Brain Scaffold Builder

Release 4.0.1

Robin De Schepper

Mar 29, 2024

GETTING STARTED

The BSB is a **black box component framework** for multiparadigm neural modelling: we provide structure, architecture and organization, and you provide the use-case specific parts of your model. In our framework, your model is described in a code-free configuration of **components** with parameters.

For the framework to reliably use components, and make them work together in a complex workflow, it asks a fixed set of questions per component type: e.g. a connection component will be asked how to connect cells. These contracts of cooperation between you and the framework are called **interfaces**. The framework executes a transparently parallelized workflow, and calls your components to fulfill their role.

This way, by *implementing our component interfaces* and declaring them in a configuration file, most models end up being code-free, well-parametrized, self-contained, human-readable, multi-scale models!

(PS: If we missed any hyped-up hyphenated adjectives, let us know!)

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

Tip: Use virtual environments!

The BSB framework can be installed using `pip`:

```
pip install "bsb~=4.0"
```

You can verify that the installation works with:

```
from bsb import Scaffold

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

You can now head over to the *get started*.

1.1 Parallel support

The BSB parallelizes the network reconstruction using MPI, and translates simulator instructions to the simulator backends with it as well, for effortless parallel simulation. To use MPI from Python the mpi4py package is required, which in turn needs a working MPI implementation installed in your environment.

On your local machine you can install OpenMPI:

```
sudo apt-get update && sudo apt-get install -y libopenmpi-dev openmpi-bin
```

On Windows, install Microsoft MPI. On supercomputers it is usually installed already, otherwise contact your administrator.

To then install the BSB with parallel MPI support:

```
pip install "bsb[parallel]~=4.0"
```

1.2 Simulator backends

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

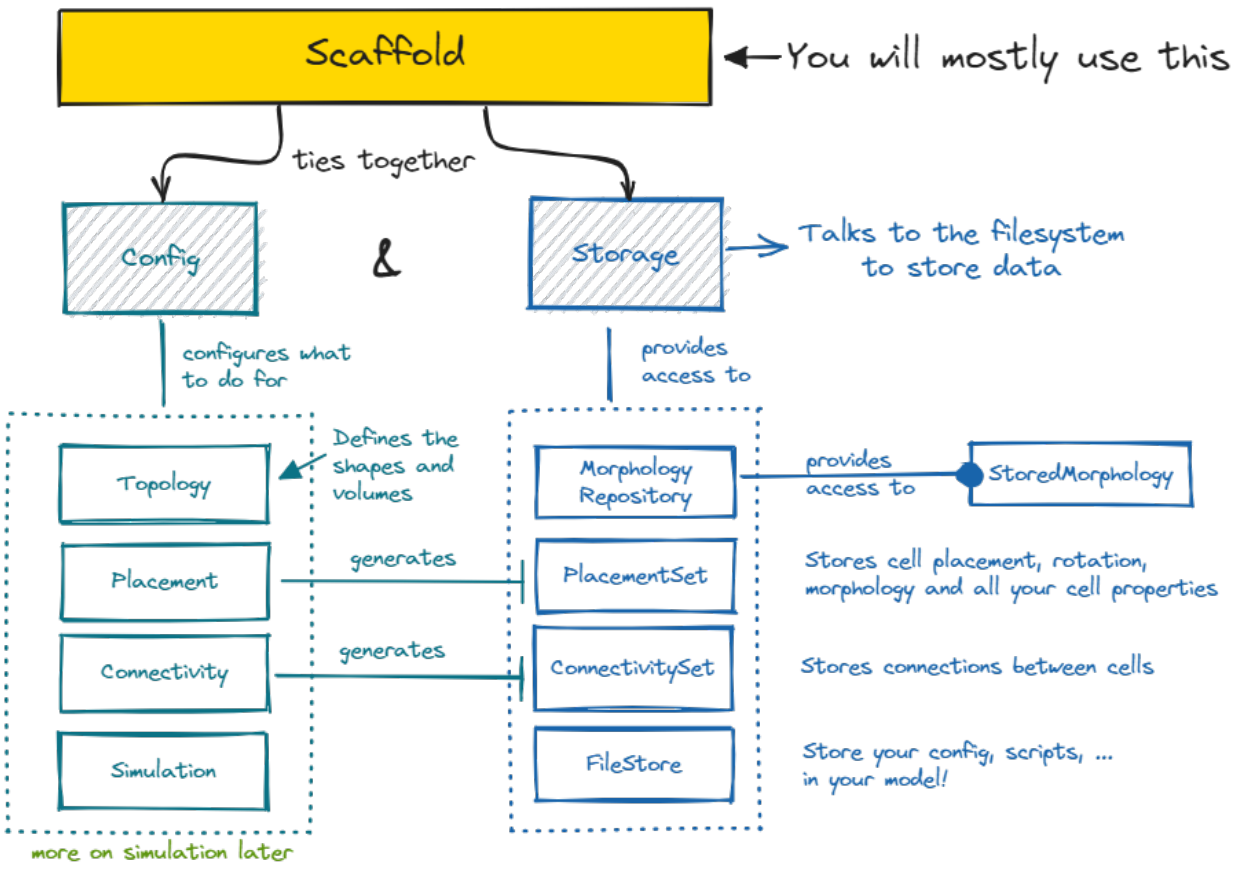
```
pip install bsb[nest]
# or
pip install bsb[arbor]
# or
pip install bsb[neuron]
# or any combination
pip install bsb[arbor,nest,neuron]
```

Note: This does not install the simulators themselves. It installs the Python tools that the BSB needs to support them. Install the simulators separately according to their respective installation instructions.

CHAPTER

TWO

