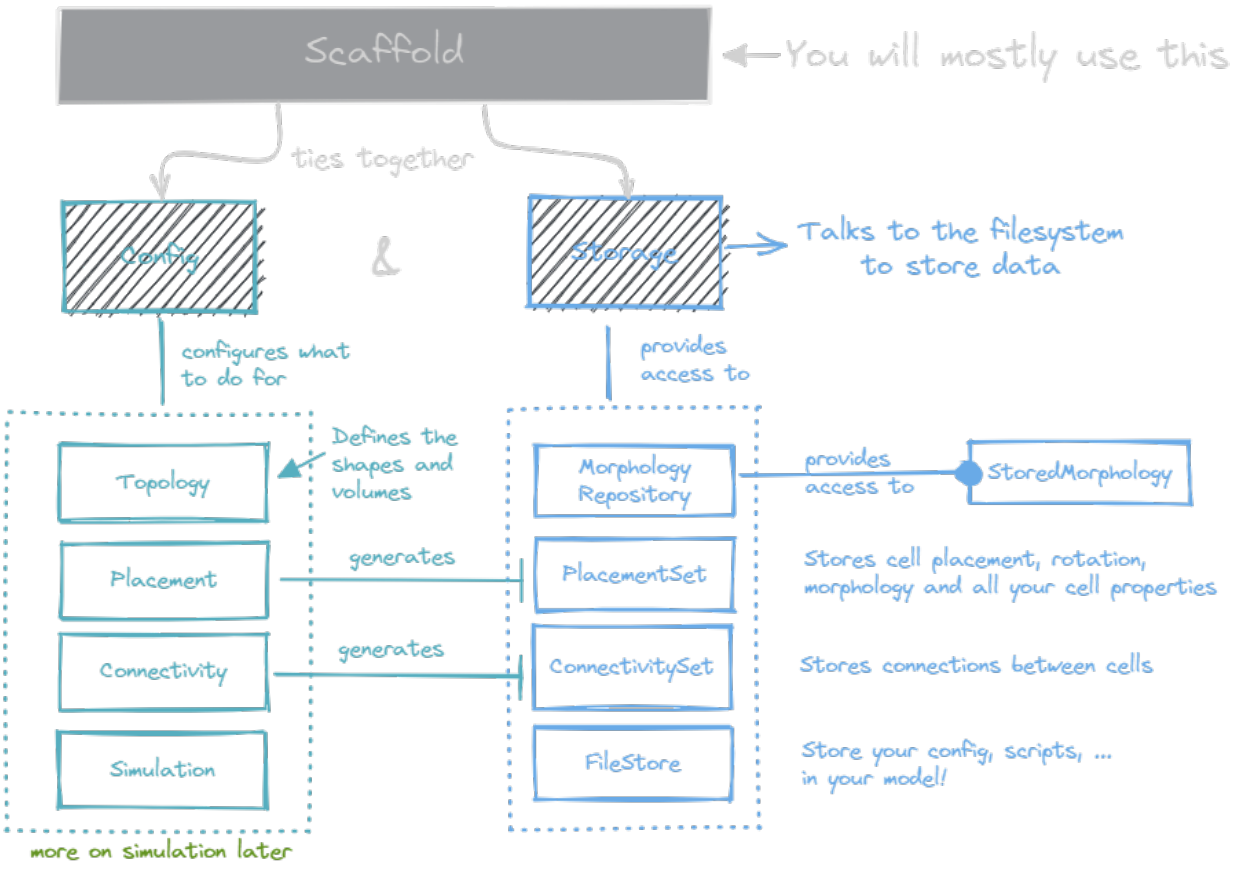
TOP LEVEL GUIDE



The Brain **Scaffold** Builder revolves around the *Scaffold* object. A scaffold ties together all the information in the *Configuration* with the *Storage*. The configuration contains your entire model description, while the storage contains your model data, like concrete cell positions or connections.

Using the scaffold object one can turn the abstract model configuration into a concrete storage object full of neuroscience. For it to do so, the configuration needs to describe which steps to take to place cells, called `Placement`, which steps to take to connect cells, called `Connectivity`, and what representations to use during `Simulation` for those cells and connections. All of these configurable objects can be accessed from the scaffold object. Placement under `scaffold.placement`, etc etc...

Also, using the scaffold object, you can inspect the data in the storage by using the *PlacementSet* and *ConnectivitySet* APIs. PlacementSets can be obtained with *scaffold.get_placement_set*, ConnectivitySets with *scaffold.get_connectivity_set* etc etc...

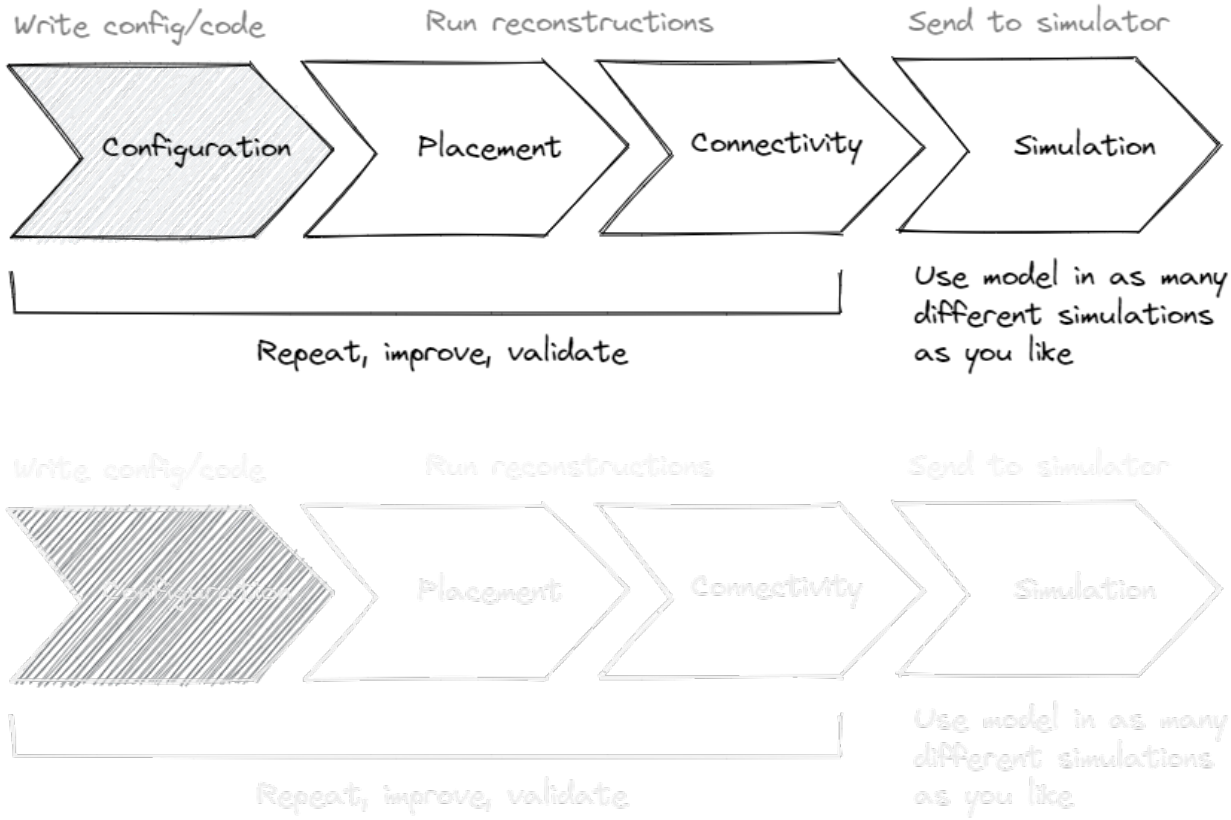

Scaffold
You will mostly use this
ties together
Config
&
Storage
Talks to the filesystem
to store data
configures what
to do for
provides
access to
Defines the
shapes and
volumes
Topology
Placement
Connectivity
Simulation
generates
generates
Morphology
Repository
provides
access to
StoredMorphology
PlacementSet
ConnectivitySet
FileStore
Stores cell placement, rotation,
morphology and all your cell properties
Stores connections between cells
Store your config, scripts, ...
in your model!
more on simulation later

The configuration object contains a structured tree of configurable objects, that in totality describe your network model. You can either fill out configuration file to be parsed, or write the objects yourself in Python. There are several parts of a configuration to be filled out, take a look at *config*.

The storage object provides access to an underlying engine that performs read and write operations in a certain data format. You can use the storage object to manipulate the data in your model, but usually it's better if the scaffold object is allowed to translate configuration directly into data, so that anyone can take a look at the config and know exactly what data is in storage, and how it got there!

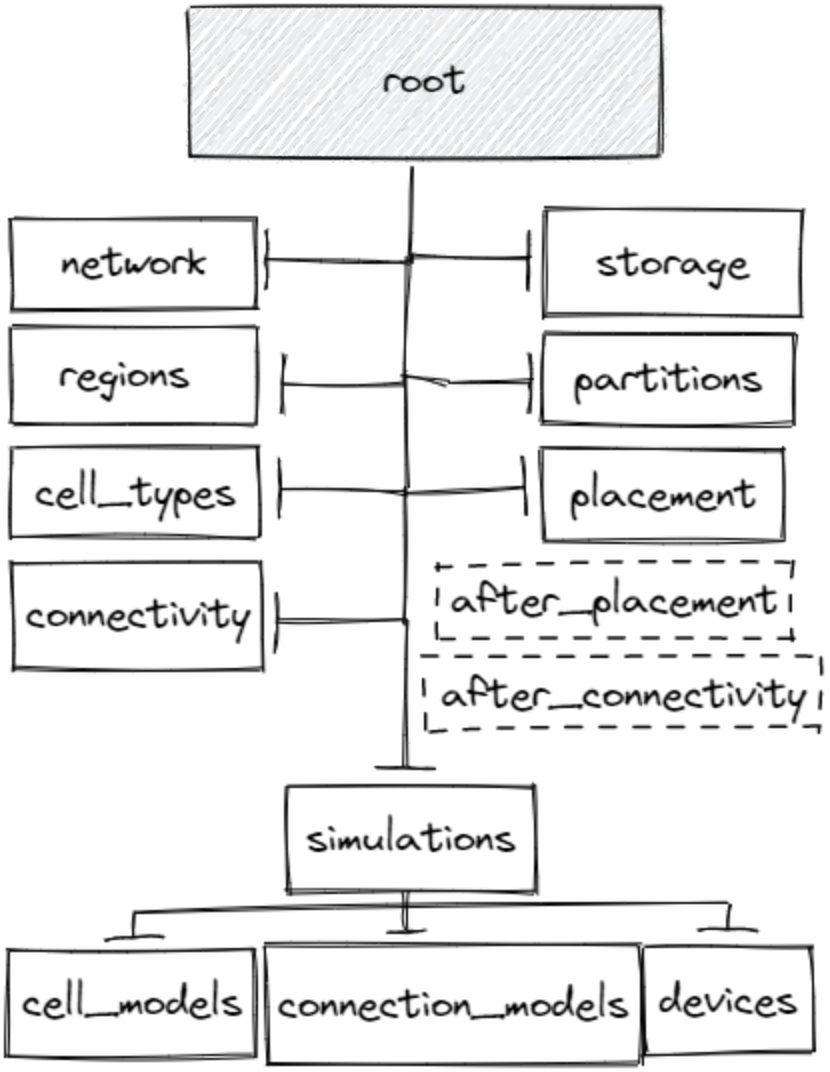
Ultimately this is the goal of the entire framework: To let you explicitly define every component that is a part of your model, and all its parameters, in such a way that a single CLI command, `bsb compile`, can turn your configuration into a reconstructed biophysically detailed large scale neural network, with all its parameters explicitly presented to any reader in a human readable configuration file.

2.1 Workflow

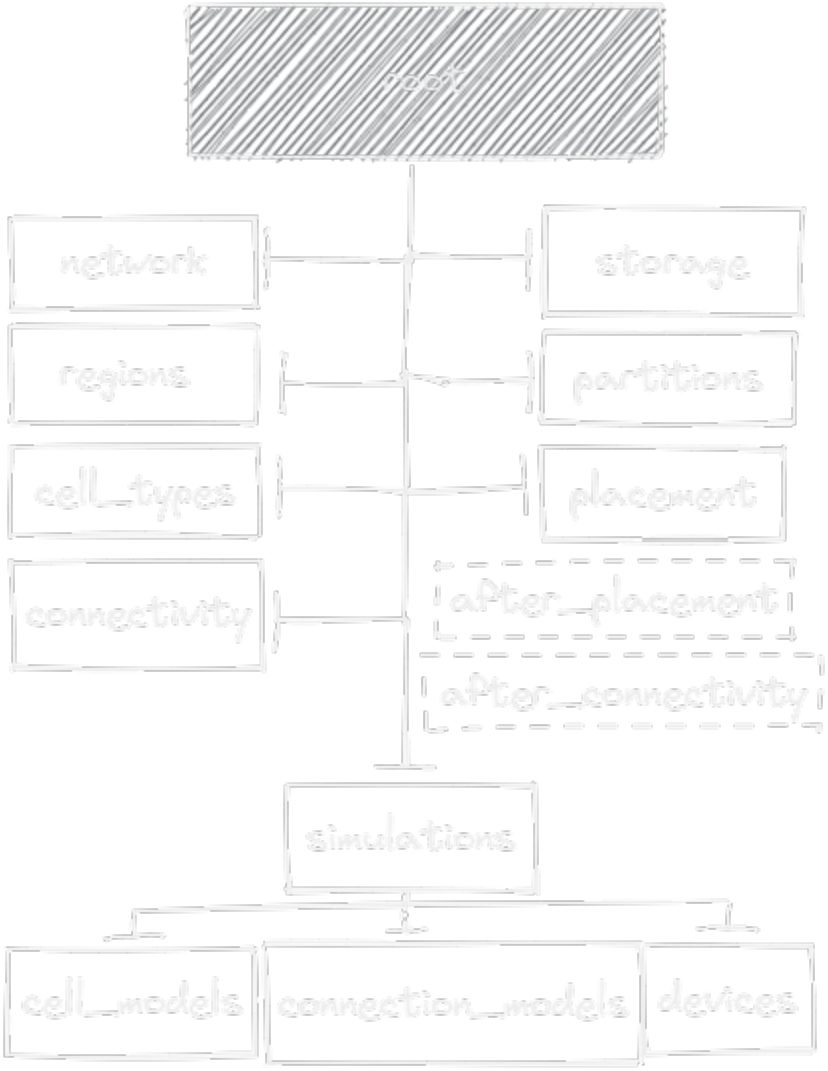


The framework promotes iterative improvements on the model. Start small, and incrementally add on every piece you need after validating the last!

2.2 Configuration



Configuration objects are trees with the above blocks defined (dashed = optional) and can be loaded from various formats, JSON by default, or created from code.


network
storage
regions
partitions
cell_types
placement
connectivity
after_placement
after_connectivity
simulations
cell_models
connection_models
devices

CHAPTER

THREE

GETTING STARTED

Follow the *Installation Guide*:

- Set up a new environment
- Install the software into the environment

Note: This guide aims to get your first model running with the bare minimum steps. If you'd like to familiarize yourself with the core concepts and get a more top level understanding first, check out the *Top Level Guide* before you continue.

There are 2 ways of building models using the Brain Scaffold Builder (BSB), the first is through **configuration**, the second is **scripting**. The 2 methods complement each other so that you can load the general model from a configuration file and then layer on more complex steps under your full control in a Python script. Be sure to take a quick look at each code tab to see the equivalent forms of configuration coding!

3.1 Create a project

Use the command below to create a new project directory and some starter files:

```
> bsb new my_first_model
      Config template [skeleton.json]: starting_example.json
      Config file [network_configuration.json]:
      > cd my_first_model
```

You'll be asked some questions; enter appropriate values, and be sure to select the `starting_example.json` as the template configuration file, and to navigate your terminal into the new folder.

The project now contains a couple of important files:

- A configuration file: your components are declared and parametrized here.
- A `pyproject.toml` file: your project settings are declared here.
- A `placement.py` and `connectome.py` file to put your code in.

Take a look at `starting_example.json`; it contains a nondescript `brain_region`, a `base_layer`, a `base_type` and an `example_placement`. These minimal components are enough to *compile* your first network. You can do this from the CLI or Python:

BASH

```
bsb compile --verbosity 3 --plot
```

PYTHON

```
from bsb.core import Scaffold
from bsb.config import from_json
from bsb.plotting import plot_network
import bsb.options

bsb.options.verbosity = 3
config = from_json("starting_example.json")
scaffold = Scaffold(config)
scaffold.compile()
plot_network(scaffold)
```

The `verbosity` helps you follow along what instructions the framework is executing and `plot` should.. open a plot .

3.2 Define starter components

3.2.1 Topology

Your network model needs a description of its shape, which is called the topology of the network. The topology exists of 2 types of components: *Regions* and *Partitions*. Regions combine multiple partitions and/or regions together, in a hierarchy, all the way up to a single topmost region, while partitions are exact pieces of volume that can be filled with cells.

To get started, we'll add a `cortex` region, and populate it with a `base_layer`:

```
{
  "regions": {
    "cortex": {
      "origin": [0.0, 0.0, 0.0],
                                "partitions": ["base_layer"]
    }
  },
  "partitions": {
    "base_layer": {
                                "type": "layer",
      "thickness": 100
    }
  }
}
```

The `cortex` does not specify a region *type*, so it is a group. The *type* of `base_layer` is `layer`, they specify their size in 1 dimension, and fill up the space in the other dimensions. See *Introduction* for more explanation on topology components.

3.2.2 Cell types

The *CellType* is a definition of a cell population. During placement 3D positions, optionally rotations and morphologies or other properties will be created for them. In the simplest case you define a soma *radius* and *density* or fixed *count*:

```
{
  "cell_types": {
    "cell_type_A": {
      "spatial": {
        "radius": 7,
                                        "density": 1e-3
      }
    },
    "cell_type_B": {
      "spatial": {
        "radius": 7,
                                        "count": 10
      }
    }
  }
}
```

3.2.3 Placement

```
{
        "placement": {
                "cls": "bsb.placement.ParticlePlacement",
                "cell_types": ["cell_type_A", "cell_type_B"],
                "partitions": ["base_layer"]
        }
}
```

The `placement` blocks use the cell type indications to place cell types into partitions. You can use *PlacementStrategies* provided out of the box by the BSB or your own component by setting the *cls*. The *ParticlePlacement* considers the cells as somas and bumps them around as repelling particles until there is no overlap between the somas. The data is stored in *PlacementSets* per cell type.

Take another look at your network:

```
bsb compile -v 3 -p
```

Note: We're using the short forms `-v` and `-p` of the CLI options `--verbosity` and `--plot`, respectively. You can use `bsb --help` to inspect the CLI options.

3.2.4 Connectivity

```
{
              "connectivity": {
    "A_to_B": {
      "cls": "bsb.connectivity.AllToAll",
      "pre": {
        "cell_types": ["cell_type_A"]
      },
      "post": {
          "cell_types": ["cell_type_B"]
      }
    }
  }
}
```

The `connectivity` blocks specify connections between systems of cell types. They can create connections between single or multiple pre and postsynaptic cell types, and can produce one or many *ConnectivitySets*.

Regenerate the network once more, now it will also contain your connections! With your cells and connections in place, you're ready to move to the *Simulations* stage.

What next?

Continue getting started Follow the rest of the guides for basics on as `CellTypes`, `Placement` blocks, `Connectivity` blocks and `Simulations`.

Components Learn how to write your own components to e.g. place or connect cells.

Simulations Learn how to simulate your network models

Examples View examples explained step by step

Plugins Learn to package your code for others to use!

Contributing Help out the project by contributing code.

CHAPTER

FOUR

COMMAND LINE INTERFACE

4.1 List of command line commands

Note: Parameters included between angle brackets are example values, parameters between square brackets are optional, leave off the brackets in the actual command.

Every command starts with: `bsb [OPTIONS]`, where `[OPTIONS]` can be any combination of *BSB options*.

4.1.1 Creating a project

`bsb [OPTIONS] new <project-name> <parent-folder>`

Creates a new project directory at `folder`. You will be prompted to fill in some project settings.

- `project-name`: Name of the project, and of the directory that will be created for it.
- `parent-folder`: Filesystem location where the project folder will be created.

4.1.2 Creating a configuration

`bsb [OPTIONS] make-config <template.json> <output.json> [--path <path1> <path2 ...>]`

Create a configuration in the current directory, based off the template. Specify additional paths to search extra locations, if the configuration isn't a registered template.

- `template.json`: Filename of the template to look for. Templates can be registered through the `bsb.config.templates` *plugin endpoint*. Does not need to be a json file, just a file that can be parsed by your installed parsers.
- `output.json`: Filename to be created.
- `--path`: Give additional paths to be searched for the template here.

4.1.3 Compiling a network

`bsb [OPTIONS] compile [my-config.json] [-p] [-o file]`

Compiles a network architecture according to the configuration. If no configuration is specified, the project default is used.

- `my-config.json`: Path to the configuration file that should be compiled. If omitted the *project configuration* path is used.
- `-p`: Plot the created network.
- `-o`, `--output`: Output the result to a specific file. If omitted the value from the configuration, the project default, or a timestamped filename are used.

4.1.4 Running a simulation

`bsb [OPTIONS] simulate <path/to/netw.hdf5> <sim-name>`

Run a simulation from a compiled network architecture.

- `path/to/netw.hdf5`: Path to the network file.
- `sim-name`: Name of the simulation.

4.1.5 Checking the global cache

`bsb [OPTIONS] cache [--clear]`

Check which files are currently cached, and optionally clear them.

CHAPTER

FIVE

SIMULATIONS

```
{
  "simulations": {
    "nrn_example": {
      "simulator": "neuron",
      "temperature": 32,
      "resolution": 0.1,
      "duration": 1000,
      "cell_models": {

      },
      "connection_models": {

      },
      "devices": {

      }
    },
    "nest_example": {
      "simulator": "nest",
      "default_neuron_model": "iaf_cond_alpha",
      "default_synapse_model": "static_synapse",
      "duration": 1000.0,
      "modules": ["my_extension_module"],
      "cell_models": {

      }
    }
  }
}
```

The definition of simulations begins with chosing a simulator, either `nest`, `neuron` or `arbor`. Each simulator has their adapter and each adapter its own requirements, see *Simulation adapters*. All of them share the commonality that they configure `cell_models`, `connection_models` and `devices`.

5.1 Defining cell models

A cell model is used to describe a member of a cell type during a simulation.

5.1.1 NEURON

A cell model is described by loading external `arborize.CellModel` classes:

```
{
  "cell_models": {
    "cell_type_A": {
      "model": "dbbs_models.GranuleCell",
      "record_soma": true,
      "record_spikes": true
    },
    "cell_type_B": {
      "model": "dbbs_models.PurkinjeCell",
      "record_soma": true,
      "record_spikes": true
    }
  }
}
```

This example dictates that during simulation setup, any member of `cell_type_A` should be created by importing and using `dbbs_models.GranuleCell`. Documentation incomplete, see `arborize` docs ad interim.

5.1.2 NEST

In NEST the cell models need to correspond to the available models in NEST and parameters can be given:

```
{
  "cell_models": {
    "cell_type_A": {
      "neuron_model": "iaf_cond_alpha",
      "parameters": {
        "t_ref": 1.5,
        "C_m": 7.0,
        "V_th": -41.0,
        "V_reset": -70.0,
        "E_L": -62.0,
        "I_e": 0.0,
        "tau_syn_ex": 5.8,
        "tau_syn_in": 13.61,
        "g_L": 0.29
      }
    },
    "cell_type_B": {
      "neuron_model": "iaf_cond_alpha",
      "parameters": {
        "t_ref": 1.5,
        "C_m": 7.0,
```

(continues on next page)

(continued from previous page)

```
        "V_th": -41.0,
        "V_reset": -70.0,
        "E_L": -62.0,
        "I_e": 0.0,
        "tau_syn_ex": 5.8,
        "tau_syn_in": 13.61,
        "g_L": 0.29
      }
    }
  }
}
```

5.2 Defining connection models

Connection models represent the connections between cells during a simulation.

5.2.1 NEURON

Once more the connection models are predefined inside of `arborize` and they can be referenced by name:

```
{
  "connection_models": {
    "A_to_B": {
      "synapses": ["AMPA", "NMDA"]
    }
  }
}
```

5.2.2 NEST

Connection models need to match the available connection models in NEST:

```
{
  "connection_models": {
    "A_to_B": {
      "synapse_model": "static_synapse",
      "connection": {
        "weight":-0.3,
        "delay": 5.0
      },
      "synapse": {
        "static_synapse": {}
      }
    }
  }
}
```

5.3 Defining devices

5.3.1 NEURON

In NEURON an assortment of devices is provided by the BSB to send input, or record output. See *List of NEURON devices* for a complete list. Some devices like voltage and spike recorders can be placed by requesting them on cell models using *record_soma* or *record_spikes*.

In addition to voltage and spike recording we'll place a spike generator and a voltage clamp:

```
{
  "devices": {
    "stimulus": {
      "io": "input",
      "device": "spike_generator",
      "targetting": "cell_type",
      "cell_types": ["cell_type_A"],
      "synapses": ["AMPA"],
      "start": 500,
      "number": 10,
      "interval": 10,
      "noise": true
    },
    "voltage_clamp": {
      "io": "input",
      "device": "voltage_clamp",
      "targetting": "cell_type",
      "cell_types": ["cell_type_B"],
      "cell_count": 1,
      "section_types": ["soma"],
      "section_count": 1,
      "parameters": {
        "delay": 0,
        "duration": 1000,
        "after": 0,
        "voltage": -63
      }
    }
  }
}
```

The voltage clamp targets 1 random `cell_type_B` which is a bit awkward, but either the `targetting` (docs incomplete) or the `labelling` system (docs incomplete) can help you target exactly the right cells.

5.4 Running a simulation

Simulations can be run through the CLI tool, or for more control through the `bsb` library. When using the CLI, the framework sets up a "hands off" simulation:

- Read the network file
- Read the simulation configuration
- Translate the simulation configuration to the simulator
- Create all cells, connections and devices
- Run the simulation
- Collect all the output

```
bsb simulate my_network.hdf5 my_sim_name
```

When you use the library, you can set up more complex workflows, for example, this is a parameter sweep that loops and modifies the release probability of the AMPA synapse in the cerebellar granule cell:

```
from bsb.core import from_hdf5

# A module with cerebellar cell models
import dbbs_models

# A module to run NEURON simulations in isolation
import nrnsub

# A module to read HDF5 data
import h5py

# Read the network file
network = from_hdf("my_network.hdf5")


@nrnsub.isolate
def sweep(param):
    # Get an adapter to the simulation
    adapter = network.create_adapter("my_sim_name")
    # Modify the parameter to sweep
    dbbs_models.GranuleCell.synapses["AMPA"]["U"] = param
    # Prepare simulator & instantiate all the cells and connections
    simulation = adapter.prepare()

    # (Optionally perform more custom operations before the simulation here.)

    # Run the simulation
    adapter.simulate(simulation)

    # (Optionally perform more operations or even additional simulation steps here.)

    # Collect all results in an HDF5 file and get the path to it.
    result_file = adapter.collect_output()
    return result_file
```

(continues on next page)

(continued from previous page)

```
for i in range(11):
    # Sweep parameter from 0 to 1 in 0.1 increments
    result_file = sweep(i / 10)

    # Analyze each run's results here
    with h5py.File(result_file, "r") as results:
        print("What did I record?", list(results["recorders"].keys()))
```

5.5 Parallel simulations

To parallelize any task the BSB can execute you can prepend the MPI command in front of the BSB CLI command, or the Python script command:

```
mpirun -n 4 bsb simulate my_network.hdf5 your_simulation
mpirun -n 4 python my_simulation_script.py
```

Where `n` is the number of parallel nodes you'd like to use.

CHAPTER

SIX

INTRODUCTION

A configuration file describes a scaffold model. It contains the instructions to place and connect neurons, how to represent the cells and connections as models in simulators and what to stimulate and record in simulations.

The default configuration format is JSON and a standard configuration file might look like this:

```
{
  "storage": {

  },
  "network": {

  },
  "regions": {

  },
  "partitions": {

  },
  "cell_types": {

  },
  "placement": {

  },
  "after_placement": {

  },
  "connectivity": {

  },
  "after_connectivity": {

  },
  "simulations": {

  }
}
```

The *regions*, *partitions*, *cell_types*, *placement* and *connectivity* spaceholders hold the configuration for *Regions*, *Partitions*, *CellTypes*, *PlacementStrategies* and *ConnectionStrategies* respectively.

When you're configuring a model you'll mostly be using **configuration attributes**, **configuration nodes/dictionaries**

and **configuration lists**. These basic concepts and their JSON expressions are explained in *Configuration units*.

The main goal of the configuration file is to provide data to Python classes that execute certain tasks such as placing cells, connecting them or simulating them. In order to link your Python classes to the configuration file they should be **importable**. The Python documentation explains what modules are and are a great starting point.

In short, `my_file.py` is importable as `my_file` when it is in the working directory or on the path Python searches. Any classes inside of it can be referenced in a config file as `my_file.MyClass`. Although this basic use works fine for a single directory, we have a *best practices guide* on how to properly make your classes discoverable on your entire machine. You can even distribute them as a package to other people the same way.

Here's an example of how you could use the `MySpecialConnection` class in your Python file `connectome.py` as a class in the configuration:

```
{
  "storage": {
    "engine": "hdf5",
    "root": "my_network.hdf5"
  },
  "network": {
    "x": 200,
    "z": 200
  },
  "regions": {

  },
  "partitions": {
  },
  "cell_types": {

  },
  "connectivity": {
    "A_to_B": {
      "cls": "connectome.MySpecialConnection",
      "value1": 15,
      "thingy2": [4, 13]
    }
  }
}
```

Any extra configuration data (such as `value1` and `thingy2`) is automatically passed to it!

For more information on creating your own configuration nodes see *Nodes*.

6.1 JSON Parser

The BSB uses a json parser with some extras. The parser has 2 special mechanisms, JSON references and JSON imports. This allows parts of the configuration file to be reusable across documents and to compose the document from prefab blocks where only some key aspects are adjusted. For example, an entire simulation protocol could be imported and the start and stop time of a stimulus adjusted:

```
{
  "simulations": {
    "premade_sim": {
```

(continues on next page)

(continued from previous page)

```
      "$ref": "premade_simulations.json#/simulations/twin_pulse",
      "devices": {
        "pulse1": {
          "start": 100,
          "stop": 200
        }
      }
    }
  }
}
```

This would import `/simulations/twin_pulse` from the `premade_simulations.json` JSON document and overwrite the `start` and `stop` time of the `pulse1` device.

See *BSB JSON parser* to read more on the JSON parser.

6.1.1 Default configuration

You can create a default configuration by calling *`Configuration.default`*. It corresponds to the following JSON:

<<<insert default>>>

CHAPTER

SEVEN

CONFIG MODULE

7.1 Overview

7.1.1 Role in the scaffold

Configuration plays a key role in the scaffold builder. It is the main mechanism to describe a model. A scaffold model can be initialized from a Configuration object, either from a standalone file or provided by the *Storage*. In both cases the raw configuration string is parsed into a Python tree of dictionaries and lists. This configuration tree is then passed to the Configuration class for *casting*. How a tree is to be cast into a Configuration object can be described using configuration unit syntax.

7.1.2 Configuration units

When the configuration tree is being cast into a Configuration object there are 5 key units:

- A **configuration attribute** represented by a key-value pair.
- A **configuration reference** points to another location in the configuration.
- A **configuration node** represented by a dictionary.
- A **configuration dictionary** represented by a dictionary where each key-value pair represents another configuration unit.
- A **configuration list** represented by a list where each value represents another configuration unit.

Note: If a list or dictionary contains regular values instead of other configuration units, the *types.list* and *types.dict* are used instead of the *config.list* and *config.dict*.

Configuration nodes

A node in the configuration can be described by creating a class and applying the `@config.node` decorator to it. This decorator will look for `config.attr` and other configuration unit constructors on the class to create the configuration information on the class. This node class can then be used in the type argument of another configuration attribute, dictionary, or list:

```
from bsb import config

@config.node
class CandyNode:
```

(continues on next page)

(continued from previous page)

```
  name = config.attr(type=str, required=True)
  sweetness = config.attr(type=float, default=3.0)
```

This candy node class now represents the following JSON dictionary:

```
{
  "name": "Lollypop",
  "sweetness": 12.0
}
```

You will mainly design configuration nodes and other configuration logic when designing custom strategies.

Dynamic nodes

An important part to the interfacing system of the scaffold builder are custom strategies. Any user can implement a simple functional interface such as the *PlacementStrategy* to design a new way of placing cells. Placement configuration nodes can then use these strategies by specifying the *cls* attribute:

```
{
  "my_cell_type": {
    "placement": {
      "cls": "my_package.MyStrategy"
    }
  }
}
```

This dynamic loading is achieved by creating a node class with the `@config.dynamic` decorator instead of the node decorator. This will add a configuration attribute `cls` to the node class and use the value of this class to create an instance of another node class, provided that the latter inherits from the former, enforcing the interface.

```
@config.dynamic
class PlacementStrategy:
  @abc.abstractmethod
  def place(self):
    pass
```

Configuration attributes

An attribute can refer to a singular value of a certain type, or to another node:

```
from bsb import config

@config.node
class CandyStack:
  count = config.attr(type=int, required=True)
  candy = config.attr(type=CandyNode)
```

```
{
  "count": 12,
  "candy": {
```

(continues on next page)

(continued from previous page)

```
    "name": "Hardcandy",
    "sweetness": 4.5
  }
}
```

Configuration dictionaries

Configuration dictionaries hold configuration nodes. If you need a dictionary of values use the *types.dict* syntax instead.

```
from bsb import config

@config.node
class CandyNode:
  name = config.attr(key=True)
  sweetness = config.attr(type=float, default=3.0)

@config.node
class Inventory:
  candies = config.dict(type=CandyStack)
```

```
{
  "candies": {
    "Lollypop": {
      "sweetness": 12.0
    },
    "Hardcandy": {
      "sweetness": 4.5
    }
  }
}
```

Items in configuration dictionaries can be accessed using dot notation or indexing:

```
inventory.candies.Lollypop == inventory.candies["Lollypop"]
```

Using the `key` keyword argument on a configuration attribute will pass the key in the dictionary to the attribute so that `inventory.candies.Lollypop.name == "Lollypop"`.

Configuration lists

Configuration dictionaries hold unnamed collections of configuration nodes. If you need a list of values use the *types.list* syntax instead.

```
from bsb import config

@config.node
class InventoryList:
  candies = config.list(type=CandyStack)
```

```
{
  "candies": [
    {
      "count": 100,
      "candy": {
        "name": "Lollypop",
        "sweetness": 12.0
      }
    },
    {
      "count": 1200,
      "candy": {
        "name": "Hardcandy",
        "sweetness": 4.5
      }
    }
  ]
}
```

Configuration references

References refer to other locations in the configuration. In the configuration the configured string will be fetched from the referenced node:

```
{
  "locations": {"A": "very close", "B": "very far"},
  "where": "A"
}
```

Assuming that `where` is a reference to `locations`, location `A` will be retrieved and placed under `where` so that in the config object:

```
>>> print(conf.locations)
{'A': 'very close', 'B': 'very far'}

>>> print(conf.where)
'very close'

>>> print(conf.where_reference)
'A'
```

References are defined inside of configuration nodes by passing a reference object to the *config.ref()* function:

```
@config.node
class Locations:
  locations = config.dict(type=str)
  where = config.ref(lambda root, here: here["locations"])
```

After the configuration has been cast all nodes are visited to check if they are a reference and if so the value from elsewhere in the configuration is retrieved. The original string from the configuration is also stored in `node.<ref>_reference`.

After the configuration is loaded it's possible to either give a new reference key (usually a string) or a new reference value. In most cases the configuration will automatically detect what you're passing into the reference:

```
>>> cfg = from_json("mouse_cerebellum.json")
>>> cfg.cell_types.granule_cell.placement.layer.name
'granular_layer'
>>> cfg.cell_types.granule_cell.placement.layer = 'molecular_layer'
>>> cfg.cell_types.granule_cell.placement.layer.name
'molecular_layer'
>>> cfg.cell_types.granule_cell.placement.layer = cfg.layers.purkinje_layer
>>> cfg.cell_types.granule_cell.placement.layer.name
'purkinje_layer'
```

As you can see, by passing the reference a string the object is fetched from the reference location, but we can also directly pass the object the reference string would point to. This behavior is controlled by the `ref_type` keyword argument on the `config.ref` call and the `is_ref` method on the reference object. If neither is given it defaults to checking whether the value is an instance of `str`:

```
@config.node
class CandySelect:
  candies = config.dict(type=Candy)
  special_candy = config.ref(lambda root, here: here.candies, ref_type=Candy)

class CandyReference(config.refs.Reference):
  def __call__(self, root, here):
    return here.candies

  def is_ref(self, value):
    return isinstance(value, Candy)

@config.node
class CandySelect:
  candies = config.dict(type=Candy)
  special_candy = config.ref(CandyReference())
```

The above code will make sure that only `Candy` objects are seen as references and all other types are seen as keys that need to be looked up. It is recommended you do this even in trivial cases to prevent bugs.

Reference object

The reference object is a callable object that takes 2 arguments: the configuration root node and the referring node. Using these 2 locations it should return a configuration node from which the reference value can be retrieved.

```
def locations_reference(root, here):
  return root.locations
```

This reference object would create the link seen in the first reference example.

Reference lists

Reference lists are akin to references but instead of a single key they are a list of reference keys:

```
{
  "locations": {"A": "very close", "B": "very far"},
  "where": ["A", "B"]
}
```

Results in `cfg.where == ["very close", "very far"]`. As with references you can set a new list and all items will either be looked up or kept as is if they're a reference value already.

Warning: Appending elements to these lists currently does not convert the new value. Also note that reference lists are quite indestructible; setting them to *None* just resets them and the reference key list (`.<attr>_references`) to `[]`.

Bidirectional references

The object that a reference points to can be "notified" that it is being referenced by the `populate` mechanism. This mechanism stores the referrer on the referee creating a bidirectional reference. If the `populate` argument is given to the `config.ref` call the referrer will append itself to the list on the referee under the attribute given by the value of the `populate` kwarg (or create a new list if it doesn't exist).

```
{
  "containers": {
    "A": {}
  },
  "elements": {
    "a": {"container": "A"}
  }
}
```

```
@config.node
class Container:
  name = config.attr(key=True)
  elements = config.attr(type=list, default=list, call_default=True)

@config.node
class Element:
  container = config.ref(container_ref, populate="elements")
```

This would result in `cfg.containers.A.elements == [cfg.elements.a]`.

You can overwrite the default *append or create* population behavior by creating a descriptor for the population attribute and define a `__populate__` method on it:

```
class PopulationAttribute:
  # Standard property-like descriptor protocol
  def __get__(self, instance, objtype=None):
    if instance is None:
      return self
```

(continues on next page)

(continued from previous page)

```
    if not hasattr(instance, "_population"):
      instance._population = []
    return instance._population

  # Prevent population from being overwritten
  # Merge with new values into a unique list instead
  def __set__(self, instance, value):
    instance._population = list(set(instance._population) + set(value))

  # Example that only stores referrers if their name in the configuration is "square".
  def __populate__(self, instance, value):
    print("We're referenced in", value.get_node_name())
    if value.get_node_name().endswith("square"):
      self.__set__(instance, [value])
    else:
      print("We only store referrers coming from a .square configuration attribute")
```

todo: Mention `pop_unique`

7.1.3 Casting

When the Configuration object is loaded it is cast from a tree to an object. This happens recursively starting at a configuration root. The default *Configuration* root is defined in `scaffold/config/_config.py` and describes how the scaffold builder will read a configuration tree.

You can cast from configuration trees to configuration nodes yourself by using the class method `__cast__`:

```
inventory = {
  "candies": {
    "Lollypop": {
      "sweetness": 12.0
    },
    "Hardcandy": {
      "sweetness": 4.5
    }
  }
}

# The second argument would be the node's parent if it had any.
conf = Inventory.__cast__(inventory, None)
print(conf.candies.Lollypop.sweetness)
>>> 12.0
```

Casting from a root node also resolves references.

7.2 Nodes

Nodes are the recursive backbone backbone of the Configuration object. Nodes can contain other nodes under their attributes and in that way recurse deeper into the configuration. Nodes can also be used as types of configuration dictionaries or lists.

Node classes contain the description of a node type in the configuration. Here's an example to illustrate:

```
from bsb import config

@config.node
class CellType:
  name = config.attr(key=True)
  color = config.attr()
  radius = config.attr(type=float, required=True)
```

This node class describes the following configuration:

```
{
  "cell_type_name": {
    "radius": 13.0,
    "color": "red"
  }
}
```

The `@config.node` decorator takes the ordinary class and injects the logic it needs to fulfill the tasks of a configuration node. Whenever a node of this type is used in the configuration an instance of the node class is created and some work needs to happen:

- The parsed configuration dictionary needs to be cast into an instance of the node class.
- The configuration attributes of this node class and its parents need to be collected.
- The attributes on this instance need to be initialized with a default value or `None`.
- The keys that are present in the configuration dictionary need to be transferred to the node instance and converted to the specified type (the default type is `str`)

7.2.1 Dynamic nodes

Dynamic nodes are those whose node class is configurable from inside the configuration node itself. This is done through the use of the `@dynamic` decorator instead of the node decorator. This will automatically create a required `class` attribute.

The value that is given to this class attribute will be used to import a class to instantiate the node:

```
@config.dynamic
class PlacementStrategy:
  @abc.abstractmethod
  def place(self):
    pass
```

And in the configuration:

```
{
  "cls": "bsb.placement.LayeredRandomWalk"
}
```

This would import the `bsb.placement` module and use its `LayeredRandomWalk` class to decorate the node.

Note: The child class must inherit from the dynamic node class.

Configuring the dynamic attribute

Additional keyword arguments can be passed to the *dynamic* decorator to specify the properties of the dynamic attribute. All keyword args are passed to the *attr* decorator to create the attribute on the class that specifies the dynamics.

- `attr_name`, `required` and `default`:

```
@config.dynamic(attr_name="example_type", required=False, default="Example")
class Example:
  pass

@config.node
class Explicit(Example):
  pass
```

`Example` can then be defined as either:

```
{
  "example_type": "Explicit"
}
```

or use the default `Example` implicitly by omitting the dynamic attribute:

```
{

}
```

Class maps

A preset map of shorter entries can be given to be mapped to an absolute or relative class path, or a class object:

```
@dynamic(classmap={"short": "pkg.with.a.long.name.DynClass"})
class Example:
    pass
```

If `short` is used the dynamic class will resolve to `pkg.with.a.long.name.DynClass`.

Automatic class maps

Automatic class maps can be generated by setting the `auto_classmap` keyword argument. Child classes can then register themselves in the classmap of the parent by providing the `classmap_entry` keyword argument in their class definition argument list.

```
@dynamic(auto_classmap=True)
class Example:
  pass

class MappedChild(Example, classmap_entry="short"):
  pass
```

This will generate a mapping from `short` to the `my.module.path.MappedChild` class.

If the base class is not supposed to be abstract, it can be added to the classmap as well:

```
@dynamic(auto_classmap=True, classmap_entry="self")
class Example:
  pass

class MappedChild(Example, classmap_entry="short"):
  pass
```

7.2.2 Root node

The root node is the Configuration object and is at the basis of the tree of nodes.

7.2.3 Pluggable nodes

A part of your configuration file might be using plugins, these plugins can behave quite different from eachother and forcing them all to use the same configuration might hinder their function or cause friction for users to configure them properly. To solve this parts of the configuration are *pluggable*. This means that what needs to be configured in the node can be determined by the plugin that you select for it. Homogeneity can be enforced by defining *slots*. If a slot attribute is defined inside of a then the plugin must provide an attribute with the same name.

Note: Currently the provided attribute slots enforce just the presence, not any kind of inheritance or deeper inspection. It's up to a plugin author to understand the purpose of the slot and to comply with its intentions.

Consider the following example:

```
import bsb.plugins, bsb.config

@bsb.config.pluggable(key="plugin", plugin_name="puppy generator")
class PluginNode:
  @classmethod
  def __plugins__(cls):
      if not hasattr(cls, "_plugins"):
          cls._plugins = bsb.plugins.discover("puppy_generators")
      return cls._plugins
```

```
{
  "plugin": "labradoodle",
  "labrador_percentage": 110,
  "poodle_percentage": 60
}
```

The decorator argument `key` determines which attribute will be read to find out which plugin the user wants to configure. The class method `__plugins__` will be used to fetch the plugins every time a plugin is configured (usually finding these plugins isn't that fast so caching them is recommended). The returned plugin objects should be configuration node classes. These classes will then be used to further handle the given configuration.

7.3 Configuration types

Configuration types convert given configuration values. Values incompatible with the given type are rejected and the user is warned. This makes typing the most immediate form of validation that a configuration unit can declare. All configuration attributes, dictionaries and lists have types that they are converted to. The default type is `str`.

Any callable that takes 1 argument can be used as a type handler. The *config.types* module provides extra functionality such as validation of list and dictionaries and even more complex combinations of types. Every configuration node itself can be used as a type aswell.

Warning: All of the members of the *config.types* module are factory methods: they need to be **called** in order to produce the type handler. Using `config.attr(type=types.any)` is incorrect and will lead to cryptic or silent errors, use `config.attr(type=types.any())` instead.

7.3.1 Examples

```
from bsb import config
from bsb.config import types

@config.node
class TestNode
  name = config.attr()

@config.node
class TypeNode
  # Default string
  some_string = config.attr()
  # Explicit & required string
  required_string = config.attr(type=str, required=True)
  # Float
  some_number = config.attr(type=float)
              # types.float / types.int
  bounded_float = config.attr(type=types.float(min=0.3, max=17.9))
  # Float, int or bool (attempted to cast in that order)
  combined = config.attr(type=types.or_(float, int, bool))
  # Another node
  my_node = config.attr(type=TestNode)
  # A list of floats
```

(continues on next page)

(continued from previous page)

```
list_of_numbers = config.attr(
  type=types.list(type=float)
)
# 3 floats
list_of_numbers = config.attr(
  type=types.list(type=float, size=3)
)
            # A scipy.stats distribution
            chi_distr = config.attr(type=types.distribution())
            # A python statement evaluation
            statement = config.attr(type=types.evaluation())
# Create an np.ndarray with 3 elements out of a scalar
expand = config.attr(
    type=types.scalar_expand(
        scalar_type=int,
        expand=lambda s: np.ones(3) * s
    )
)
# Create np.zeros of given shape
zeros = config.attr(
    type=types.scalar_expand(
        scalar_type=types.list(type=int),
        expand=lambda s: np.zeros(s)
    )
)
# Anything
any = config.attr(type=types.any())
# One of the following strings: "all", "some", "none"
give_me = config.attr(type=types.in_(["all", "some", "none"]))
# The answer to life, the universe, and everything else
answer = config.attr(type=lambda x: 42)
# You're either having cake or pie
cake_or_pie = config.attr(type=lambda x: "cake" if bool(x) else "pie")
```

7.4 Configuration hooks

The BSB provides a small and elegant hook system. The system allows the user to hook methods of classes. It is intended to be a hooking system that requires bidirectional cooperation: the developer declares which hooks they provide and the user is supposed to only hook those functions. Using the hooks in other places will behave slightly different, see the note on *wild hooks*.

For a list of BSB endorsed hooks see *list of hooks*.

7.4.1 Calling hooks

A developer can call the user-registered hook using *bsb.config.run_hook()*:

```
import bsb.config

bsb.config.run_hook(instance, "my_hook")
```

This will check the class of instance and all of its parent classes for implementations of `__my_hook__` and execute them in closest relative first order, starting from the class of `instance`. These `__my_hook_` methods are known as *essential hooks*.

7.4.2 Adding hooks

Hooks can be added to class methods using the *bsb.config.on()* decorator (or *bsb.config.before()*/*bsb.config.after()*). The decorated function will then be hooked onto the given class:

```
from bsb import config
from bsb.core import Scaffold
from bsb.simulation import Simulation

@config.on(Simulation, "boot")
def print_something(self):
  print("We're inside of `Simulation`'s `boot` hook!")
  print(f"The {self.name} simulation uses {self.simulator}.")

cfg = config.Configuration.default()
cfg.simulations["test"] = Simulation(simulator="nest", ...)
scaffold = Scaffold(cfg)
# We're inside of the `Simulation`s `boot` hook!
# The test simulation uses nest.
```

7.4.3 Essential hooks

Essential hooks are those that follow Python's "magic method" convention (`__magic__`). Essential hooks allow parent classes to execute hooks even if child classes override the direct `my_hook` method. After executing these essential hooks `instance.my_hook` is called which will contain all of the non-essential class hooks. Unlike non-essential hooks they are not run whenever the hooked method is executed but only when the hooked method is invoked through

7.4.4 Wild hooks

Since the non-essential hooks are wrappers around the target method you could use the hooking system to hook methods of classes that aren't ever invoked as a hook, but still used during the operation of the class and your hook will be executed anyway. You could even use the hooking system on any class not part of the BSB at all. Just keep in mind that if you place an essential hook onto a target method that's never explicitly invoked as a hook that it will never run at all.

7.4.5 List of hooks

__boot__?

class bsb.config.**Configuration**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

The main Configuration object containing the full definition of a scaffold model.

after_connectivity

after_placement

attr_name = '{root}'

cell_types

connectivity

classmethod default()

get_node_name()

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

network
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

node_name = '{root}'

partitions

placement

regions

simulations

storage
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

bsb.config.**after**(*hook*, *cls*, *essential=False*)
Register a class hook to run after the target method.

Parameters

- **hook** (*str*) – Name of the method to hook.
- **cls** (*type*) – Class to hook.
- **essential** (*bool*) – If the hook is essential, it will always be executed even in child classes that override the hook. Essential hooks are only lost if the method on `cls` is replaced.

bsb.config.**attr**(***kwargs*)
Create a configuration attribute.

Only works when used inside of a class decorated with the *node*, *dynamic*, *root* or *pluggable* decorators.

Parameters

- **type** (*Callable*) – Type of the attribute's value.
- **required** (*bool*) – Should an error be thrown if the attribute is not present?
- **default** (*Any*) – Default value.

- **call_default** (*bool*) – Should the default value be used (False) or called (True). Useful for default values that should not be shared among objects.
- **key** – If True the key under which the parent of this attribute appears in its parent is stored on this attribute. Useful to store for example the name of a node appearing in a dict

bsb.config.**before**(*hook*, *cls*, *essential=False*)

Register a class hook to run before the target method.

Parameters

- **hook** (*str*) – Name of the method to hook.
- **cls** (*type*) – Class to hook.
- **essential** (*bool*) – If the hook is essential, it will always be executed even in child classes that override the hook. Essential hooks are only lost if the method on `cls` is replaced.

bsb.config.**catch_all**(***kwargs*)

Catches any unknown key with a value that can be cast to the given type and collects them under the attribute name.

bsb.config.**copy_template**(*template*, *output='network_configuration.json'*, *path=None*)

bsb.config.**dict**(***kwargs*)

Create a configuration attribute that holds a key value pairs of configuration values. Best used only for configuration nodes. Use an *attr()* in combination with a *types.dict* type for simple values.

bsb.config.**dynamic**(*node_cls=None*, *attr_name='cls'*, *classmap=None*, *auto_classmap=False*, *classmap_entry=None*, ***kwargs*)

Decorate a class to be castable to a dynamically configurable class using a class configuration attribute.

Example: Register a required string attribute `class` (this is the default):

```
@dynamic
class Example:
    pass
```

Example: Register a string attribute `type` with a default value 'pkg.DefaultClass' as dynamic attribute:

```
@dynamic(attr_name='type', required=False, default='pkg.DefaultClass')
class Example:
    pass
```

Parameters

- **attr_name** (*str*) – Name under which to register the class attribute in the node.
- **kwargs** – All keyword arguments are passed to the constructor of the *attribute*.

bsb.config.**from_content**(*content*, *path=None*)

bsb.config.**from_file**(*file*)

bsb.config.**get_config_path**()

bsb.config.**get_parser**(*parser_name*)

Create an instance of a configuration parser that can parse configuration strings into configuration trees, or serialize trees into strings.

Configuration trees can be cast into Configuration objects.

bsb.config.**has_hook**(*instance*, *hook*)
Checks the existence of a method or essential method on the `instance`.

Parameters

- **instance** (*object*) – Object to inspect.
- **hook** (*str*) – Name of the hook to look for.

bsb.config.**list**(***kwargs*)
Create a configuration attribute that holds a list of configuration values. Best used only for configuration nodes. Use an *attr()* in combination with a *types.list* type for simple values.

bsb.config.**node**(*node_cls*, *root=False*, *dynamic=False*, *pluggable=False*)
Decorate a class as a configuration node.

bsb.config.**on**(*hook*, *cls*, *essential=False*, *before=False*)
Register a class hook.

Parameters

- **hook** (*str*) – Name of the method to hook.
- **cls** (*type*) – Class to hook.
- **essential** (*bool*) – If the hook is essential, it will always be executed even in child classes that override the hook. Essential hooks are only lost if the method on `cls` is replaced.
- **before** (*bool*) – If `before` the hook is executed before the method, otherwise afterwards.

bsb.config.**pluggable**(*key*, *plugin_name=None*)
Create a node whose configuration is defined by a plugin.

Example: If you want to use the *attr* to chose from all the installed *dbbs_scaffold.my_plugin* plugins:

```
@pluggable('attr', 'my_plugin')
class PluginNode:
    pass
```

This will then read *attr*, load the plugin and configure the node from the node class specified by the plugin.

Parameters **plugin_name** (*str*) – The name of the category of the plugin endpoint

bsb.config.**property**(*val=None*, */*, ***kwargs*)
Provide a value for a parent class' attribute. Can be a value or a callable, a property object will be created from it either way.

bsb.config.**ref**(*reference*, ***kwargs*)
Create a configuration reference.

Configuration references are attributes that transform their value into the value of another node or value in the document:

```
{
  "keys": {
      "a": 3,
      "b": 5
  },
  "simple_ref": "a"
}
```

With `simple_ref = config.ref(lambda root, here:  here["keys"])` the value `a` will be looked up in the configuration object (after all values have been cast) at the location specified by the callable first argument.

bsb.config.**reflist**(*reference*, ***kwargs*)
Create a configuration reference list.

bsb.config.**root**(*root_cls*)
Decorate a class as a configuration root node.

bsb.config.**run_hook**(*obj*, *hook*, **args*, ***kwargs*)
Execute the hook hook of obj.

Runs the hook method obj but also looks through the class hierarchy for essential hooks with the name __<hook>__.

Note: Essential hooks are only ran if the method is called using run_hook while non-essential hooks are wrapped around the method and will always be executed when the method is called (see https://github.com/dbbs-lab/bsb/issues/158).

---

bsb.config.**slot**(***kwargs*)
Create an attribute slot that is required to be overriden by child or plugin classes.

bsb.config.**unset**()
Override and unset an inherited configuration attribute.

bsb.config.**walk_node_attributes**(*node*)
Walk over all of the child configuration nodes and attributes of node.

Returns attribute, node, parents

Return type Tuple[*ConfigurationAttribute*, Any, Tuple]

bsb.config.**walk_nodes**(*node*)
Walk over all of the child configuration nodes of node.

Returns node generator

Return type Any

CHAPTER

EIGHT

BSB JSON PARSER

The BSB's JSON parser is built on top of Python's json module and adds 2 additional features:

- JSON references
- JSON imports

8.1 JSON References

References point to another JSON dictionary somewhere in the same or another document and copy over that dictionary into the parent of the reference statement:

```
{
  "target": {
    "A": "value",
    "B": "value"
  },
  "parent": {
    "$ref": "#/target"
  }
}
```

Will be parsed into:

```
{
  "target": {
    "A": "value",
    "B": "value"
  },
  "parent": {
    "A": "value",
    "B": "value"
  }
}
```

Note: The data that you import/reference will be combined with the data that's already present in the parent. The data that is already present in the parent will overwrite keys that are imported. In the special case that the import and original both specify a dictionary both dictionaries' keys will be merged, with again (and recursively) the original data overwriting the imported data.

8.1.1 Reference statement

The reference statement consists of the `$ref` key and a 2-part value. The first part of the statement before the `#` is the `document`-clause and the second part the `reference`-clause. If the `#` is omitted the entire value is considered a `reference`-clause.

The document clause can be empty or omitted and the reference will point to somewhere within the same document. When a document clause is given it can be an absolute or relative path to another JSON document.

The reference clause must be a JSON path, either absolute or relative to a JSON dictionary. JSON paths use the `/` to traverse a JSON document:

```
{
  "walk": {
    "down": {
      "the": {
        "path": {}
      }
    }
  }
}
```

Where the deepest node could be accessed with the JSON path `/walk/down/the/path`.

Warning: Relative reference clauses are valid! It's easy to forget the initial `/` of a reference clause! Take `other_doc.json#some/path` as example. If this reference is given from `my/own/path` then you'll be looking for `my/own/path/some/path` in the other document!

8.2 JSON Imports

Imports are the bigger cousin of the reference. They can import multiple dictionaries from a common parent at the same time. Where the reference would only be able to import either the whole parent or a single child, the import can selectively pick children to copy as siblings:

```
{
  "target": {
    "A": "value",
    "B": "value",
    "C": "value"
  },
  "parent": {
    "$import": {
      "ref": "#/target",
      "values": ["A", "C"]
    }
  }
}
```

Will be parsed into:

```
{
  "target": {
```

(continues on next page)

(continued from previous page)

```
    "A": "value",
    "B": "value",
    "C": "value"
  },
  "parent": {
    "A": "value",
    "C": "value"
  }
}
```

Note: The data that you import/reference will be combined with the data that's already present in the parent. The data that is already present in the parent will overwrite keys that are imported. In the special case that the import and original both specify a dictionary both dictionaries' keys will be merged, with again (and recursively) the original data overwriting the imported data.

8.2.1 The import statement

The import statement consists of the `$import` key and a dictionary with 2 keys:

- The `ref` key (note there's no `$`) which will be treated as a reference statement. And used to point at the import's reference target.
- The `value` key which lists which keys to import from the reference target.

CHAPTER

NINE

CONFIGURATION REFERENCE

9.1 Root nodes

```
{
  "storage": {

  },
  "network": {

  },
  "regions": {

  },
  "partitions": {

  },
  "cell_types": {

  },
  "placement": {

  },
  "after_placement": {

  },
  "connectivity": {

  },
  "after_connectivity": {

  },
  "simulations": {

  }
}
```

9.1.1 Storage

```
{
  "storage": {
    "engine": "hdf5",
    "root":  "my_file.hdf5"
  }
}
```

- *engine*: The name of the storage engine to use.
- *root*: The storage engine specific identifier of the location of the storage.

Note: Storage nodes are plugins and can contain plugin specific configuration.

9.1.2 Network

```
{
  "network": {
    "x": 200,
    "y": 200,
    "z": 200,
    "chunk_size":  50
  }
}
```

- ***x*, *y* and *z*: Loose indicators of the** scale of the network. They are handed to the topology of the network to scale itself. They do not restrict cell placement.
- *chunk_size*: The size used to parallelize the topology into multiple rhomboids.

9.1.3 Regions

```
{
  "regions": {
    "my_region": {
      "cls": "stack",
      "offset": [100.0, 0.0, 0.0]
    }
  }
}
```

- *cls*: Class of the region.
- *offset*: Offset of this region to its parent in the topology.

Note: Region nodes are dynamic and can contain class specific configuration.

9.1.4 Partitions

```
{
  "partitions": {
    "my_partition": {
      "cls": "layer",
      "region": "my_region",
      "thickness": 100.0,
      "stack_index": 0
    }
  }
}
```

- *cls*: Class of the partition.
- *region*: By-name reference to a region.

Note: Partition nodes are dynamic and can contain class specific configuration.

9.1.5 Cell types

```
{
  "cell_types": {
    "my_cell_type": {
      "entity": false,
      "spatial": {
        "radius": 10.0,
        "geometrical": {
          "axon_length": 150.0,
          "other_hints": "hi!"
        },
        "morphological": [
          {
            "selector": "by_name",
            "names": ["short_*"]
          },
          {
            "selector": "by_name",
            "names": ["long_*"]
          }
        ]
      },
      "plotting": {
        "display_name": "Fancy Name",
        "color": "pink",
        "opacity": 1.0
      }
    }
  }
}
```

CHAPTER

TEN

INTRODUCTION

The command line interface is composed of a collection of pluggable commands. Open up your favorite terminal and enter the `bsb --help` command to verify you correctly installed the software.

Each command can give command specific arguments, options or set global options. For example:

```
# Without arguments, relying on project settings defaults
bsb compile
# Providing the argument
bsb compile my_config.json
# Overriding the global verbosity option
bsb compile --verbosity 4
```

10.1 Writing your own commands

You can add your own commands into the CLI by creating a class that inherits from *bsb.cli.commands.BsbCommand* and registering its module as a `bsb.commands` entry point. You can provide a `name` and `parent` in the class argument list. If no parent is given the command is added under the root `bsb` command:

```
# BaseCommand inherits from BsbCommand too but contains the default CLI command
# functions already implemented.
from bsb.commands import BaseCommand

class MyCommand(BaseCommand, name="test"):
  def handler(self, namespace):
    print("My command was run")

class MySubcommand(BaseCommand, name="sub", parent=MyCommand):
  def handler(self, namespace):
    print("My subcommand was run")
```

In setup.py (assuming the above module is importable as `my_pkg.commands`):

```
"entry_points": {
  "bsb.commands" = ["my_commands = my_pkg.commands"]
}
```

After installing the setup with pip your command will be available:

```
$> bsb test
My command was run
$> bsb test sub
My subcommand was run
```

CHAPTER

ELEVEN

LIST OF COMMANDS

11.1 compile

Creates a network

11.2 simulate

Run a simulation

CHAPTER

TWELVE

INTRODUCTION

The topology module allows you to make abstract descriptions of the spatial layout of pieces of the region you are modelling. *Partitions* help you define shapes to place into your region such as layers, cubes, spheres, meshes and so on. *Regions* help you put those pieces together by arranging them on top of each other, next to each other, away from each other, ... You can define your own `Partitions` and `Regions`; as long as each partition knows how to transform itself into a collection of voxels (volume pixels) and each region knows how to arrange its children these elements can become the building blocks of an arbitrarily large and parallelizable model description.

CHAPTER

THIRTEEN

TOPOLOGY MODULE

13.1 Overview

The topology module helps the placement module determine the shape and organization of the simulated space. Every simulated space contains a flat collection of *Partitions* organized into a hierarchy by a tree of *Regions*.

Partitions are defined by a least dominant corner (e.g. `(50, 50, 50)`) and a most dominant corner (e.g. `(90, 90, 90)`) referred to as the LDC and MDC respectively. With that information the outer bounds of the partition are defined. Partitions have to be able to determine a `volume`, `surface` and `voxels` given some bounds to intersect with. On top of that they have to be able to return a list of `chunks` they belong to given a chunk size.

13.2 Partitions

- What are Partitions supposed to be able to do? * Chunk themselves * (Voxelize themselves)

13.3 Regions

- What are regions supposed to do? * Arrange their children * Check the bounds of their children! –> Introduce `check_bounds` with def. impl.?

CHAPTER

FOURTEEN

REGIONS

14.1 List of builtin regions

CHAPTER

FIFTEEN

PARTITIONS

15.1 Voxels

Voxel partitions are an irregular shape in space, described by a group of rhomboids, called a *VoxelSet*. The voxel partition needs to be configured with a *VoxelLoader* to load the voxelset from somewhere. Most brain atlases scan the brain in a 3D grid and publish their data in the same way, usually in the Nearly Raw Raster Data format, NRRD. In general, whenever you have a voxelized 3D image, a `Voxels` partition will help you define the shapes contained within.

15.1.1 NRRD

To load data from NRRD files use the *NrrdVoxelLoader*. By default it will load all the nonzero values in a source file:

JSON

```
{
  "partitions": {
    "my_voxel_partition": {
      "region": "some_region",
      "voxels": {
        "type": "nrrd",
        "source": "data/my_nrrd_data.nrrd",
        "voxel_size": 25
      }
    }
  }
}
```

PYTHON

```
from bsb.topology.partition import Voxels
from bsb.voxels import NrrdVoxelLoader

loader = NrrdVoxelLoader(source="data/my_nrrd_data.nrrd", voxel_size=25)
partition = Voxels(voxels=loader)
```

The loader has a *get_voxelset()* method to access the loaded *VoxelSet*. The nonzero values will be stored on the *VoxelSet* as a *data column*. Data columns can be accessed through the *data* property:

```
loader = NrrdVoxelLoader(source="data/my_nrrd_data.nrrd", voxel_size=25)
vs = loader.get_voxelset()
# Prints the information about the VoxelSet, like how many there are etc.
print(vs)
# Prints an (Nx1) array with one nonzero value for each selected voxel.
print(vs.data.shape)
partition = Voxels(voxels=loader)
```

Using masks

Instead of capturing the nonzero values, you can give a *mask_value* to select all voxels with that value. Additionally, you can specify a dedicated NRRD file that contains a mask, the *mask_source*, and fetch the data of the source file(s) based on this mask. This is useful when one file contains the shapes of certain brain structure, and other files contain cell population density values, gene expression values, … and you need to fetch the values associated to your brain structure:

JSON

```
{
  "partitions": {
    "my_voxel_partition": {
      "region": "some_region",
      "voxels": {
        "type": "nrrd",
        "mask_value": 55,
        "mask_source": "data/brain_structures.nrrd",
        "source": "data/whole_brain_cell_densities.nrrd",
        "voxel_size": 25
      }
    }
  }
}
```

PYTHON

```
from bsb.topology.partition import Voxels
from bsb.voxels import NrrdVoxelLoader

loader = NrrdVoxelLoader(
  mask_value=55,
  mask_source="data/brain_structures.nrrd",
  source="data/whole_brain_cell_densities.nrrd",
  voxel_size=25,
)
vs = loader.get_voxelset()
# This prints the density data of all voxels that were tagged with `55`
# in the mask source file (your brain structure).
print(vs.data)
partition = Voxels(voxels=loader)
```

Using multiple source files

It's possible to use multiple source files. If no mask source is applied, a supermask will be created from all the source file selections, and in the end, this supermask is applied to each source file. Each source file will generate a data column, in the order that they appear in the *sources* attribute:

JSON

```
{
  "partitions": {
    "my_voxel_partition": {
      "region": "some_region",
      "voxels": {
        "type": "nrrd",
        "mask_value": 55,
        "mask_source": "data/brain_structures.nrrd",
        "sources": [
          "data/type1_data.nrrd",
          "data/type2_data.nrrd",
          "data/type3_data.nrrd",
        ],
        "voxel_size": 25
      }
    }
  }
}
```

PYTHON

```
from bsb.topology.partition import Voxels
from bsb.voxels import NrrdVoxelLoader

loader = NrrdVoxelLoader(
  mask_value=55,
  mask_source="data/brain_structures.nrrd",
  sources=[
    "data/type1_data.nrrd",
    "data/type2_data.nrrd",
    "data/type3_data.nrrd",
  ],
  voxel_size=25,
)
vs = loader.get_voxelset()
# `data` will be an (Nx3) matrix that contains `type1` in `data[:, 0]`, `type2` in
# `data[:, 1]` and `type3` in `data[:, 2]`.
print(vs.data.shape)
partition = Voxels(voxels=loader)
```

Tagging the data columns with keys

Instead of using the order in which the sources appear, you can add data keys to associate a name with each column. Data columns can then be indexed as strings:

JSON

```
{
  "partitions": {
    "my_voxel_partition": {
      "region": "some_region",
      "voxels": {
        "type": "nrrd",
        "mask_value": 55,
        "mask_source": "data/brain_structures.nrrd",
        "sources": [
          "data/type1_data.nrrd",
          "data/type2_data.nrrd",
          "data/type3_data.nrrd",
        ],
        "keys": ["type1", "type2", "type3"],
        "voxel_size": 25
      }
    }
  }
}
```

PYTHON

```
from bsb.topology.partition import Voxels
from bsb.voxels import NrrdVoxelLoader

loader = NrrdVoxelLoader(
  mask_value=55,
  mask_source="data/brain_structures.nrrd",
  sources=[
    "data/type1_data.nrrd",
    "data/type2_data.nrrd",
    "data/type3_data.nrrd",
  ],
  keys=["type1", "type2", "type3"],
  voxel_size=25,
)
vs = loader.get_voxelset()
# Access data columns as strings
print(vs.data[:, "type1"])
# Index multiple columns like this:
print(vs.data[:, "type1", "type3"])
partition = Voxels(voxels=loader)
```

15.1.2 Allen Atlas integration

The [Allen Brain Atlas](https://mouse.brain-map.org/) provides NRRD files and brain structure annotations; with the BSB these can be seamlessly integrated into your workflow using the *AllenStructureLoader*. In Allen-speak, partitions are `Structures`, each structure has an id, name and acronym. The BSB accepts any of those identifiers and will load the Allen Atlas data and select the structure for you. You can then download any Allen Atlas image as a local NRRD file, and associate it to the structure:

JSON

```
{
  "partitions": {
    "my_voxel_partition": {
      "region": "some_region",
      "voxels": {
        "type": "allen",
        "struct_name": "VAL",
        "sources": [
          "data/allen_gene_expression_25.nrrd"
        ],
        "keys": ["expression"],
        "voxel_size": 25
      }
    }
  }
}
```

PYTHON

```
from bsb.topology.partition import Voxels
from bsb.voxels import AllenStructureLoader

loader = AllenStructureLoader(
  # Loads the "ventroanterolateral thalamic nucleus" from the
  # ALlen Mouse Brain Atlas
  struct_name="VAL",
  mask_source="data/brain_structures.nrrd",
  sources=[
    "data/allen_gene_expression_25.nrrd",
  ],
  keys=["expression"],
  voxel_size=25,
)
partition = Voxels(voxels=loader)
```

CHAPTER

SIXTEEN

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]
```

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.

16.1 Constructing morphologies

Although morphologies are usually imported from files into storage, it can be useful to know how to create them for debugging, testing and validating. First create your branches, then attach them together and provide the roots to the Morphology constructor:

```
from bsb.morphologies import Branch, Morphology
import numpy as np

# x, y, z, radii
branch = Branch(
  np.array([0, 1, 2]),
  np.array([0, 1, 2]),
  np.array([0, 1, 2]),
```

(continues on next page)

(continued from previous page)

```
  np.array([1, 1, 1]),
)
child_branch = Branch(
  np.array([2, 3, 4]),
  np.array([2, 3, 4]),
  np.array([2, 3, 4]),
  np.array([1, 1, 1]),
)
branch.attach_child(child_branch)
m = Morphology([branch])
```

Note: Attaching branches is merely a graph-level connection that aids in iterating the morphology, no spatial connection information is inferred between the branches. Detaching and attaching it elsewhere won't result in any spatial changes, it will only affect iteration order. Keep in mind that that still affects how they are stored and still has drastic consequences if connections have already been made using that morphology (as connections use branch indices).

16.1.1 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 *Morphology repositories*.

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.load("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)
```

(continues on next page)

(continued from previous page)

```
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 an object you can do this by flattening the morphology:

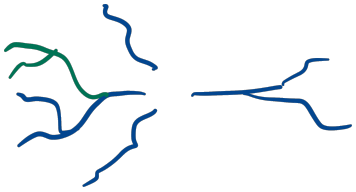
```
from bsb.core import from_hdf5

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

print("All the branches in depth-first order:", morfo.branches)
print("All the points on those branches in depth first order:")
print("- As vectors:", morfo.flatten())
print("- As matrix:", morfo.flatten(matrix=True).shape)
```

16.2 Subtree transformations

A subtree is a (sub)set of a morphology defined by a set of *roots* and all of its downstream branches (i.e. the branches *emanating* from a set of roots). A subtree with roots equal to the roots of the morphology is equal to the entire morphology, and all transformations valid on a subtree are also valid morphology transformations.

16.2.1 Selection

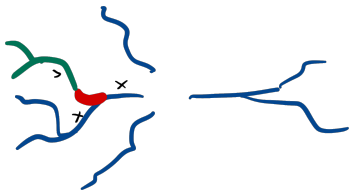
Subtrees can be selected using label(s) on the morphology.



```
axon = morfo.select("axon")
# Multiple labels can be given
hybrid = morfo.select("proximal", "distal")
```

> **Warning:** Only branches that have all of their points labelled with a label will be selected.

Selection will always select all emanating branches as well:



```
tuft = morfo.select("dendritic_piece")
```

16.2.2 Translation

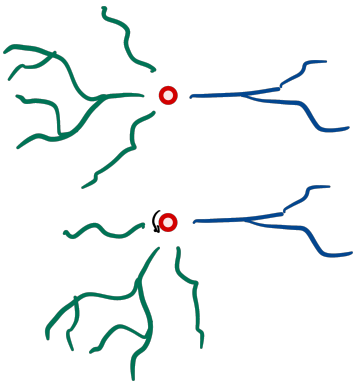
```
axon.translate([24, 100, 0])
```

16.2.3 Centering

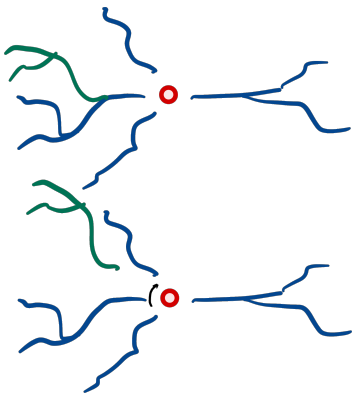
Subtrees may center themselves by offsetting the geometric mean of the origins of each root.

16.2.4 Rotation

Subtrees may be rotated around a singular point (by default around 0), by given 2 orientation vectors:



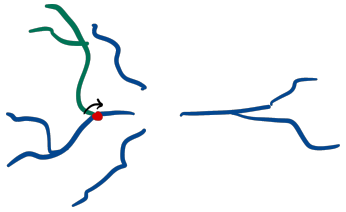
```
dendrites.rotate([0, 1, 0], [1, 0, 0])
```



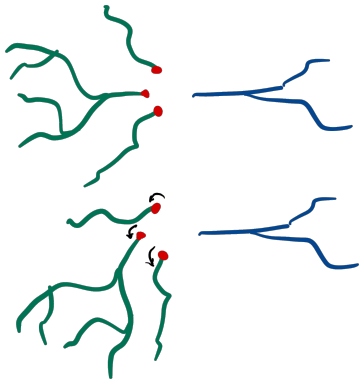
```
dendrite.rotate([0, 1, 0], [1, 0, 0])
```

16.2.5 Root-rotation

Subtrees may rotate each subtree around their respective roots:



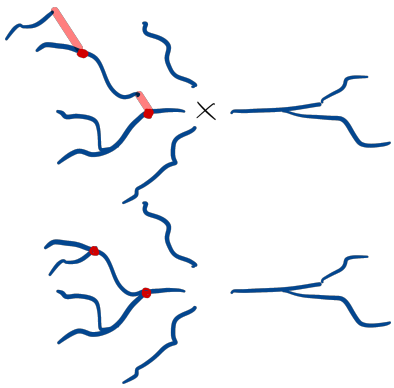
```
dendrite.root_rotate([0, 1, 0], [1, 0, 0])
```



```
dendrites.root_rotate([0, 1, 0], [1, 0, 0])
```

16.2.6 Gap closing

Subtree gaps between parent and child branches can be closed:



```
dendrites.close_gaps()
```

Note: The gaps between any subtree branch and its parent will be closed, even if the parent is not part of the subtree. This means that gaps of roots of a subtree may be closed as well.

Note: Gaps between roots are not collapsed.

16.2.7 Collapsing

Collapse the roots of a subtree onto a single point, by default the origin.

16.3 Morphology preloading

Reading the morphology data from the repository takes time. Usually morphologies are passed around in the framework as *StoredMorphologies*. These objects have a *storage.interfaces.StoredMorphology.load()* method to load the *morphologies.Morphology* object from storage and a *storage.interfaces.StoredMorphology.get_meta()* method to return the metadata.

16.4 Morphology selectors

The most common way of telling the framework which morphologies to use is through *MorphologySelectors*. A selector should implement *validate()* and *pick()* methods.

`validate` can be used to assert that all the required morphologies are present, while `pick` needs to return `True`/`False` to include a morphology or not. Both methods are handed *storage.interfaces.StoredMorphology* objects, only `load` morphologies if it is impossible to determine the outcome from the metadata.

```
from bsb.objects.cell_type import MorphologySelector
from bsb import config

@config.node
class MySizeSelector(MorphologySelector, classmap_entry="by_size"):
  min_size = config.attr(type=float, default=20)
  max_size = config.attr(type=float, default=50)

  def validate(self, morphos):
    if not all("size" in m.get_meta() for m in morphos):
      raise Exception("Missing size metadata for the size selector")

  def pick(self, morpho):
    meta = morpho.get_meta()
    return meta["size"] > self.min_size and meta["size"] < self.max_size
```

16.5 Morphology metadata

Currently unspecified, up to the Storage and MorphologyRepository support to return a dictionary of available metadata from *get_meta()*.

16.6 Morphology distributors

16.7 MorphologySets

MorphologySets are the result of *distributors* assigning morphologies to placed cells. They consist of a list of *StoredMorphologies*, a vector of indices referring to these stored morphologies and a vector of rotations. You can use *iter_morphologies()* to iterate over each morphology.

```
ps = network.get_placement_set("my_detailed_neurons")
positions = ps.load_positions()
morphology_set = ps.load_morphologies()
rotations = ps.load_rotations()
cache = morphology_set.iter_morphologies(cache=True)
for pos, morpho, rot in zip(positions, cache, rotations):
  morpho.rotate(rot)
```

16.8 Reference

Sorry robots of the future, this is still just a quick internal stub I haven't properly finished.

It goes `morphology-on-file` into `repository` that the `storage` needs to provide support for. Then after a placement job has placed cells for a chunk, the positions are sent to a `distributor` that is supposed to use the `indicators` to ask the `storage.morphology_repository` which `loaders` are appropriate for the given `selectors`, then, still hopefully using just morpho metadata the `distributor` generates indices and rotations. In more complex cases the `selector` and `distributor` can both load the morphologies but this will slow things down.

In the simulation step, these (possibly dynamically modified) morphologies are passed to the cell model instantiators.

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.

as_arc()

Return the branch as a vector of arclengths in the closed interval [0, 1]. An arclength is the distance each point to the start of the branch along the branch axis, normalized by total branch length. A point at the start will have an arclength close to 0, and a point near the end an arclength close to 1

Returns Vector of branch points as arclengths.

Return type numpy.ndarray

as_matrix(*with_radius=False*)

Return the branch as a (PxV) matrix. The different vectors (V) are columns and each point (P) is a row.

Parameters **with_radius** (*bool*) – Include the radius vector. Defaults to `False`.

Returns Matrix of the branch vectors.

Return type numpy.ndarray

attach_child(*branch*)
Attach a branch as a child to this branch.

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

ceil_arc_point(*arc*)
Get the index of the nearest distal arc point.

property children
Collection of the child branches of this branch.

Returns list of *Branches*

Return type list

copy()
Return a parentless and childless copy of the branch.

detach_child(*branch*)
Remove a branch as a child from this branch.

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

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

Parameters **vectors** (*list[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 Union[Tuple[numpy.ndarray],numpy.ndarray]

floor_arc_point(*arc*)
Get the index of the nearest proximal arc point.

get_arc_point(*arc*, *eps=1e-10*)
Strict search for an arc point within an epsilon.

Parameters

- **arc** (*float*) – Arclength position to look for.
- **eps** (*float*) – Maximum distance/tolerance to accept an arc point as a match.

Returns The matched arc point index, or `None` if no match is found

Return type Union[int, None]

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 the ones fully labelled with any of the given labels.

Return type list

get_labelled_points(*label*)
Filter out all points with a certain label

Parameters **label** (*str*) – The label to check for.

Returns All points with the label.

Return type List[numpy.ndarray]

has_any_label(*labels*)
Check if this branch is branch labelled with any of `labels`.

Parameters **labels** (*list*) – The labels to check for.

Return type bool

has_label(*label*)
Check if this branch is branch labelled with `label`.

Parameters **label** (*str*) – The label to check for.

Return type bool

introduce_arc_point(*arc_val*)
Introduce a new point at the given arc length.

Parameters **arc_val** (*float*) – Arc length between 0 and 1 to introduce new point at.

Returns The index of the new point.

Return type int

introduce_point(*index*, **args*, *labels=None*)
Insert a new point at `index`, before the existing point at `index`.

Parameters

- **index** (*int*) – Index of the new point.
- **args** (*float*) – Vector coordinates of the new point
- **labels** (*list*) – The labels to assign to the point.

property is_root
Returns whether this branch is root or if it has a parent.

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

Return type bool

property is_terminal
Returns whether this branch is terminal or if it has children.

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

Return type bool

label_all(**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*, *join=<built-in function or_>*)
Add labels to specific points on the branch. See *label_all()* to label the entire branch.

Parameters

- **label** (*str*) – Label to apply to the points.
- **mask** (*numpy.ndarray[bool]*) – Boolean mask equal in size to the branch. Elements set to *True* will be considered labelled.
- **join** (*Callable*) – If the label already existed, this determines how the existing and new masks are joined together. Defaults to `|` (`operator.or_`).

label_walk()
Iterate over the labels of each point in the branch.

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

root_rotate(*rot*)
Rotate the subtree emanating from each root around the start of the root

rotate(*rot*, *center=None*)
Point rotation

Parameters **rot** – Scipy rotation

Type scipy.spatial.transform.Rotation

property size
Returns the amount of points on this branch

Returns Number of points on the branch.

Return type int

walk()
Iterate over the points in the branch.

class bsb.morphologies.**Morphology**(*roots*, *meta=None*)
A multicompartmental spatial representation of a cell based on a directed acyclic graph of branches whom consist of data vectors, each element of a vector being a coordinate or other associated data of a point on the branch.

class bsb.morphologies.**MorphologySet**(*loaders*, *m_indices*)
Associates a set of *StoredMorphologies* to cells

iter_morphologies(*cache=True*, *unique=False*, *hard_cache=False*)
Iterate over the morphologies in a MorphologySet with full control over caching.

Parameters

- **cache** (*bool*) – Use *Soft caching* (1 copy stored in mem per cache miss, 1 copy created from that per cache hit).
- **hard_cache** – Use *Soft caching* (1 copy stored on the loader, always same copy returned from that loader forever).

class bsb.morphologies.**RotationSet**(*data*)
Set of rotations. Returned rotations are of scipy.spatial.transform.Rotation

class bsb.morphologies.**SubTree**(*branches*, *sanitize=True*)
Collection of branches, not necesarily all connected.

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[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 Union[Tuple[numpy.ndarray],numpy.ndarray]

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 the ones fully labelled with any of the given labels.

Return type list

root_rotate(*rot*)
Rotate the subtree emanating from each root around the start of the root

rotate(*rot*, *center=None*)
Point rotation

Parameters **rot** – Scipy rotation

Type scipy.spatial.transform.Rotation

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

CHAPTER

SEVENTEEN

MORPHOLOGY REPOSITORIES

Morphology repositories (MRs) are an interface of the *storage* module and can be supported by the *Engine* so that morphologies can be stored inside the network storage.

The MR of a network is accessible as `network.morphologies` and has a *save()* method to store *Morphology*. To access a *Morphology* you can use *load()* or create a preloader that loads the meta information, you can then use its `load` method to load the *Morphology* if you need it.

class bsb.storage.interfaces.**MorphologyRepository**(*engine*)

CHAPTER

EIGHTEEN

MORPHOLOGYSET

18.1 Soft caching

Every time a morphology is loaded, it has to be read from disk and pieced together. If you use soft caching, upon loading a morphology it is kept in cache and each time it is re-used a copy of the cached morphology is created. This means that the storage only has to be read once per morphology, but additional memory is used for each unique morphology in the set. If you're iterating, the soft cache is cleared immediately after the iteration stops. Soft caching is available by passing `cache=True` to *iter_morphologies()*:

```
from bsb.core import from_hdf5

network = from_hdf5
ps = network.get_placement_set("my_cell")
ms = ps.load_morphologies()
for morpho in ms.iter_morphologies(cache=True):
  morpho.close_gaps()
```

CHAPTER

NINETEEN

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.

19.1 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.

19.2 Arbor

19.2.1 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.

19.2.2 Connection models

todo: doc

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

19.2.3 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

19.3 NEST

19.4 NEURON

CHAPTER

TWENTY

SIMULATION ADAPTERS

Simulation adapters form a link between the BSB and the simulation backend. They translate the stored networks into simulator specific instructions.

There are currently adapters for Arbor, NEST and NEURON.

20.1 NEURON

20.1.1 List of NEURON devices

CHAPTER

TWENTYONE

BSB

21.1 bsb package

21.1.1 Subpackages

bsb.cli package

Subpackages

bsb.cli.commands package

Module contents

Contains all of the logic required to create commands. It should always suffice to import just this module for a user to create their own commands.

Inherit from *BaseCommand* for regular CLI style commands, or from *BsbCommand* if you want more freedom in what exactly constitutes a command to the BSB.

class bsb.cli.commands.**BaseCommand**

Bases: *bsb.cli.commands.BsbCommand*

add_locals(*context*)

add_parser_arguments(*parser*)

add_parser_options(*parser*, *context*, *locals*, *level*)

add_subparsers(*parser*, *context*, *commands*, *locals*, *level*)

add_to_parser(*parent*, *context*, *locals*, *level*)

execute_handler(*namespace*, *dryrun=False*)

get_options()

class bsb.cli.commands.**BaseParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

Bases: argparse.ArgumentParser

Inherits from argparse.ArgumentParser and overloads the `error` method so that when an error occurs, instead of exiting and exception is thrown.

error(*message*)
Raise message, instead of exiting.

Parameters **message** (*str*) – Error message

class bsb.cli.commands.**BsbCommand**
Bases: object

add_to_parser()

handler(*context*)

class bsb.cli.commands.**RootCommand**
Bases: *bsb.cli.commands.BaseCommand*

get_options()

get_parser(*context*)

handler(*context*)

name = 'bsb'

bsb.cli.commands.**load_root_command**()

Module contents

bsb.cli.**handle_cli**()

bsb.cli.**handle_command**(*command*, *dryrun=False*, *exit=False*)

bsb.config package

Subpackages

bsb.config.parsers package

Submodules

bsb.config.parsers.json module

JSON parsing module. Built on top of the Python `json` module. Adds JSON imports and references.

class bsb.config.parsers.json.**JsonMeta**
Bases: object

class bsb.config.parsers.json.**JsonParser**
Bases: *bsb.config.parsers._parser.Parser*

Parser plugin class to parse JSON configuration files.

data_description = 'JSON'

data_extensions = ('json',)

parse(*content*, *path=None*)

class bsb.config.parsers.json.**json_imp**(*node*, *doc*, *ref*, *values*)
Bases: *bsb.config.parsers.json.json_ref*

resolve(*parser*, *target*)

class bsb.config.parsers.json.**json_ref**(*node*, *doc*, *ref*)
Bases: object

resolve(*parser*, *target*)

class bsb.config.parsers.json.**parsed_dict**
Bases: dict, *bsb.config.parsers.json.parsed_node*

merge(*other*)
Recursively merge the values of another dictionary into us

rev_merge(*other*)
Recursively merge ourself onto another dictionary

class bsb.config.parsers.json.**parsed_list**(*iterable=()*, */*)
Bases: list, *bsb.config.parsers.json.parsed_node*

class bsb.config.parsers.json.**parsed_node**
Bases: object

location()

Module contents

class bsb.config.parsers.**Parser**
Bases: abc.ABC

abstract parse(*content*, *path=None*)

bsb.config.templates package

Module contents

Submodules

bsb.config.nodes module

class bsb.config.nodes.**Distribution**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

distribution
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

draw(*n*)

get_node_name()

parameters

class bsb.config.nodes.**NetworkNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

boot()

chunk_size
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

x
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

y
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

z
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.config.nodes.**StorageNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

engine
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

root

bsb.config.refs module

This module contains shorthand `reference` definitions. References are used in the configuration module to point to other locations in the Configuration object.

Minimally a reference is a function that takes the configuration root and the current node as arguments, and returns another node in the configuration object:

```
def some_reference(root, here):
    return root.other.place
```

More advanced usage of references will include custom reference errors.

bsb.config.types module

class bsb.config.types.**TypeHandler**
Bases: abc.ABC

Base class for any type handler that cannot be described as a single function.

Declare the *__call__(self, value)* method to convert the given value to the desired type, raising a *TypeError* if it failed in an expected manner.

Declare the *__name__(self)* method to return a name for the type handler to display in messages to the user such as errors.

Declare the optional *__inv__* method to invert the given value back to its original value, the type of the original value will usually be lost but the type of the returned value can still serve as a suggestion.

bsb.config.types.**any**()

bsb.config.types.**class_**(*module_path=None*)
Type validator. Attempts to import the value as the name of a class, relative to the *module_path* entries, absolute or just returning it if it is already a class.

Parameters **module_path** (*list[str]*) – List of the modules that should be searched when doing a relative import.

Raises TypeError when value can't be cast.

Returns Type validator function

Return type Callable

bsb.config.types.**constant_distr**()
Type handler that turns a float into a distribution that always returns the float. This can be used in places where a distribution is expected but the user might want to use a single constant value instead.

Returns Type validator function

Return type Callable

class bsb.config.types.**deg_to_radian**
Bases: *bsb.config.types.TypeHandler*

Type validator. Type casts the value from degrees to radians.

bsb.config.types.**dict**(*type=<class 'str'>*)
Type validator for dicts. Type casts each element to the given type.

Parameters **type** (*Callable*) – Type validator of the elements.

Returns Type validator function

Return type Callable

bsb.config.types.**distribution**()
Type validator. Type casts a float to a constant distribution or a dict to a *Distribution* node.

Returns Type validator function

Return type Callable

class bsb.config.types.**evaluation**
Bases: *bsb.config.types.TypeHandler*

Type validator. Provides a structured way to evaluate a python statement from the config. The evaluation context provides `numpy` as `np`.

Returns Type validator function

Return type Callable

get_original(*value*)
Return the original configuration node associated with the given evaluated value.

Parameters **value** (*Any*) – A value that was produced by this type handler.

Raises NoneReferenceError when *value* is *None*, InvalidReferenceError when there is no config associated to the object id of this value.

bsb.config.types.**float**(*min=None*, *max=None*)
Type validator. Attempts to cast the value to an float, optionally within some bounds.

Parameters

- **min** (*float*) – Minimum valid value
- **max** (*float*) – Maximum valid value

Returns Type validator function

Raises TypeError when value can't be cast.

Return type Callable

bsb.config.types.**fraction**()
Type validator. Type casts the value into a rational number between 0 and 1 (inclusive).

Returns Type validator function

Return type Callable

bsb.config.types.**in_**(*container*)
Type validator. Checks whether the given value occurs in the given container. Uses the *in* operator.

Parameters **container** (*list*) – List of possible values

Returns Type validator function

Return type Callable

bsb.config.types.**in_classmap**()
Type validator. Checks whether the given string occurs in the class map of a dynamic node.

Returns Type validator function

Return type Callable

bsb.config.types.**int**(*min=None*, *max=None*)
Type validator. Attempts to cast the value to an int, optionally within some bounds.

Parameters

- **min** (*int*) – Minimum valid value
- **max** (*int*) – Maximum valid value

Returns Type validator function

Raises TypeError when value can't be cast.

Return type Callable

bsb.config.types.**list**(*type=<class 'str'>*, *size=None*)
Type validator for lists. Type casts each element to the given type and optionally validates the length of the list.

Parameters

- **type** (*Callable*) – Type validator of the elements.
- **size** (*int*) – Mandatory length of the list.

Returns Type validator function

Return type Callable

bsb.config.types.**list_or_scalar**(*scalar_type*, *size=None*)
Type validator that accepts a scalar or list of said scalars.

Parameters

- **scalar_type** (*type*) – Type of the scalar
- **size** (*int*) – Expand the scalar to an array of a fixed size.

Returns Type validator function

Return type Callable

bsb.config.types.**mut_excl**(**mutuals*, *required=True*, *max=1*)
Requirement handler for mutually exclusive attributes.

Parameters

- **mutuals** (*str*) – The keys of the mutually exclusive attributes.
- **required** (*bool*) – Whether at least one of the keys is required

- **max** (*int*) – The maximum amount of keys that may occur together.

Returns Requirement function

Return type Callable

bsb.config.types.**number**(*min=None*, *max=None*)
Type validator. If the given value is an int returns an int, tries to cast to float otherwise

Parameters

- **min** (*float*) – Minimum valid value
- **max** (*float*) – Maximum valid value

Returns Type validator function

Raises TypeError when value can't be cast.

Return type Callable

bsb.config.types.**or_**(**type_args*)
Type validator. Attempts to cast the value to any of the given types in order.

Parameters **type_args** (*Callable*) – Another type validator

Returns Type validator function

Raises TypeError if none of the given type validators can cast the value.

Return type Callable

bsb.config.types.**scalar_expand**(*scalar_type*, *size=None*, *expand=None*)
Create a method that expands a scalar into an array with a specific size or uses an expansion function.

Parameters

- **scalar_type** (*type*) – Type of the scalar
- **size** (*int*) – Expand the scalar to an array of a fixed size.
- **expand** (*Callable*) – A function that takes the scalar value as argument and returns the expanded form.

Returns Type validator function

Return type Callable

bsb.config.types.**str**(*strip=False*, *lower=False*, *upper=False*)
Type validator. Attempts to cast the value to an str, optionally with some sanitation.

Parameters

- **strip** (*bool*) – Trim whitespaces
- **lower** (*bool*) – Convert value to lowercase
- **upper** (*bool*) – Convert value to uppercase

Returns Type validator function

Raises TypeError when value can't be cast.

Return type Callable

bsb.config.types.**voxel_size**()

Module contents

class bsb.config.**Configuration**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

The main Configuration object containing the full definition of a scaffold model.

after_connectivity

after_placement

attr_name = '{root}'

cell_types

connectivity

classmethod default()

get_node_name()

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

network
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

node_name = '{root}'

partitions

placement

regions

simulations

storage
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.config.**ConfigurationAttribute**(*type=None*, *default=None*, *call_default=None*, *required=False*, *key=False*, *unset=False*)
Bases: object

Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

flag_dirty(*instance*)

flag_pristine(*instance*)

get_default()

get_node_name(*instance*)

is_dirty(*instance*)

should_call_default()

tree(*instance*)

bsb.config.**after**(*hook*, *cls*, *essential=False*)
Register a class hook to run after the target method.

Parameters

- **hook** (*str*) – Name of the method to hook.
- **cls** (*type*) – Class to hook.

- **essential** (*bool*) – If the hook is essential, it will always be executed even in child classes that override the hook. Essential hooks are only lost if the method on `cls` is replaced.

bsb.config.**attr**(***kwargs*)
Create a configuration attribute.

Only works when used inside of a class decorated with the *node*, *dynamic*, *root* or *pluggable* decorators.

Parameters

- **type** (*Callable*) – Type of the attribute's value.
- **required** (*bool*) – Should an error be thrown if the attribute is not present?
- **default** (*Any*) – Default value.
- **call_default** (*bool*) – Should the default value be used (False) or called (True). Useful for default values that should not be shared among objects.
- **key** – If True the key under which the parent of this attribute appears in its parent is stored on this attribute. Useful to store for example the name of a node appearing in a dict

bsb.config.**before**(*hook*, *cls*, *essential=False*)
Register a class hook to run before the target method.

Parameters

- **hook** (*str*) – Name of the method to hook.
- **cls** (*type*) – Class to hook.
- **essential** (*bool*) – If the hook is essential, it will always be executed even in child classes that override the hook. Essential hooks are only lost if the method on `cls` is replaced.

bsb.config.**catch_all**(***kwargs*)
Catches any unknown key with a value that can be cast to the given type and collects them under the attribute name.

bsb.config.**copy_template**(*template*, *output='network_configuration.json'*, *path=None*)

bsb.config.**dict**(***kwargs*)
Create a configuration attribute that holds a key value pairs of configuration values. Best used only for configuration nodes. Use an *attr()* in combination with a *types.dict* type for simple values.

bsb.config.**dynamic**(*node_cls=None*, *attr_name='cls'*, *classmap=None*, *auto_classmap=False*, *classmap_entry=None*, ***kwargs*)
Decorate a class to be castable to a dynamically configurable class using a class configuration attribute.

Example: Register a required string attribute `class` (this is the default):

```
@dynamic
class Example:
    pass
```

Example: Register a string attribute `type` with a default value 'pkg.DefaultClass' as dynamic attribute:

```
@dynamic(attr_name='type', required=False, default='pkg.DefaultClass')
class Example:
    pass
```

Parameters

- **attr_name** (*str*) – Name under which to register the class attribute in the node.

- **kwargs** – All keyword arguments are passed to the constructor of the *attribute*.

bsb.config.**from_content**(*content*, *path=None*)

bsb.config.**from_file**(*file*)

bsb.config.**get_config_path**()

bsb.config.**get_parser**(*parser_name*)
Create an instance of a configuration parser that can parse configuration strings into configuration trees, or serialize trees into strings.

Configuration trees can be cast into Configuration objects.

bsb.config.**has_hook**(*instance*, *hook*)
Checks the existence of a method or essential method on the `instance`.

Parameters

- **instance** (*object*) – Object to inspect.
- **hook** (*str*) – Name of the hook to look for.

bsb.config.**list**(***kwargs*)
Create a configuration attribute that holds a list of configuration values. Best used only for configuration nodes. Use an *attr()* in combination with a *types.list* type for simple values.

bsb.config.**node**(*node_cls*, *root=False*, *dynamic=False*, *pluggable=False*)
Decorate a class as a configuration node.

bsb.config.**on**(*hook*, *cls*, *essential=False*, *before=False*)
Register a class hook.

Parameters

- **hook** (*str*) – Name of the method to hook.
- **cls** (*type*) – Class to hook.
- **essential** (*bool*) – If the hook is essential, it will always be executed even in child classes that override the hook. Essential hooks are only lost if the method on `cls` is replaced.
- **before** (*bool*) – If `before` the hook is executed before the method, otherwise afterwards.

bsb.config.**pluggable**(*key*, *plugin_name=None*)
Create a node whose configuration is defined by a plugin.

Example: If you want to use the *attr* to chose from all the installed *dbbs_scaffold.my_plugin* plugins:

```
@pluggable('attr', 'my_plugin')
class PluginNode:
    pass
```

This will then read *attr*, load the plugin and configure the node from the node class specified by the plugin.

Parameters **plugin_name** (*str*) – The name of the category of the plugin endpoint

bsb.config.**property**(*val=None*, */*, ***kwargs*)
Provide a value for a parent class' attribute. Can be a value or a callable, a property object will be created from it either way.

bsb.config.**ref**(*reference*, ***kwargs*)
Create a configuration reference.

Configuration references are attributes that transform their value into the value of another node or value in the document:

```
{
  "keys": {
      "a": 3,
      "b": 5
  },
  "simple_ref": "a"
}
```

With `simple_ref = config.ref(lambda root, here: here["keys"])` the value `a` will be looked up in the configuration object (after all values have been cast) at the location specified by the callable first argument.

bsb.config.**reflist**(*reference*, ***kwargs*)
Create a configuration reference list.

bsb.config.**root**(*root_cls*)
Decorate a class as a configuration root node.

bsb.config.**run_hook**(*obj*, *hook*, **args*, ***kwargs*)
Execute the `hook` hook of `obj`.

Runs the `hook` method `obj` but also looks through the class hierarchy for essential hooks with the name `__<hook>__`.

Note: Essential hooks are only ran if the method is called using `run_hook` while non-essential hooks are wrapped around the method and will always be executed when the method is called (see https://github.com/dbbs-lab/bsb/issues/158).

---

bsb.config.**slot**(***kwargs*)
Create an attribute slot that is required to be overriden by child or plugin classes.

bsb.config.**unset**()
Override and unset an inherited configuration attribute.

bsb.config.**walk_node_attributes**(*node*)
Walk over all of the child configuration nodes and attributes of `node`.

Returns attribute, node, parents

Return type Tuple[*ConfigurationAttribute*, Any, Tuple]

bsb.config.**walk_nodes**(*node*)
Walk over all of the child configuration nodes of `node`.

Returns node generator

Return type Any

bsb.connectivity package

Subpackages

bsb.connectivity.detailed package

Submodules

bsb.connectivity.detailed.fiber_intersection module

class bsb.connectivity.detailed.fiber_intersection.**FiberIntersection**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *bsb.connectivity.detailed.shared.Intersectional*, *bsb.connectivity.strategy.ConnectionStrategy*

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

affinity
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

assert_voxelization(*morphology*, *compartment_types*)

connect()

contacts
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

interpolate_branches(*branches*)

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

resolution
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

to_plot
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

transformation
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

voxelize_branches(*branches*, *position*, *bounding_box=None*, *voxel_tree=None*, *map=None*, *voxel_list=None*)

class bsb.connectivity.detailed.fiber_intersection.**FiberTransform**
Bases: abc.ABC

boot()

abstract transform_branch()

transform_branches(*branches*, *offset=None*)

class bsb.connectivity.detailed.fiber_intersection.**QuiverTransform**
Bases: *bsb.connectivity.detailed.fiber_intersection.FiberTransform*

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

casts = {'vol_res': <class 'float'>}

defaults = {'quivers': None, 'vol_res': 10.0, 'vol_start': [0.0, 0.0, 0.0]}

get_branch_direction(*branch*)

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** (:~class:.*morphologies.Branch*) – a branch of the current fiber to be transformed

Returns a transformed branch

Return type :~class:.*morphologies.Branch*

validate()

bsb.connectivity.detailed.shared module

class bsb.connectivity.detailed.shared.**Intersectional**
Bases: object

candidate_intersection(*target_coll*, *candidate_coll*)

get_region_of_interest(*chunk*)

bsb.connectivity.detailed.touch_detection module

class bsb.connectivity.detailed.touch_detection.**TouchDetector**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *bsb.connectivity.strategy.ConnectionStrategy*, *bsb.connectivity.detailed.shared.Intersectional*

Connectivity based on intersection of detailed morphologies

allow_zero_contacts
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

cell_intersection_plane
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

cell_intersection_radius
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

compartment_intersection_plane
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

compartment_intersection_radius
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

connect()

contacts
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_compartment_intersections(*touch_info*, *from_pos*, *to_pos*)

get_node_name()

get_search_radius(*cell_type*)

intersect_cells(*touch_info*)

intersect_compartments(*touch_info*, *candidate_map*)

class bsb.connectivity.detailed.touch_detection.**TouchInformation**(*from_cell_type*, *from_cell_compartments*, *to_cell_type*, *to_cell_compartments*)

Bases: object

bsb.connectivity.detailed.voxel_intersection module

class bsb.connectivity.detailed.voxel_intersection.**VoxelIntersection**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *bsb.connectivity.detailed.shared.Intersectional*, *bsb.connectivity.strategy.ConnectionStrategy*

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
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

cache
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

connect(*pre*, *post*)

contacts
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

favor_cache
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

validate()

voxels_post
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

voxels_pre
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

Module contents

Submodules

bsb.connectivity.general module

class bsb.connectivity.general.**AllToAll**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.connectivity.strategy.ConnectionStrategy*

All to all connectivity between two neural populations

connect()

get_region_of_interest(*chunk*)

class bsb.connectivity.general.**Convergence**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.connectivity.strategy.ConnectionStrategy*

Implementation of a general convergence connectivity between two populations of cells (this does not work with entities)

connect()

convergence
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

validate()

class bsb.connectivity.general.**ExternalConnections**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.connectivity.strategy.ConnectionStrategy*

Load the connection matrix from an external source.

casts = {'format': <class 'str'>, 'headers': <class 'bool'>, 'use_map': <class 'bool'>, 'warn_missing': <class 'bool'>}

check_external_source()

connect()

defaults = {'delimiter': ',', 'format': 'csv', 'headers': True, 'use_map': False, 'warn_missing': True}

get_external_source()

has_external_source = True

required = ['source']

validate()

bsb.connectivity.strategy module

class bsb.connectivity.strategy.**ConnectionCollection**(*scaffold*, *cell_types*, *roi*)
Bases: object

property placement

class bsb.connectivity.strategy.**ConnectionStrategy**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: abc.ABC, *bsb.helpers.SortableByAfter*

after

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

abstract connect(*presyn_collection*, *postsyn_collection*)

connect_cells(*pre_set*, *post_set*, *src_locs*, *dest_locs*, *tag=None*)

create_after()

get_after()

get_cell_types()

get_node_name()

classmethod get_ordered(*objects*)

abstract get_region_of_interest(*chunk*)

has_after()

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

postsynaptic
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

presynaptic
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

queue(*pool*)
Specifies how to queue this connectivity strategy into a job pool. Can be overridden, the default implementation asks each partition to chunk itself and creates 1 placement job per chunk.

class bsb.connectivity.strategy.**HemitypeNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

cell_types

compartments
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

labels
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

Module contents

bsb.morphologies package

Module contents

Sorry robots of the future, this is still just a quick internal stub I haven't properly finished.

It goes `morphology-on-file` into `repository` that the `storage` needs to provide support for. Then after a placement job has placed cells for a chunk, the positions are sent to a `distributor` that is supposed to use the `indicators` to ask the `storage.morphology_repository` which `loaders` are appropriate for the given `selectors`, then, still hopefully using just morpho metadata the `distributor` generates indices and rotations. In more complex cases the `selector` and `distributor` can both load the morphologies but this will slow things down.

In the simulation step, these (possibly dynamically modified) morphologies are passed to the cell model instantiators.

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

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.

as_arc()
Return the branch as a vector of arclengths in the closed interval [0, 1]. An arclength is the distance each point to the start of the branch along the branch axis, normalized by total branch length. A point at the start will have an arclength close to 0, and a point near the end an arclength close to 1

Returns Vector of branch points as arclengths.

Return type numpy.ndarray

as_matrix(*with_radius=False*)
Return the branch as a (PxV) matrix. The different vectors (V) are columns and each point (P) is a row.

Parameters **with_radius** (*bool*) – Include the radius vector. Defaults to `False`.

Returns Matrix of the branch vectors.

Return type numpy.ndarray

attach_child(*branch*)
Attach a branch as a child to this branch.

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

cached_voxelize(*N*, *labels=None*)

ceil_arc_point(*arc*)
Get the index of the nearest distal arc point.

center()

property children
Collection of the child branches of this branch.

Returns list of *Branches*

Return type list

close_gaps()

collapse(*on=None*)

copy()
Return a parentless and childless copy of the branch.

detach_child(*branch*)
Remove a branch as a child from this branch.

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

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

Parameters **vectors** (*list[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 Union[Tuple[numpy.ndarray],numpy.ndarray]

floor_arc_point(*arc*)
Get the index of the nearest proximal arc point.

get_arc_point(*arc*, *eps=1e-10*)
Strict search for an arc point within an epsilon.

Parameters

- **arc** (*float*) – Arclength position to look for.
- **eps** (*float*) – Maximum distance/tolerance to accept an arc point as a match.

Returns The matched arc point index, or `None` if no match is found

Return type Union[int, None]

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 the ones fully labelled with any of the given labels.

Return type list

get_labelled_points(*label*)
Filter out all points with a certain label

Parameters **label** (*str*) – The label to check for.

Returns All points with the label.

Return type List[numpy.ndarray]

has_any_label(*labels*)
Check if this branch is branch labelled with any of `labels`.

Parameters **labels** (*list*) – The labels to check for.

Return type bool

has_label(*label*)
Check if this branch is branch labelled with `label`.

Parameters **label** (*str*) – The label to check for.

Return type bool

introduce_arc_point(*arc_val*)
Introduce a new point at the given arc length.

Parameters **arc_val** (*float*) – Arc length between 0 and 1 to introduce new point at.

Returns The index of the new point.

Return type int

introduce_point(*index*, **args*, *labels=None*)
Insert a new point at `index`, before the existing point at `index`.

Parameters

- **index** (*int*) – Index of the new point.
- **args** (*float*) – Vector coordinates of the new point
- **labels** (*list*) – The labels to assign to the point.

property is_root
Returns whether this branch is root or if it has a parent.

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

Return type bool

property is_terminal
Returns whether this branch is terminal or if it has children.

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

Return type bool

label_all(**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*, *join=<built-in function or_>*)
Add labels to specific points on the branch. See *label_all()* to label the entire branch.

Parameters

- **label** (*str*) – Label to apply to the points.
- **mask** (*numpy.ndarray[bool]*) – Boolean mask equal in size to the branch. Elements set to *True* will be considered labelled.
- **join** (*Callable*) – If the label already existed, this determines how the existing and new masks are joined together. Defaults to `|` (`operator.or_`).

label_walk()
Iterate over the labels of each point in the branch.

property parent

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

root_rotate(*rot*)
Rotate the subtree emanating from each root around the start of the root

rotate(*rot*, *center=None*)
Point rotation

Parameters **rot** – Scipy rotation

Type scipy.spatial.transform.Rotation

select(**labels*)

property size
Returns the amount of points on this branch

Returns Number of points on the branch.

Return type int

translate(*point*)

vectors = ['x', 'y', 'z', 'radii']

voxelize(*N*, *labels=None*)

walk()
Iterate over the points in the branch.

class bsb.morphologies.**Morphology**(*roots*, *meta=None*)
Bases: *bsb.morphologies.SubTree*

A multicompartmental spatial representation of a cell based on a directed acyclic graph of branches whom consist of data vectors, each element of a vector being a coordinate or other associated data of a point on the branch.

copy()

property meta

class bsb.morphologies.**MorphologySet**(*loaders*, *m_indices*)
Bases: object

Associates a set of *StoredMorphologies* to cells

clear_soft_cache()

get(*index*, *cache=True*, *hard_cache=False*)

get_indices()

iter_meta(*unique=False*)

iter_morphologies(*cache=True*, *unique=False*, *hard_cache=False*)
Iterate over the morphologies in a MorphologySet with full control over caching.

Parameters

- **cache** (*bool*) – Use *Soft caching* (1 copy stored in mem per cache miss, 1 copy created from that per cache hit).
- **hard_cache** – Use *Soft caching* (1 copy stored on the loader, always same copy returned from that loader forever).

merge(*other*)

class bsb.morphologies.**RotationSet**(*data*)
Bases: object

Set of rotations. Returned rotations are of scipy.spatial.transform.Rotation

iter(*cache=False*)

class bsb.morphologies.**SubTree**(*branches*, *sanitize=True*)
Bases: object

Collection of branches, not necesarily all connected.

property bounds

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

cached_voxelize(*N*, *labels=None*)

center()

close_gaps()

collapse(*on=None*)

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

Parameters **vectors** (*list[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 Union[Tuple[numpy.ndarray],numpy.ndarray]

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 the ones fully labelled with any of the given labels.

Return type list

property origin

root_rotate(*rot*)
Rotate the subtree emanating from each root around the start of the root

rotate(*rot*, *center=None*)
Point rotation

Parameters **rot** – Scipy rotation

Type scipy.spatial.transform.Rotation

select(**labels*)

translate(*point*)

voxelize(*N*, *labels=None*)

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

bsb.objects package

Submodules

bsb.objects.cell_type module

Module for the CellType configuration node and its dependencies.

class bsb.objects.cell_type.**CellType**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

clear(*force=False*)

clear_connections(*force=False*)

clear_placement(*force=False*)

entity
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

get_placement_set(*chunks=None*)

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

plotting
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

relay
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

spatial
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.objects.cell_type.**MorphologySelector**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: abc.ABC

get_node_name()

abstract pick(*morphology*)

selector
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

abstract validate(*all_morphos*)

class bsb.objects.cell_type.**NameSelector**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.objects.cell_type.MorphologySelector*

get_node_name()

names

pick(*morphology*)

validate(*all_morphos*)

class bsb.objects.cell_type.**Plotting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

color
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

display_name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

opacity
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.objects.cell_type.**Representation**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.indicator.PlacementIndications*

geometrical

get_node_name()

morphological

Module contents

bsb.placement package

Submodules

bsb.placement.arrays module

class bsb.placement.arrays.**ParallelArrayPlacement**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.PlacementStrategy*

Implementation of the placement of cells in parallel arrays.

angle
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

place(*chunk*, *indicators*)
Cell placement: Create a lattice of parallel arrays/lines in the layer's surface.

spacing_x
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

bsb.placement.indicator module

class bsb.placement.indicator.**PlacementIndications**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

count
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

count_ratio
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

density
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

density_ratio
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

planar_density
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

radius
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

relative_to

class bsb.placement.indicator.**PlacementIndicator**(*strat*, *cell_type*)
Bases: object

assert_indication(*attr*)

property cell_type

get_radius()

guess(*chunk=None*)

indication(*attr*)

use_morphologies()

bsb.placement.particle module

class bsb.placement.particle.**ParticlePlacement**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.PlacementStrategy*

bounded
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

place(*chunk*, *indicators*)
Central method of each placement strategy. Given a chunk, should fill that chunk with cells by calling the scaffold's (available as `self.scaffold`) *place_cells()* method.

prune
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

restrict
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

bsb.placement.satellite module

class bsb.placement.satellite.**Satellite**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.PlacementStrategy*

Implementation of the placement of cells in layers as satellites of existing cells

Places cells as a satellite cell to each associated cell at a random distance depending on the radius of both cells.

boot()

get_after()

get_node_name()

indicator_class
alias of *bsb.placement.satellite.SatelliteIndicator*

partitions

per_planet
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

place(*chunk*, *indicators*)
Central method of each placement strategy. Given a chunk, should fill that chunk with cells by calling the scaffold's (available as `self.scaffold`) *place_cells()* method.

place_type(*chunk*, *indicator*)

planet_types

class bsb.placement.satellite.**SatelliteIndicator**(*strat*, *cell_type*)
Bases: *bsb.placement.indicator.PlacementIndicator*

guess(*chunk=None*)

bsb.placement.strategy module

class bsb.placement.strategy.**Distributor**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: abc.ABC

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

abstract distribute(*partitions*, *indicator*, *positions*)

get_node_name()

class bsb.placement.strategy.**DistributorsNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

get_node_name()

morphologies
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

properties

rotations
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.placement.strategy.**Entities**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.PlacementStrategy*

Implementation of the placement of entities that do not have a 3D position, but that need to be connected with other cells of the network.

entities = True

place(*chunk*, *indicators*)
Central method of each placement strategy. Given a chunk, should fill that chunk with cells by calling the scaffold's (available as `self.scaffold`) *place_cells()* method.

queue(*pool*, *chunk_size*)
Specifies how to queue this placement strategy into a job pool. Can be overridden, the default implementation asks each partition to chunk itself and creates 1 placement job per chunk.

class bsb.placement.strategy.**ExplicitNoRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.RotationDistributor*

distribute(*partitions*, *indicator*, *positions*)

class bsb.placement.strategy.**ExternalPlacement**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.PlacementStrategy*

casts = {'format': <class 'str'>, 'warn_missing': <class 'bool'>}

check_external_source()

defaults = {'delimiter': ',', 'format': 'csv', 'map_header': None, 'warn_missing': True, 'x_header': 'x', 'y_header': 'y', 'z_header': 'z'}

get_external_source()

get_placement_count()

has_external_source = True

place()
Central method of each placement strategy. Given a chunk, should fill that chunk with cells by calling the scaffold's (available as `self.scaffold`) *place_cells()* method.

required = ['source']

validate()

class bsb.placement.strategy.**FixedPositions**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.PlacementStrategy*

get_node_name()

guess_cell_count()

place(*chunk*, *indicators*)
Central method of each placement strategy. Given a chunk, should fill that chunk with cells by calling the scaffold's (available as `self.scaffold`) *place_cells()* method.

positions
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.placement.strategy.**Implicit**
Bases: `object`

class bsb.placement.strategy.**ImplicitNoRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.ExplicitNoRotations*, *bsb.placement.strategy.Implicit*

class bsb.placement.strategy.**MorphologyDistributor**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.Distributor*

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

class bsb.placement.strategy.**PlacementStrategy**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: `abc.ABC`, *bsb.helpers.SortableByAfter*

Quintessential interface of the placement module. Each placement strategy defines an approach to placing neurons into a volume.

after

cell_types

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

create_after()

distribute
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_after()

get_indicators()
Return indicators per cell type. Indicators collect all configuration information into objects that can produce guesses as to how many cells of a type should be placed in a volume.

get_node_name()

classmethod get_ordered(*objects*)

guess_cell_count()

has_after()

indicator_class
alias of *bsb.placement.indicator.PlacementIndicator*

is_entities()

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

overrides

partitions

abstract place(*chunk*, *indicators*)
Central method of each placement strategy. Given a chunk, should fill that chunk with cells by calling the scaffold's (available as `self.scaffold`) *place_cells()* method.

place_cells(*indicator*, *positions*, *chunk*)

queue(*pool*, *chunk_size*)
Specifies how to queue this placement strategy into a job pool. Can be overridden, the default implementation asks each partition to chunk itself and creates 1 placement job per chunk.

class bsb.placement.strategy.**RandomMorphologies**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.MorphologyDistributor*

Distributes morphologies and rotations for a given set of placement indications and placed cell positions.

If omitted in the configuration the default `random` distributor is used that assigns selected morphologies randomly without rotating them.

```
{ "placement": { "place_XY": {
  "distribute": {
      "morphologies": {"cls": "random"}
  }
}}}
```

distribute(*partitions*, *indicator*, *positions*)
Uses the morphology selection indicators to select morphologies and returns a MorphologySet of randomly assigned morphologies

class bsb.placement.strategy.**RotationDistributor**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.placement.strategy.Distributor*

Rotates everything by nothing!

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

abstract distribute(*partitions*, *indicator*, *positions*)

get_node_name()

Module contents

bsb.simulation package

Submodules

bsb.simulation.adapter module

class bsb.simulation.adapter.**ProgressEvent**(*duration*, *time*)
Bases: object

class bsb.simulation.adapter.**Simulation**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

add_progress_listener(*listener*)

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

cell_models

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

connection_models

devices

duration
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

abstract get_rank()
Return the rank of the current node.

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

abstract prepare(*hdf5*, *simulation_config*)
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.

progress(*step*)
Report simulation progress.

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

simulator
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

start_progress(*duration*)
Start a progress meter.

step_progress(*duration*, *step=1*)

bsb.simulation.cell module

class bsb.simulation.cell.**CellModel**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.component.SimulationComponent*

cell_type

get_node_name()

is_relay()

property relay

bsb.simulation.component module

class bsb.simulation.component.**SimulationComponent**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.helpers.SortableByAfter*

create_after()

get_after()

get_node_name()

classmethod get_ordered(*objects*)

has_after()

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

bsb.simulation.connection module

class bsb.simulation.connection.**ConnectionModel**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.component.SimulationComponent*

get_node_name()

bsb.simulation.device module

class bsb.simulation.device.**DeviceModel**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.component.SimulationComponent*

create_patterns()

get_node_name()

get_pattern(*target*, *cell=None*, *section=None*, *synapse=None*)

get_patterns()
Return the patterns of the device.

implement(*target*, *location*)

initialise_patterns()

validate_specifics()

class bsb.simulation.device.**Patternless**
Bases: object

create_patterns(***kwargs*)

get_pattern(***kwargs*)

bsb.simulation.results module

class bsb.simulation.results.**ClosureRecorder**(*path_func*, *data_func*, *meta_func=None*)
Bases: *bsb.simulation.results.SimulationRecorder*

class bsb.simulation.results.**MultiRecorder**
Bases: *bsb.simulation.results.SimulationRecorder*

get_data(***kwargs*)

multi_collect(**args*, ***kwargs*)

class bsb.simulation.results.**PresetMetaMixin**
Bases: object

get_meta()

class bsb.simulation.results.**PresetPathMixin**
Bases: object

get_path()

class bsb.simulation.results.**SimulationRecorder**
Bases: object

get_data()

get_meta()

get_path()

class bsb.simulation.results.**SimulationResult**
Bases: object

add(*recorder*)

collect()

create_recorder(*path_func*, *data_func*, *meta_func=None*)

safe_collect()

bsb.simulation.targetting module

class bsb.simulation.targetting.**ByIdTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.targetting.NeuronTargetting*

Targetting mechanism (use `"type": "by_id"`) to target all given identifiers.

get_node_name()

get_targets()

targets
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.simulation.targetting.**CellTypeTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.targetting.NeuronTargetting*

Targetting mechanism (use `"type": "cell_type"`) to target all identifiers of certain cell types.

cell_types
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

get_targets()

class bsb.simulation.targetting.**CylindricalTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.targetting.NeuronTargetting*

Targetting mechanism (use `"type": "cylinder"`) to target all cells in a horizontal cylinder (xz circle expanded along y).

boot()

cell_types
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

get_targets()
Target all or certain cells within a cylinder of specified radius.

origin
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

radius
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.simulation.targetting.**NeuronTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: `object`

get_node_name()

get_targets()

type
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.simulation.targetting.**RepresentativesTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.targetting.NeuronTargetting*

Targetting mechanism (use `"type": "representatives"`) to target all identifiers of certain cell types.

cell_types
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

get_targets()

class bsb.simulation.targetting.**SphericalTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.simulation.targetting.NeuronTargetting*

Targetting mechanism (use `"type": "sphere"`) to target all cells in a sphere.

boot()

cell_types
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

get_targets()
Target all or certain cells within a cylinder of specified radius.

origin
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

radius
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.simulation.targetting.**TargetsSections**
Bases: object

target_section(*cell*)

Module contents

bsb.storage package

Subpackages

bsb.storage.engines package

Subpackages

bsb.storage.engines.hdf5 package

Submodules

bsb.storage.engines.hdf5.chunks module

The chunks module provides the tools for the HDF5 engine to store the chunked placement data received from the placement module in seperate datasets to arbitrarily parallelize and scale scaffold models.

The module provides the *ChunkLoader* mixin for *Resource* objects (e.g. PlacementSet, ConnectivitySet) to organize *ChunkedProperty* and *ChunkedCollection* objects within them.

class bsb.storage.engines.hdf5.chunks.**ChunkLoader**
Bases: object

Resource mixin to organize chunked properties and collections within itself.

Parameters

- **properties** – An iterable of functions that construct *ChunkedProperty*.
- **properties** – An iterable of names for constructing *ChunkedCollection*.

Type Iterable

Type Iterable

chunk_context(**chunks*)

clear(*chunks=None*)

clear_chunks()

get_all_chunks()

get_chunk_path(*chunk=None*)
Return the full HDF5 path of a chunk.

Parameters **chunk** (*storage.Chunk*) – Chunk

Returns HDF5 path

Return type str

get_loaded_chunks()

load_chunk(*chunk*)
Add a chunk to read data from when loading properties/collections.

require_chunk(*chunk*, *handle=None*)
Create a chunk if it doesn't exist yet, or do nothing.

set_chunks(*chunks*)

unload_chunk(*chunk*)
Remove a chunk to read data from when loading properties/collections.

class bsb.storage.engines.hdf5.chunks.**ChunkedCollection**(*loader*, *property*)
Bases: object

Chunked collections are stored inside the chunks group of the *ChunkLoader* they belong to. Inside the chunks group another group is created per chunk, inside of which a group exists per collection. Arbitrarily named datasets can be stored inside of this collection.

class bsb.storage.engines.hdf5.chunks.**ChunkedProperty**(*loader*, *property*, *shape*, *dtype*, *insert=None*, *extract=None*)
Bases: object

Chunked properties are stored inside the chunks group of the *ChunkLoader* they belong to. Inside the chunks group another group is created per chunk, inside of which a dataset exists per property.

append(*chunk*, *data*)
Append data to a property chunk. Will create it if it doesn't exist.

Parameters **chunk** (*storage.Chunk*) – Chunk

clear(*chunk*)

load(*raw=False*)

bsb.storage.engines.hdf5.connectivity_set module

class bsb.storage.engines.hdf5.connectivity_set.**ConnectivitySet**(*engine*, *tag*)
Bases: *bsb.storage.engines.hdf5.resource.Resource*, *bsb.storage.interfaces.ConnectivitySet*

Fetches placement data from storage.

Note: Use *Scaffold.get_connectivity_set* to correctly obtain a *ConnectivitySet*.

append_data(*src_chunk*, *dest_chunk*, *src_locs*, *dest_locs*, *handle=None*)

clear()
Override with a method to clear (some chunks of) the placement set

classmethod create(*engine*, *pre_type*, *post_type*, *tag=None*)
Create the structure for this connectivity set in the HDF5 file. Connectivity sets are stored under `/connectivity/<tag>`.

static exists(*engine*, *tag*, *handle=None*)
Override with a method to check existence of the placement set

classmethod get_tags(*engine*)

muxed_append(*pre_set*, *post_set*, *src_locs*, *dest_locs*)

classmethod require(*engine*, *pre_type*, *post_type*, *tag=None*)
Can be overridden with a method to make sure the placement set exists. The default implementation uses the class's `exists` and `create` methods.

bsb.storage.engines.hdf5.file_store module

class bsb.storage.engines.hdf5.file_store.**FileStore**(*engine*)
Bases: *bsb.storage.engines.hdf5.resource.Resource*, *bsb.storage.interfaces.FileStore*

all()
Return all ids and associated metadata in the file store.

load(*id*)
Load the content of an object in the file store.

Parameters **id** (*str*) – id of the content to be loaded.

Returns The content of the stored object

Return type str

Raises **FileNotFoundError** – The given id doesn't exist in the file store.

load_active_config()
Load the active configuration stored in the file store.

Returns The active configuration

Return type *Configuration*

Raises **Exception** – When there's no active configuration in the file store.

remove(*id*)
Remove the content of an object in the file store.

Parameters **id** (*str*) – id of the content to be removed.

Raises **FileNotFoundError** – The given id doesn't exist in the file store.

store(*content*, *meta=None*, *id=None*)
Store content in the file store.

Parameters

- **content** (*str*) – Content to be stored
- **id** (*str*) – Optional specific id for the content to be stored under.
- **meta** (*dict*) – Metadata for the content

Returns The id the content was stored under

Return type str

store_active_config(*config*)
Store configuration in the file store and mark it as the active configuration of the stored network.

Parameters **config** (*Configuration*) – Configuration to be stored

Returns The id the config was stored under

Return type str

stream(*id*, *binary=False*)
Stream the content of an object in the file store.

Parameters

- **id** (*str*) – id of the content to be streamed.
- **binary** (*bool*) – Whether to return file in text or bytes mode.

Returns A readable file-like object of the content.

Raises **FileNotFoundError** – The given id doesn't exist in the file store.

bsb.storage.engines.hdf5.morphology_repository module

class bsb.storage.engines.hdf5.morphology_repository.**MorphologyRepository**(*engine*)
Bases: *bsb.storage.engines.hdf5.resource.Resource*, *bsb.storage.interfaces.MorphologyRepository*

all()

get_meta(*name*)

has(*name*)

load(*name*)

preload(*name*)

remove(*name*)

save(*name*, *morphology*, *overwrite=False*)

select(**selectors*)

bsb.storage.engines.hdf5.placement_set module

class bsb.storage.engines.hdf5.placement_set.**PlacementSet**(*engine*, *cell_type*)
Bases: *bsb.storage.engines.hdf5.resource.Resource*, *bsb.storage.engines.hdf5.chunks.ChunkLoader*, *bsb.storage.interfaces.PlacementSet*

Fetches placement data from storage.

Note: Use *Scaffold.get_placement_set* to correctly obtain a PlacementSet.

---

append_additional(*name*, *chunk*, *data*)

append_data(*chunk*, *positions=None*, *morphologies=None*, *rotations=None*, *additional=None*, *count=None*)
Append data to the PlacementSet.

Parameters

- **positions** (numpy.ndarray) – Cell positions
- **rotations** (numpy.ndarray) – Cell rotations
- **morphologies** (class:~.*storage.interfaces.MorphologySet*) – The associated MorphologySet.

append_entities(*chunk*, *count*, *additional=None*)

classmethod create(*engine*, *cell_type*)
Create the structure for this placement set in the HDF5 file. Placement sets are stored under `/placement/<tag>`.

static exists(*engine*, *cell_type*)
Override with a method to check existence of the placement set

load_morphologies()
Load the cell morphologies.

Raises DatasetNotFoundError when the morphology data is not found.

load_positions()
Load the cell positions.

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

load_rotations()
Load the cell rotations.

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

classmethod require(*engine*, *cell_type*)
Can be overridden with a method to make sure the placement set exists. The default implementation uses the class's `exists` and `create` methods.

bsb.storage.engines.hdf5.resource module

class bsb.storage.engines.hdf5.resource.**Resource**(*engine*, *path*)
Bases: object

append(*new_data*, *dtype=<class 'float'>*)

property attributes

create(*data*, **args*, ***kwargs*)

exists()

get_attribute(*name*)

get_dataset(*selector=()*)

keys()

remove()

require(*handle*)

property shape

unmap(*selector=()*, *mapping=<function Resource.<lambda>>*, *data=None*)

unmap_one(*data*, *mapping=None*)

Module contents

class bsb.storage.engines.hdf5.**HDF5Engine**(*root*)
Bases: *bsb.storage.interfaces.Engine*

clear_connectivity()

clear_placement()

create()

exists()

move(*new_root*)

remove()

class bsb.storage.engines.hdf5.**StorageNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.config.nodes.StorageNode*

get_node_name()

root
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

bsb.storage.engines.in_memory package

Module contents

Module contents

Submodules

bsb.storage.interfaces module

class bsb.storage.interfaces.**ConnectivitySet**(*engine*)
Bases: *bsb.storage.interfaces.Interface*

abstract clear(*chunks=None*)
Override with a method to clear (some chunks of) the placement set

abstract classmethod create(*engine*, *tag*)
Override with a method to create the placement set.

abstract static exists(*self*, *engine*, *tag*)
Override with a method to check existence of the placement set

abstract classmethod get_tags(*engine*)

require(*engine*, *tag*)
Can be overridden with a method to make sure the placement set exists. The default implementation uses the class's `exists` and `create` methods.

class bsb.storage.interfaces.**Engine**(*root*)
Bases: *bsb.storage.interfaces.Interface*

abstract clear_connectivity()

abstract clear_placement()

abstract create()

abstract exists()

property format

abstract move(*new_root*)

abstract remove()

class bsb.storage.interfaces.**FileStore**(*engine*)
Bases: *bsb.storage.interfaces.Interface*

Interface for the storage and retrieval of files essential to the network description.

abstract all()
Return all ids and associated metadata in the file store.

abstract load(*id*)
Load the content of an object in the file store.

Parameters **id** (*str*) – id of the content to be loaded.

Returns The content of the stored object

Return type str

Raises **FileNotFoundError** – The given id doesn't exist in the file store.

abstract load_active_config()
Load the active configuration stored in the file store.

Returns The active configuration

Return type *Configuration*

Raises **Exception** – When there's no active configuration in the file store.

abstract remove(*id*)
Remove the content of an object in the file store.

Parameters **id** (*str*) – id of the content to be removed.

Raises **FileNotFoundError** – The given id doesn't exist in the file store.

abstract store(*content*, *id=None*, *meta=None*)
Store content in the file store.

Parameters

- **content** (*str*) – Content to be stored
- **id** (*str*) – Optional specific id for the content to be stored under.
- **meta** (*dict*) – Metadata for the content

Returns The id the content was stored under

Return type str

abstract store_active_config(*config*)
Store configuration in the file store and mark it as the active configuration of the stored network.

Parameters **config** (*Configuration*) – Configuration to be stored

Returns The id the config was stored under

Return type str

abstract stream(*id*, *binary=False*)
Stream the content of an object in the file store.

Parameters

- **id** (*str*) – id of the content to be streamed.
- **binary** (*bool*) – Whether to return file in text or bytes mode.

Returns A readable file-like object of the content.

Raises **FileNotFoundError** – The given id doesn't exist in the file store.

class bsb.storage.interfaces.**Interface**(*engine*)
Bases: abc.ABC

class bsb.storage.interfaces.**MorphologyRepository**(*engine*)
Bases: *bsb.storage.interfaces.Interface*

abstract all()

abstract get_meta(*name*)

abstract has(*selector*)

import_arb(*morphology*, *labels*, *name*, *overwrite=False*, *centering=True*)

import_asc(*file*, *name*, *overwrite=False*)
Import and store .asc file contents as a morphology in the repository.

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

abstract load(*selector*)

abstract preload(*selector*)

abstract save(*selector*)

abstract select(*selector*)

class bsb.storage.interfaces.**NetworkDescription**(*engine*)
Bases: *bsb.storage.interfaces.Interface*

class bsb.storage.interfaces.**PlacementSet**(*engine*, *cell_type*)
Bases: *bsb.storage.interfaces.Interface*

abstract append_additional(*name*, *chunk*, *data*)

abstract append_data(*chunk*, *positions=None*, *morphologies=None*, *rotations=None*, *additional=None*)

property cell_type

abstract clear(*chunks=None*)
Override with a method to clear (some chunks of) the placement set

abstract classmethod create(*engine*, *type*)
Override with a method to create the placement set.

abstract static exists(*self*, *engine*, *type*)
Override with a method to check existence of the placement set

abstract get_all_chunks()

load_box_tree(*cache=None*)

load_boxes(*cache=None*, *itr=True*)

abstract load_morphologies()
Return a *MorphologySet* associated to the cells.

Returns Set of morphologies

Return type *MorphologySet*

abstract load_positions()
Return a dataset of cell positions.

abstract load_rotations()
Return a *RotationSet*.

require(*engine*, *type*)
Can be overridden with a method to make sure the placement set exists. The default implementation uses the class's `exists` and `create` methods.

property tag

class bsb.storage.interfaces.**StoredMorphology**(*name*, *loader*, *meta*)
Bases: `object`

cached_load()

get_meta()

load()

Module contents

This module imports all supported storage engines, objects that read and write data, which are present as subfolders of the *engine* folder, and provides them transparently to the user, as a part of the *Storage* factory class. The module scans the *storage.interfaces* module for any class that inherits from *Interface* to collect all Feature Interfaces and then scans the `storage.engines.*` submodules for any class that provides an implementation of those features.

These features, because they all follow the same interface can then be passed on to consumers and can be used independent of the underlying storage engine, which is the end goal of this module.

class bsb.storage.**Chunk**(*chunk*, *chunk_size*)
Bases: `numpy.ndarray`

Chunk identifier, consisting of chunk coordinates and size.

property box

property dimensions

classmethod from_id(*id*, *size*)

property id

property ldc

property mdc

class bsb.storage.**NotSupported**(*engine*, *operation*)
Bases: `object`

Utility class that throws a `NotSupported` error when it is used. This is the default "implementation" of every storage feature that isn't provided by an engine.

class bsb.storage.**Storage**(*engine*, *root*, *comm=None*, *master=0*)
Bases: `object`

Factory class that produces all of the features and shims the functionality of the underlying engine.

assert_support(*feature*)

clear_connectivity()

clear_placement(*scaffold=None*)

create()
Create the minimal requirements at the root for other features to function and for the existence check to pass.

exists()
Check whether the storage exists at the root.

property files

property format

get_connectivity_set(*tag*)
Get a connection set.

Parameters **tag** (*str*) – Connection tag

Returns *ConnectivitySet*

get_connectivity_sets()
Return a ConnectivitySet for the given type.

Parameters **type** (*CellType*) – Specific cell type.

Returns *ConnectivitySet*

get_placement_set(*type*, *chunks=None*)
Return a PlacementSet for the given type.

Parameters

- **type** (*CellType*) – Specific cell type.
- **chunks** (*list[tuple[float, float, float]]*) – Optionally load a specific list of chunks.

Returns *PlacementSet*

init(*scaffold*)
Initialize the storage to be ready for use by the specified scaffold.

init_placement(*scaffold*)

is_master()

load()
Load a scaffold from the storage.

Returns *Scaffold*

load_active_config()
Load the configuration object from the storage.

Returns *Configuration*

property morphologies

move(*new_root*)
Move the storage to a new root.

property preexisted

remove()
Remove the storage and all data contained within. This is an irreversible destructive action!

renew(*scaffold*)
Remove and recreate an empty storage container for a scaffold.

require_connectivity_set(*tag*, *pre=None*, *post=None*)
Get a connection set.

Parameters **tag** (*str*) – Connection tag

Returns *ConnectivitySet*

property root

store_active_config(*config*)
Store a configuration object in the storage.

supports(*feature*)

bsb.storage.**get_engines**()
Get a dictionary of all available storage engines.

bsb.storage.**view_support**(*engine=None*)
Return which storage engines support which features.

bsb.topology package

Submodules

bsb.topology.partition module

Module for the Partition configuration nodes and its dependencies.

class bsb.topology.partition.**Layer**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.topology.partition.Partition*

get_dependencies()
Return other partitions or regions that need to be laid out before this.

get_node_name()

layout(*boundaries*)

stack_index
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

thickness
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

xz_center
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

xz_scale
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.topology.partition.**Partition**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

chunk_to_voxels(*chunk*)
Return an approximation of this partition intersected with a chunk as a list of voxels.

Default implementation creates a parallellepepid intersection between the LDC, MDC and chunk boundaries.

get_node_name()

layout(*boundaries*)

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

region

surface(*chunk=None*)

to_chunks(*chunk_size*)

type
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

volume(*chunk=None*)

class bsb.topology.partition.**Voxels**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.topology.partition.Partition*

chunk_to_voxels(*chunk*)
Return an approximation of this partition intersected with a chunk as a list of voxels.

Default implementation creates a parallellepepid intersection between the LDC, MDC and chunk boundaries.

get_node_name()

layout(*boundaries*)

to_chunks(*chunk_size*)

voxels
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

property voxelset

bsb.topology.region module

Module for the Region types.

class bsb.topology.region.**Region**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

Base region.

When arranging will simply call arrange/layout on its children but won't cause any changes itself.

arrange(*boundaries*)

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_dependencies()

get_node_name()

name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

offset
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

partitions

class bsb.topology.region.**RegionGroup**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.topology.region.Region*

origin = None

class bsb.topology.region.**Stack**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.topology.region.Region*

Stack components on top of each other based on their `stack_index` and adjust its own height accordingly.

arrange(*boundary*)

axis
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

Module contents

Topology module

class bsb.topology.**Boundary**(*ldc*, *mdc*)
Bases: `object`

Base boundary class describing a region between a Least Dominant Corner (ldc; lowest value in each dimension) and Most Dominant Corner (mdc; highest value in each dimension).

All child boundary classes must be able to describe themselves based only on these 2 values. For example a sphere normally described as a center and radius would take the corners of the tangent parallelopiped instead.

copy()
Copy this boundary to a new instance.

property depth

property dimensions

property height

offset(*offset*)

property width

property x

property y

property z

class bsb.topology.**BoxBoundary**(*point*, *dimensions*, *centered=False*)
Bases: *bsb.topology.Boundary*

Boundary class describing a Box starting from or centered around a point with certain dimensions.

copy()
Copy this boundary to a new instance.

property point

bsb.topology.**create_topology**(*regions*)
Create a topology from group of regions. Will check for root regions, if there's not exactly 1 root region a *RegionGroup* will be created as new root.

Parameters **regions** (*Iterable*) – Any iterable of regions.

bsb.topology.**get_partitions**(*regions*)
Get all of the partitions belonging to the group of regions and their subregions.

Parameters **regions** (*Iterable*) – Any iterable of regions.

bsb.topology.**get_root_regions**(*regions*)
Get all of the root regions, not belonging to any other region in the given group.

Parameters **regions** (*Iterable*) – Any iterable of regions.

21.1.2 Submodules

21.1.3 bsb.core module

class bsb.core.**ReportListener**(*scaffold*, *file*)
Bases: object

class bsb.core.**Scaffold**(*config=None*, *storage=None*, *clear=False*)
Bases: object

This is the main object of the bsb package, it represents a network and puts together all the pieces that make up the model description such as the *Configuration* with the technical side like the *Storage*.

property after_connectivity

property after_placement

attr = 'simulations'

property cell_types

clear()
Clears the storage. This deletes any existing network data!

clear_connectivity()
Clears the connectivity storage.

clear_placement()
Clears the placement storage.

compile(*skip_placement=False*, *skip_connectivity=False*, *skip_after_placement=False*, *skip_after_connectivity=False*, *only=None*, *skip=None*, *clear=False*, *append=False*, *redo=False*, *force=False*)
Run reconstruction steps in the scaffold sequence to obtain a full network.

property configuration

connect_cells()

property connectivity

create_adapter(*simulation_name*)
Create an adapter for a simulation. Adapters are the objects that translate scaffold data into simulator data.

create_entities(*cell_type*, *count*)
Create entities in the simulation space.

Entities are different from cells because they have no positional data and don't influence the placement step. They do have a representation in the connection and simulation step.

Parameters

- **cell_type** (*CellType*) – The cell type of the entities
- **count** (*int*) – Number of entities to place

Todo Allow *additional* data for entities

property files

get_cell_types()
Return a list of all cell types in the network.

get_connectivity(*anywhere=None*, *presynaptic=None*, *postsynaptic=None*, *skip=None*, *only=None*)

get_connectivity_set(*tag=None*, *pre=None*, *post=None*)
Return a connectivity set from the output formatter.

Parameters **tag** (*str*) – Unique identifier of the connectivity set in the output formatter

Returns A connectivity set

Return type *ConnectivitySet*

get_connectivity_sets()
Return all connectivity sets from the output formatter.

Parameters **tag** (*str*) – Unique identifier of the connectivity set in the output formatter

Returns A connectivity set

Return type *ConnectivitySet*

get_labels(*pattern=None*)
Retrieve the set of labels that match a label pattern. Currently only exact matches or strings ending in a wildcard are supported:

```
# Will return only ["label-53"] if it is known to the scaffold.
labels = scaffold.get_labels("label-53")
# Might return multiple labels such as ["label-53", "label-01", ...]
labels = scaffold.get_labels("label-*")
```

Parameters **pattern** (*str*) – An exact match or pattern ending in a wildcard (*) character.

Returns All labels matching the pattern

Return type list

get_placement(*cell_types=None*, *skip=None*, *only=None*)

get_placement_of(**cell_types*)
Find all of the placement strategies that given certain cell types.

Parameters **cell_types** (Union[*CellType*, str]) – Cell types (or their names) of interest.

get_placement_set(*type*, *chunks=None*)
Return a cell type's placement set from the output formatter.

Parameters **tag** (*str*) – Unique identifier of the placement set in the storage

Returns A placement set

Return type *PlacementSet*

get_simulation(*simulation_name*)
Retrieve the default single-instance adapter for a simulation.

label_cells(*ids*, *label*)
Store labels for the given cells. Labels can be used to identify subsets of cells.

Parameters **ids** (*Iterable*) – global identifiers of the cells that need to be labelled.

merge(*other*, *label=None*)

property morphologies

property network

property partitions

place_cells(*cell_type*, *positions*, *morphologies=None*, *rotations=None*, *additional=None*, *chunk=None*)
Place cells inside of the scaffold

```
# Add one granule cell at position 0, 0, 0
cell_type = scaffold.get_cell_type("granule_cell")
scaffold.place_cells(cell_type, cell_type.layer_instance, [[0., 0., 0.]])
```

Parameters

- **cell_type** (*CellType*) – The type of the cells to place.
- **positions** (Any *np.concatenate* type of shape (N, 3).) – A collection of xyz positions to place the cells on.

property placement

prepare_simulation(*simulation_name*)
Retrieve and prepare the default single-instance adapter for a simulation.

property regions

require_connectivity_set(*pre*, *post*, *tag=None*)

resize(*x=None*, *y=None*, *z=None*)
Updates the topology boundary indicators. Use before placement, updates only the abstract topology tree, does not rescale, prune or otherwise alter already existing placement data.

run_after_connectivity()
Run after placement hooks.

run_after_placement()
Run after placement hooks.

run_connectivity(*strategies=None*, *DEBUG=True*)
Run connection strategies.

run_placement(*strategies=None*, *DEBUG=True*)
Run placement strategies.

run_placement_strategy(*strategy*)
Run a single placement strategy.

run_simulation(*simulation_name*, *quit=False*)
Run a simulation starting from the default single-instance adapter.

Parameters **simulation_name** (*str*) – Name of the simulation in the configuration.

property simulations

property storage

property storage_cfg

bsb.core.**from_hdf5**(*file*)
Generate a *core.Scaffold* from an HDF5 file.

Parameters **file** – Path to the HDF5 file.

Returns A scaffold object

Return type *Scaffold*

21.1.4 bsb.exceptions module

exception bsb.exceptions.**AdapterError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**AllenApiError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.GatewayError*

exception bsb.exceptions.**ArborError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.AdapterError*

exception bsb.exceptions.**AttributeMissingError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ResourceError*

exception bsb.exceptions.**CLIError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**CastConfigurationError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**CastError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**ChunkError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.PlacementError*

exception bsb.exceptions.**CircularMorphologyError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**ClassError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**ClassMapMissingError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.DynamicClassError*

exception bsb.exceptions.**CommandError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CLIError*

exception bsb.exceptions.**CompartmentError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**CompilationError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**ConfigTemplateNotFoundError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CLIError*

exception bsb.exceptions.**ConfigurationError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**ConfigurationFormatError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**ConfigurationWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**ConnectivityError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**ConnectivityWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**ContinuityError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.PlacementError*

exception bsb.exceptions.**CriticalDataWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**DataNotFoundError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ResourceError*

exception bsb.exceptions.**DataNotProvidedError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**DatasetNotFoundError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ResourceError*

exception bsb.exceptions.**DeviceConnectionError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NeuronError*

exception bsb.exceptions.**DistributionCastError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CastError*

exception bsb.exceptions.**DryrunError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CLIError*

exception bsb.exceptions.**DynamicClassError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**DynamicClassInheritanceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.DynamicClassError*

exception bsb.exceptions.**DynamicClassNotFoundError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.DynamicClassError*

exception bsb.exceptions.**EmptySelectionError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**EmptyVoxelSetError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.VoxelSetError*

exception bsb.exceptions.**ExternalSourceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConnectivityError*

exception bsb.exceptions.**FiberTransformError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConnectivityError*

exception bsb.exceptions.**GatewayError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**IncompleteExternalMapError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ExternalSourceError*

exception bsb.exceptions.**IncompleteMorphologyError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**IndicatorError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**InputError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CLIError*

exception bsb.exceptions.**IntersectionDataNotFoundError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.DatasetNotFoundError*

exception bsb.exceptions.**InvalidReferenceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.TypeHandlingError*

exception bsb.exceptions.**JsonImportError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.JsonParseError*

exception bsb.exceptions.**JsonParseError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ParserError*

exception bsb.exceptions.**JsonReferenceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.JsonParseError*

exception bsb.exceptions.**KernelLockedError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NestError*

exception bsb.exceptions.**KernelWarning**
Bases: *bsb.exceptions.SimulationWarning*

exception bsb.exceptions.**LayoutError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.TopologyError*

exception bsb.exceptions.**MissingBoundaryError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.LayoutError*

exception bsb.exceptions.**MissingMorphologyError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**MissingSourceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ExternalSourceError*

exception bsb.exceptions.**MorphologyDataError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**MorphologyError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**MorphologyRepositoryError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.MorphologyError*

exception bsb.exceptions.**MorphologyWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**NestError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.AdapterError*

exception bsb.exceptions.**NestKernelError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NestError*

exception bsb.exceptions.**NestModelError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NestError*

exception bsb.exceptions.**NestModuleError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NestKernelError*

exception bsb.exceptions.**NeuronError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.AdapterError*

exception bsb.exceptions.**NoReferenceAttributeSignal**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ReferenceError*

exception bsb.exceptions.**NodeNotFoundError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**NoneReferenceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.TypeHandlingError*

exception bsb.exceptions.**OptionError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**OrderError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**ParallelIntegrityError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.AdapterError*

exception bsb.exceptions.**ParserError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**PlacementError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**PlacementRelationError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.PlacementError*

exception bsb.exceptions.**PlacementWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**PluginError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**QuiverFieldError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.FiberTransformError*

exception bsb.exceptions.**QuiverFieldWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**ReadOnlyOptionError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.OptionError*

exception bsb.exceptions.**ReceptorSpecificationError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NestError*

exception bsb.exceptions.**RedoError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CompilationError*

exception bsb.exceptions.**ReferenceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**RelayError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NeuronError*

exception bsb.exceptions.**RepositoryWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**RequirementError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**ResourceError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**ScaffoldError**(**args*, ***kwargs*)
Bases: errr.exception.DetailedException

exception bsb.exceptions.**ScaffoldWarning**
Bases: UserWarning

exception bsb.exceptions.**SimulationWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**SourceQualityError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ExternalSourceError*

exception bsb.exceptions.**SpatialDimensionError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**SuffixTakenError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NestError*

exception bsb.exceptions.**TopologyError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**TransmitterError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.NeuronError*

exception bsb.exceptions.**TreeError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**TypeHandlingError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

exception bsb.exceptions.**UnfitClassCastError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CastError*

exception bsb.exceptions.**UnknownConfigAttrError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConfigurationError*

exception bsb.exceptions.**UnknownGIDError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ConnectivityError*

exception bsb.exceptions.**UnknownStorageEngineError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ResourceError*

exception bsb.exceptions.**UnmanagedPartitionError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.TopologyError*

exception bsb.exceptions.**UnresolvedClassCastError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.CastError*

exception bsb.exceptions.**UserUserDeprecationWarning**
Bases: *bsb.exceptions.ScaffoldWarning*

exception bsb.exceptions.**VoxelSetError**(**args*, ***kwargs*)
Bases: *bsb.exceptions.ScaffoldError*

21.1.5 bsb.helpers module

class bsb.helpers.**SortableByAfter**
Bases: object

add_after(*after_item*)

abstract create_after()

abstract get_after()

abstract get_ordered(*objects*)

abstract has_after()

is_after_satisfied(*objects*)
Determine whether the `after` specification of this object is met. Any objects appearing in `self.after` need to occur in `objects` before the object.

Parameters **objects** (*list*) – Proposed order for which the after condition is checked.

classmethod resolve_order(*objects*)
Orders a given dictionary of objects by the class's default mechanism and then apply the *after* attribute for further restrictions.

satisfy_after(*objects*)
Given an array of objects, place this object after all of the objects specified in the `after` condition. If objects in the after condition are missing from the given array this object is placed at the end of the array. Modifies the *objects* array in place.

bsb.helpers.**get_qualified_class_name**(*x*)

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.

bsb.helpers.**suppress_stdout**()

21.1.6 bsb.networks module

class bsb.networks.**Branch**(*compartments*, *orientation*, *parent=None*, *ordered=True*)
Bases: object

add_branch(*branch*)

append(*compartment*)

detach(*compartment*)

interpolate(*resolution*)

split(*compartment*, *n*)
Split the compartment in n pieces and make those a part of the branch.

This function stores a link to the original compartment in the partial compartments in the attribute *_original*.

Parameters

- **compartment** – The compartment to split.
- **n** (*int*) – The amount of pieces to split the compartment into.

voxelize(*position*, *bounding_box*, *voxel_tree*, *map*, *voxel_list*)

walk(*start=None*)

class bsb.networks.**FiberMorphology**(*compartments*, *rotation*)
Bases: object

flatten(*branches=None*)

bsb.networks.**all_depth_first_branches**(*adjacency_list*)

bsb.networks.**create_root_branched_network**(*compartments*, *orientation*)

bsb.networks.**depth_first_branches**(*adjacency_list*, *node=0*, *return_visited=False*)

bsb.networks.**get_branch_points**(*branch_list*)

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.

21.1.7 bsb.option module

This module contains the classes required to construct options.

class bsb.option.**BsbOption**(*positional=False*)
Bases: object

Base option class. Can be subclassed to create new options.

add_to_parser(*parser*, *level*)
Register this option into an `argparse` parser.

get(*prio=None*)
Get the option's value. Cascades the script, cli, env & default descriptors together.

Returns option value

get_cli_tags()
Return the `argparse` positional arguments from the tags.

Returns `-x` or `--xxx` for each CLI tag.

Return type list

get_default()
Override to specify the default value of the option.

classmethod register()
Register this option class into the *bsb.options* module.

unregister()
Remove this option class from the *bsb.options* module, not part of the public API as removing options is undefined behavior but useful for testing.

class bsb.option.**CLIOptionDescriptor**(**tags*)
Bases: *bsb.option.OptionDescriptor*

Descriptor that retrieves its value from the given CLI command arguments.

slug = 'cli'

class bsb.option.**EnvOptionDescriptor**(**args*, *flag=False*)
Bases: *bsb.option.OptionDescriptor*

Descriptor that retrieves its value from the environment variables.

is_set(*instance*)

slug = 'env'

class bsb.option.**OptionDescriptor**(**tags*)
Bases: object

Base option property descriptor. Can be inherited from to create a cascading property such as the default CLI, env & script descriptors.

is_set(*instance*)

class bsb.option.**ProjectOptionDescriptor**(**tags*)
Bases: *bsb.option.OptionDescriptor*

Descriptor that retrieves and stores values in the *pyproject.toml* file. Traverses up the filesystem tree until one is found.

is_set(*instance*)

slug = 'project'

class bsb.option.**ScriptOptionDescriptor**(**tags*)
Bases: *bsb.option.OptionDescriptor*

Descriptor that retrieves and sets its value from/to the *bsb.options* module.

is_set(*instance*)

slug = 'script'

21.1.8 bsb.options module

This module contains the global options.

You can set options at the `script` level (which superceeds all other levels such as environment variables or project settings).

```
import bsb.options
from bsb.option import BsbOption

class MyOption(BsbOption, cli=("my_setting",), env=("MY_SETTING",), script=("my_setting",
↪ "my_alias")):
    def get_default(self):
        return 4

# Register the option into the `bsb.options` module
MyOption.register()

assert bsb.options.my_setting == 4
bsb.options.my_alias = 6
assert bsb.options.my_setting == 6
```

Your `MyOption` will also be available on all CLI commands as `--my_setting` and will be read from the `MY_SETTING` environment variable.

bsb.options.**get**(*tag*, *prio=None*)
Retrieve the cascaded value for an option.

Parameters

- **tag** (*str*) – Name the option is registered with.
- **prio** (*str*) – Give priority to a type of value. Can be any of 'script', 'cli', 'project', 'env'.

Returns (Possibly prioritized) value of the option.

Return type Any

bsb.options.**get_module_option**(*tag*)
Get the value of a module option. Does the same thing as `getattr(options, tag)`

Parameters **tag** (*str*) – Name the option is registered with in the module.

bsb.options.**get_option**(*name*)
Return an option

Parameters **name** (*str*) – Name of the option to look for.

Returns The option singleton of that name.

Return type dict[str, *bsb.option.BsbOption*]

bsb.options.**get_option_classes**()
Return all of the classes that are used to create singleton options from. Useful to access the option descriptors rather than the option values.

Returns The classes of all the installed options by name.

Return type dict[str, *bsb.option.BsbOption*]

bsb.options.**get_options**()
Get all the registered option singletons.

bsb.options.**get_project_option**(*tag*)
Find a project option

Parameters **tag** (*str*) – dot-separated path of the option. e.g. `networks.config_link`.

Returns Project option instance

Return type *option.BsbOption*

bsb.options.**is_module_option_set**(*tag*)
Check if a module option was set.

Parameters **tag** (*str*) – Name the option is registered with in the module.

Returns Whether the option was ever set from the module

Return type bool

bsb.options.**read**(*tag=None*)
Read an option value from the project settings. Returns all project settings if tag is omitted.

Parameters **tag** (*str*) – Dot-separated path of the project option

Returns Value for the project option

Return type Any

bsb.options.**register_option**(*name*, *option*)
Register an option as a global BSB option. Options that are installed by the plugin system are automatically registered on import of the BSB.

Parameters

- **name** (*str*) – Name for the option, used to store and retrieve its singleton.

- **option** (*option.BsbOption*) – Option instance, to be used as a singleton.

bsb.options.**reset_module_option**(*tag*)

bsb.options.**set_module_option**(*tag*, *value*)
Set the value of a module option. Does the same thing as `setattr(options, tag, value)`.

Parameters

- **tag** (*str*) – Name the option is registered with in the module.
- **value** (*Any*) – New module value for the option

bsb.options.**store**(*tag*, *value*)
Store an option value permanently in the project settings.

Parameters

- **tag** (*str*) – Dot-separated path of the project option
- **value** (*Any*) – New value for the project option

bsb.options.**unregister_option**(*option*)
Unregister a globally registered option. Also removes its script and project parts.

Parameters **option** (*option.BsbOption*) – Option singleton, to be removed.

21.1.9 bsb.particles module

class bsb.particles.**AdaptiveNeighbourhood**(*track_displaced=False*, *scaffold=None*)
Bases: *bsb.particles.ParticleSystem*

find_neighbourhood(*particle*)

class bsb.particles.**LargeParticleSystem**
Bases: *bsb.particles.ParticleSystem*

fill()

placing()

solve_collisions()

class bsb.particles.**Neighbourhood**(*epicenter*, *neighbours*, *neighbour_radius*, *partners*, *partner_radius*)
Bases: object

colliding()

get_overlap()

class bsb.particles.**Particle**(*radius*, *position*)
Bases: object

displace()

displace_by(*other*)

static get_displacement_force(*radius*, *distance*)

reset_displacement()

class bsb.particles.**ParticleSystem**(*track_displaced=False*, *scaffold=None*)
Bases: object

add_particle(*radius*, *position*, *type=None*)

add_particles(*radius*, *positions*, *type=None*)

deintersect(*nearest_neighbours=None*)

fill(*voxels*, *particles*)

find_colliding_particles(*freeze=False*)

find_neighbourhood(*particle*)

freeze()

get_packing_factor(*particles=None*, *volume=None*)

property positions

prune(*at_risk_particles=None*, *voxels=None*)
Remove particles that have been moved outside of the bounds of the voxels.

Parameters

- **at_risk_particles** (numpy.ndarray) – Subset of particles that might've been moved and might need to be moved, if omitted check all particles.
- **voxels** – A subset of the voxels that the particles have to be in bounds of, if omitted all voxels are used.

remove_particles(*particles_id*)

resolve_neighbourhood(*neighbourhood*)

solve_collisions()

class bsb.particles.**ParticleVoxel**(*origin*, *dimensions*)
Bases: object

class bsb.particles.**SmallestNeighbourhood**(*track_displaced=False*, *scaffold=None*)
Bases: *bsb.particles.ParticleSystem*

find_neighbourhood(*particle*)

bsb.particles.**distance**(*a*, *b*)

bsb.particles.**get_particle_trace**(*particle*)

bsb.particles.**get_particles_trace**(*particles*, *dimensions=3*, *axes={'x': 0, 'y': 1, 'z': 2}*, ***kwargs*)

bsb.particles.**plot_detailed_system**(*system*)

bsb.particles.**plot_particle_system**(*system*)

bsb.particles.**sphere_volume**(*radius*)

21.1.10 bsb.plugins module

Plugins module. Uses pkg_resources to detect installed plugins and loads them as categories.

bsb.plugins.**discover**(*category*, **args*, ***kwargs*)
Discover all plugins for a given category.

Parameters **category** (*str*) – Plugin category (e.g. adapters to load all bsb.adapters)

Returns Loaded plugins by name.

Return type dict

21.1.11 bsb.postprocessing module

class bsb.postprocessing.**AscendingAxonLengths**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.postprocessing.PostProcessingHook*

after_placement()

class bsb.postprocessing.**BidirectionalContact**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.postprocessing.PostProcessingHook*

after_connectivity()

class bsb.postprocessing.**DCNRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.postprocessing.PostProcessingHook*

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

after_placement()

class bsb.postprocessing.**LabelMicrozones**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.postprocessing.PostProcessingHook*

after_placement()

get_node_name()

label_satellites(*planet_type*, *labels*)

targets
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.postprocessing.**MissingAxon**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.postprocessing.PostProcessingHook*

after_connectivity()

validate()

class bsb.postprocessing.**PostProcessingHook**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: object

after_connectivity()

after_placement()

cls
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

get_node_name()

class bsb.postprocessing.**SpoofDetails**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.postprocessing.PostProcessingHook*

Create fake morphological intersections between already connected non-detailed connection types.

after_connectivity()

casts = {'postsynaptic': <class 'str'>, 'presynaptic': <class 'str'>}

spoof_connections(*connection_type*, *connectivity_matrix*)

21.1.12 bsb.reporting module

bsb.reporting.**get_report_file**()
Return the report file of the scaffold package.

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

Parameters

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

bsb.reporting.**set_report_file**(*v*)
Set a file to which the scaffold package should report instead of stdout.

bsb.reporting.**warn**(*message*, *category=None*, *stacklevel=2*)
Send a warning.

Parameters

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

bsb.reporting.**wrap_writer**(*stream*, *writer*)

21.1.13 bsb.statistics module

class bsb.statistics.**CellsPlaced**(*scaffold*)
Bases: object

class bsb.statistics.**Statistics**(*scaffold*)
Bases: object

property connections

21.1.14 bsb.trees module

class bsb.trees.**BoxRTree**(*boxes*)
Bases: *bsb.trees.BoxTreeInterface*

query(*boxes*)

class bsb.trees.**BoxTree**(*boxes*)
Bases: *bsb.trees.BoxRTree*

class bsb.trees.**BoxTreeInterface**
Bases: abc.ABC

abstract query(*boxes*)

21.1.15 bsb.voxels module

class bsb.voxels.**AllenStructureLoader**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.voxels.NrrdVoxelLoader*

find_structure(*id*)

get_node_name()

get_structure_mask(*find*)

get_structure_mask_condition(*find*)

mask_only
alias of *bsb.voxels.AllenStructureLoader*

mask_source
alias of *bsb.voxels.AllenStructureLoader*

source
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

sources
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

struct_id
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

struct_name
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.voxels.**NrrdVoxelLoader**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: *bsb.voxels.VoxelLoader*

get_node_name()

get_voxelset()

keys
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

mask_only
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

mask_source
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

mask_value
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

source
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

sources
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

sparse
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

strict
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

voxel_size
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.voxels.**VoxelData**(*data*, *keys=None*)
Bases: numpy.ndarray

Chunk identifier, consisting of chunk coordinates and size.

copy()
Return a new copy of the voxel data

property keys
Returns the keys, or column labels, associated to each data column.

class bsb.voxels.**VoxelLoader**(**args*, *_parent=None*, *_key=None*, ***kwargs*)
Bases: abc.ABC

get_node_name()

abstract get_voxelset()

type
Base implementation of all the different configuration attributes. Call the factory function *attr()* instead.

class bsb.voxels.**VoxelSet**(*voxels*, *size*, *data=None*, *data_keys=None*, *irregular=False*)
Bases: object

as_boxes(*cache=False*)

as_boxtree(*cache=False*)

as_spatial_coords(*copy=True*)

property bounds
The minimum and maximum coordinates of this set.

Return type tuple[numpy.ndarray, numpy.ndarray]

classmethod concatenate(**sets*)

copy()

property data
The size of the voxels. When it is 0D or 1D it counts as the size for all voxels, if it is 2D it is 1 an individual size per voxel.

Return type Union[numpy.ndarray, None]

classmethod empty(*size=None*)

classmethod from_morphology(*morphology*, *estimate_n*, *with_data=True*)

get_data(*index=None*, */*, *copy=True*)

get_raw(*copy=True*)

get_size(*copy=True*)

get_size_matrix(*copy=True*)

property has_data
Whether the set has any data associated to the voxels

Return type bool

property is_empty
Whether the set contain any voxels

Return type bool

property of_equal_size

classmethod one(*ldc*, *mdc*, *data=None*)

property raw

property regular
Whether the voxels are placed on a regular grid.

resize(*size*)

select(*ldc*, *mdc*)

select_chunk(*chunk*)

property size
The size of the voxels. When it is 0D or 1D it counts as the size for all voxels, if it is 2D it is 1 an individual size per voxel.

Return type numpy.ndarray

snap_to_grid(*grid_size*, *unique=False*)

unique()

21.1.16 Module contents

CHAPTER

TWENTYTWO

INDEX

CHAPTER

TWENTYTHREE

MODULE INDEX

CHAPTER

TWENTYFOUR

GUIDES

24.1 Options

The BSB has several global options, which can be set through a 12-factor style cascade. The cascade goes as follows, in descending priority: script, CLI, project, env. The first to provide a value will be used. For example, if both a CLI and env value are provided, the CLI value will override the env value.

The script values can be set from the `bsb.options` module, CLI values can be passed to the command line, project settings can be stored in `pyproject.toml`, and env values can be set through use of environment variables.

24.1.1 Using script values

Read option values; if no script value is set, the other values are checked in cascade order:

```
import bsb.options

print(bsb.options.verbosity)
```

Set a script value; it has highest priority for the remainder of the Python process:

```
import bsb.options

bsb.options.verbosity = 4
```

Once the Python process ends, the values are lost. If you instead would like to set a script value but also keep it permanently as a project value, use *store*.

24.1.2 Using CLI values

The second priority are the values passed through the CLI, options may appear anywhere in the command.

Compile with verbosity 4 enabled:

```
bsb -v 4 compile
bsb compile -v 4
```

24.1.3 Using project values

Project values are stored in the Python project configuration file `pyproject.toml` in the `tools.bsb` section. You can modify the TOML content in the file, or use *`options.store()`*:

```
import bsb.options

bsb.options.store("verbosity", 4)
```

The value will be written to `pyproject.toml` and saved permanently at project level. To read any `pyproject.toml` values you can use *`options.read()`*:

```
import bsb.options

link = bsb.options.read("networks.config_link")
```

24.1.4 Using env values

Environment variables are specified on the host machine, for Linux you can set one with the following command:

```
export BSB_VERBOSITY=4
```

This value will remain active until you close your shell session. To keep the value around you can store it in a configuration file like `~/.bashrc` or `~/.profile`.

24.1.5 List of options

- `verbosity`: Determines how much output is produced when running the BSB.
  - *script*: `verbosity`
  - *cli*: `v`, `verbosity`
  - *project*: `verbosity`
  - *env*: `BSB_VERBOSITY`
- `force`: Enables sudo mode. Will execute destructive actions without confirmation, error or user interaction. Use with caution.
  - *script*: `sudo`
  - *cli*: `f`, `force`
  - *project*: None.
  - *env*: `BSB_FOOTGUN_MODE`
- `version`: Tells you the BSB version. **readonly**
  - *script*: `version`
  - *cli*: `version`
  - *project*: None.
  - *env*: None.
- `config`: The default config file to use, if omitted in commands.

- *script*: None (when scripting, you should create a *Configuration*) object.
- *cli*: `config`, usually positional. e.g. `bsb compile conf.json`
- *project*: `config`
- *env*: `BSB_CONFIG_FILE`

24.1.6 `pyproject.toml` structure

The BSB's project-wide settings are all stored in `pyproject.toml` under `tools.bsb`:

```
[tools.bsb]
config = "network_configuration.json"

[tools.bsb.networks]
config_link = ["sys", "network_configuration.json", "always"]
morpho_link = ["sys", "morphologies.h5", "changes"]
```

Writing your own options

You can create your own options as a *plugin* by defining a class that inherits from *BsbOption*:

```
from bsb.options import BsbOption
from bsb.reporting import report

class GreetingsOption(
  BsbOption,
  name="greeting",
  script=("greeting",),
  env=("BSB_GREETING",),
  cli=("g", "greet"),
  action=True,
):
  def get_default(self):
    return "Hello World! The weather today is: optimal modelling conditions."

  def action(self, namespace):
    # Actions are run before the CLI options such as verbosity take global effect.
    # Instead we can read or write the command namespace and act accordingly.
    if namespace.verbosity >= 2:
      report(self.get(), level=1)

# Make `GreetingsOption` available as the default plugin object of this module.
__plugin__ = GreetingsOption
```

Plugins are installed by `pip` which takes its information from `setup.py`/`setup.cfg`, where you can specify an entry point:

```
"entry_points": {
  "bsb.options" = ["greeting = my_pkg.greetings"]
}
```

After installing the setup with `pip` your option will be available:

```
$> pip install -e .
$> bsb
$> bsb --greet
$> bsb -v 2 --greet
Hello World! The weather today is: optimal modelling conditions.
$> export BSB_GREETING="2 PIs walk into a conference..."
$> bsb -v 2 --greet
2 PIs walk into a conference...
```

For more information on setting up plugins (even just locally) see *Plugins*.

24.2 Cell types

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

24.2.1 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 *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.

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
  },
  "partitions": {
    "granular_layer": {
      "origin": [0.0, 0.0, 0.0],
      "thickness": 150
    }
  },
  "cell_types": {
    "granule_cell": {
      "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"
      }
    }
  },
  "connectivity": {},
  "simulations": {}
}
```

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

24.3 Writing components

The architecture of the framework organizes your model into reusable components. It offers out of the box components for basic operations, but often you'll need to write your own.

Importing

To use –> needs to be importable –> local code, package or plugin

Structure

- Decorate with `@config.node`
- Inherit from interface
- Parametrize with config attributes
- Implement interface functions

Parametrization

Parameters defined as class attributes –> can be specified in config/init. Make things explicitly visible and settable.

Type handling, validation, requirements

Interface & implementation

Interface gives you a set of functions you must implement. If these functions are present, framework knows how to use your class.

The framework allows you to plug in user code pretty much anywhere. Neat.

Here's how you do it (theoretically):

1. Identify which **interface** you need to extend. An interface is a programming concept that lets you take one of the objects of the framework and define some functions on it. The framework has predefined this set of functions and expects you to provide them. Interfaces in the framework are always classes.
2. Create a class that inherits from that interface and implement the required and/or interesting looking functions of its public API (which will be specified).
3. Refer to the class from the configuration by its importable module name, or use a *Class maps*.

With a quick example, there's the `MorphologySelector` interface, which lets you specify how a subset of the available morphologies should be selected for a certain group of cells:

1. The interface is `bsb.morphologies.MorphologySelector` and the docs specify it has a `validate(self, morphos)` and `pick(self, morpho)` function.
2. Instant-Python ™, just add water:

```
from bsb.objects.cell_type import MorphologySelector
from bsb import config

@config.node
class MySizeSelector(MorphologySelector):
```

(continues on next page)

(continued from previous page)

```
  min_size = config.attr(type=float, default=20)
  max_size = config.attr(type=float, default=50)

  def validate(self, morphos):
    if not all("size" in m.get_meta() for m in morphos):
      raise Exception("Missing size metadata for the size selector")

  def pick(self, morpho):
    meta = morpho.get_meta()
    return meta["size"] > self.min_size and meta["size"] < self.max_size
```

3. Assuming that that code is in a `select.py` file relative to the working directory you can now access:

```
{
  "selector": "select.MySizeSelector",
  "min_size": 30,
  "max_size": 50
}
```

Share your code with the whole world and become an author of a *Plugins*!

24.3.1 Main components

Region

Partition

PlacementStrategy

ConnectivityStrategy

24.3.2 Placement components

MorphologySelector

MorphologyDistributor

RotationDistributor

Distributor

Indicator

24.4 Connectivity

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

```
{
  "connectivity": {
    "cell_A_to_cell_B": {
      "cls": "bsb.connectivity.VoxelIntersection",
      "pre": {
        "cell_types": ["cell_A"]
      },
      "post": {
          "cell_types": ["cell_B"]
      }
    }
  }
}
```

The *cls* specifies which *ConnectionStrategy* to load. The *pre* and *post* specify the two *hemitypes*.

24.4.1 Creating your own

You can create custom connectivity patterns by creating an importable module (refer to the Python documentation) with inside a class inheriting from *ConnectionStrategy*.

What follows is an example implementation, that we'll deconstruct, step by step. The example connects cells that are near each other between a `min` and `max` distance:

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

@config.node
class ConnectBetween(ConnectionStrategy):
  # Define the class' configuration attributes
  min = config.attr(type=float, default=0)
  max = config.attr(type=float, required=True)

  def __init__(self, **kwargs):
    # Here you can check if the object was properly configured
    if self.max < self.min:
      raise ConfigurationError("Max distance should be larger than min distance.")

  def connect(self, pre, post):
    # The `connect` function is responsible for deciding which cells get connected.
    # Use the `.placement` to get a dictionary of `PlacementSet`s to connect.
    for from_type, from_set in pre.placement.items():
      from_pos = from_set.load_positions()
      for to_type, to_set in post.placement.items():
        to_pos = to_set.load_positions()
        pairw_dist = dist.cdist(from_pos, to_pos)
        matches = (pairw_dist <= max) & (pairw_dist >= min)
        # Some more numpy code to convert the distance matches to 2 location matrices
        # ...
```

(continues on next page)

(continued from previous page)

```
        pre_locs = ...
        post_locs = ...
        self.connect_cells(from_type, to_type, pre_locs, post_locs)
```

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

```
{
  "connectivity": {
    "cell_A_to_cell_B": {
      "class": "my_module.ConnectBetween",
      "min": 10,
      "max": 15.5,
      "pre": {
        "cell_types": ["cell_A"]
      },
      "post": {
        "cell_types": ["cell_B"]
      }
    }
  }
}
```

Then, when it is time, the framework will call the strategy's *connect()* method.

Accessing configuration values

In short, the objects that are decorated with `@config.node` will already be fully configured before `__init__` is called and all attributes available under `self` (e.g. `self.min` and `self.max`). For more explanation on the configuration system, see *Introduction*. For specifics on configuration nodes, see *Nodes*.

Accessing placement data

The `connect` function is handed the placement information as the `pre` and `post` parameters. The `.placement` attribute contains the placement data under consideration as *PlacementSets*.

Note: The `connect` function is called multiple times, usually once per postsynaptic "chunk" populated by the postsynaptic cell types. For each chunk, a region of interest is determined of chunks that could contain cells to be connected. This is transparent to you, as long as you use the `pre.placement` and `post.placement` given to you; they show you an encapsulated view of the placement data matching the current task. Note carefully that if you use the regular `get_placement_set` functions that they will not be encapsulated, and duplicate data processing might occur.

Creating connections

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.

24.4.2 Connection types and labels

Warning: The following documentation has not been updated to v4 yet, please bother a dev to do so .

When defining a connection type under `connectivity` 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 `connectivity` configuration, you can define the subpopulation label:

```
{
  "connectivity": {
    "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 `connectivity` 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:

```
{
  "connectivity": {
    "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"]
        }
```

(continues on next page)

(continued from previous page)

```
      ],
      "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:

```
{
  "connectivity": {
    "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`

24.5 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": {

    }
  }
}
```

24.5.1 NEST

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

24.5.2 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": {

```

(continues on next page)

(continued from previous page)

```
      },
      "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,
```

(continues on next page)

(continued from previous page)

```
        "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": {
```

(continues on next page)

(continued from previous page)

```
        "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
        }
      }
    },

    "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

24.6 List of placement strategies

24.6.1 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.

24.6.2 ParallelArrayPlacement

Class: *bsb.placement.ParallelArrayPlacement*

24.6.3 FixedPositions

Class: *bsb.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]]
        }
    },
  }
}
```

24.7 List of connection strategies

24.7.1 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.

24.7.2 `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

24.8 Placement sets

`PlacementSets` are constructed from the *`Storage`* and can be used to retrieve lists of identifiers, positions, morphologies, rotations and additional datasets.

Warning: Loading these datasets from storage is an expensive operation. Store a local reference to the data you retrieve, don't make multiple calls.

24.8.1 Retrieving a PlacementSet

Multiple `get_placement_set` methods exist in several places as shortcuts to create the same *PlacementSet*. If the placement set does not exist, a `DatesetNotFoundError` is thrown.

```
from bsb.core import from_hdf5

network = from_hdf5("my_network.hdf5")
ps = network.get_placement_set("my_cell")
ps = network.get_placement_set(network.cell_types.my_cell)
ps = network.cell_types.my_cell.get_placement_set()
# Usually not the right choice:
ps = network.storage.get_placement_set(network.cell_types.my_cell)
```

24.8.2 Identifiers

Cells have no global identifiers, instead you use the indices of their data, i.e. the n-th position belongs to cell n, and the n-th rotation will therefor also belong to it.

24.8.3 Positions

The positions of the cells can be retrieved using the *load_positions()* method.

```
for n, position in enumerate(ps.positions):
  print("I am", ps.tag, "number", n)
  print("My position is", position)
```

24.8.4 Morphologies

The positions of the cells can be retrieved using the *load_morphologies()* method.

```
for n, (pos, morpho) in enumerate(zip(ps.load_positions(), ps.load_morphologies())):
  print("I am", ps.tag, "number", n)
  print("My position is", position)
```

Warning:

Loading morphologies is especially expensive.

load_morphologies() returns a *MorphologySet*. There are better ways to iterate over it using either **soft caching** or **hard caching**.

24.8.5 Rotations

The positions of the cells can be retrieved using the *load_rotations()* method.

24.8.6 Additional datasets

Not implemented yet.

24.9 BSB Packaging Guide

TODO

CHAPTER

TWENTYFIVE

EXAMPLES

25.1 Creating networks

25.1.1 Default network

The default configuration contains a skeleton configuration, without any components in it. When you instantiate a `Scaffold` without any parameters, this configuration is used:

```
from bsb.core import Scaffold

# Create a network with the default configuration.
network = Scaffold()
network.compile()
```

CHAPTER

TWENTYSIX

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
```

CHAPTER

TWENTYSEVEN

DOCUMENTATION

To build the documentation run:

```
cd docs
make html
```

27.1 Conventions

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

CHAPTER

TWENTYEIGHT

PLUGINS

The BSB is extensively extendible. While most smaller things such as a new placement or connectivity strategy can be used simply by importing or dynamic configuration, larger components such as new storage engines, configuration parsers or simulation backends are added into the BSB through its plugin system.

28.1 Creating a plugin

The plugin system detects pip packages that define `entry_points` of the plugin category. Entry points can be specified in your package's `setup` using the `entry_point` argument. See the setuptools documentation for a full explanation. Here are some plugins the BSB itself registers:

```
entry_points={
    "bsb.adapters": [
        "nest = bsb.simulators.nest",
        "neuron = bsb.simulators.neuron",
    ],
    "bsb.engines": ["hdf5 = bsb.storage.engines.hdf5"],
    "bsb.config.parsers": ["json = bsb.config.parsers.json"],
}
```

The keys of this dictionary are the plugin category that determine where the plugin will be used while the strings that it lists follow the `entry_point` syntax:

- The string before the = will be used as the plugin name.
- Dotted strings indicate the module path.
- An optional `:` followed by a function name can be used to specify a function in the module.

What exactly should be returned from each `entry_point` depends highly on the plugin category but there are some general rules that will be applied to the advertised object:

- The object will be checked for a `__plugin__` attribute, if present it will be used instead.
- If the object is a function (strictly a function, other callables are ignored), it will be called and the return value will be used instead.

This means that you can specify just the module of the plugin and inside the module set the plugin object with `__plugin__` or define a function `__plugin__` that returns it. Or if you'd like to register multiple plugins in the same module you can explicitly specify different functions in the different entry points.

28.1.1 Examples

In Python:

```
# my_pkg.plugin1 module
__plugin__ = my_plugin
```

```
# my_pkg.plugin2 module
def __plugin__():
    return my_awesome_adapter
```

```
# my_pkg.plugins
def parser_plugin():
    return my_parser

def storage_plugin():
    return my_storage
```

In `setup`:

```
{
    "bsb.adapters": ["awesome_sim = my_pkg.plugin2"],
    "bsb.config.parsers": [
        "plugin1 = my_pkg.plugin1",
        "parser = my_pkg.plugins:parser_plugin"
    ],
    "bsb.engines": ["my_pkg.plugins:storage_plugin"]
}
```

28.2 Categories

28.2.1 Configuration parsers

Category: `bsb.config.parsers`

Inherit from *config.parsers.Parser*. When installed a `from_<plugin-name>` parser function is added to the *bsb.config* module. You can set the class variable `data_description` to describe what kind of data this parser parses to users. You can also set `data_extensions` to a sequence of extensions that this parser will be considered first for when parsing files of unknown content.

28.2.2 Storage engines

Category: `bsb.engines`

28.2.3 Simulator adapters

Category: `bsb.adapters`

PYTHON MODULE INDEX

b

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z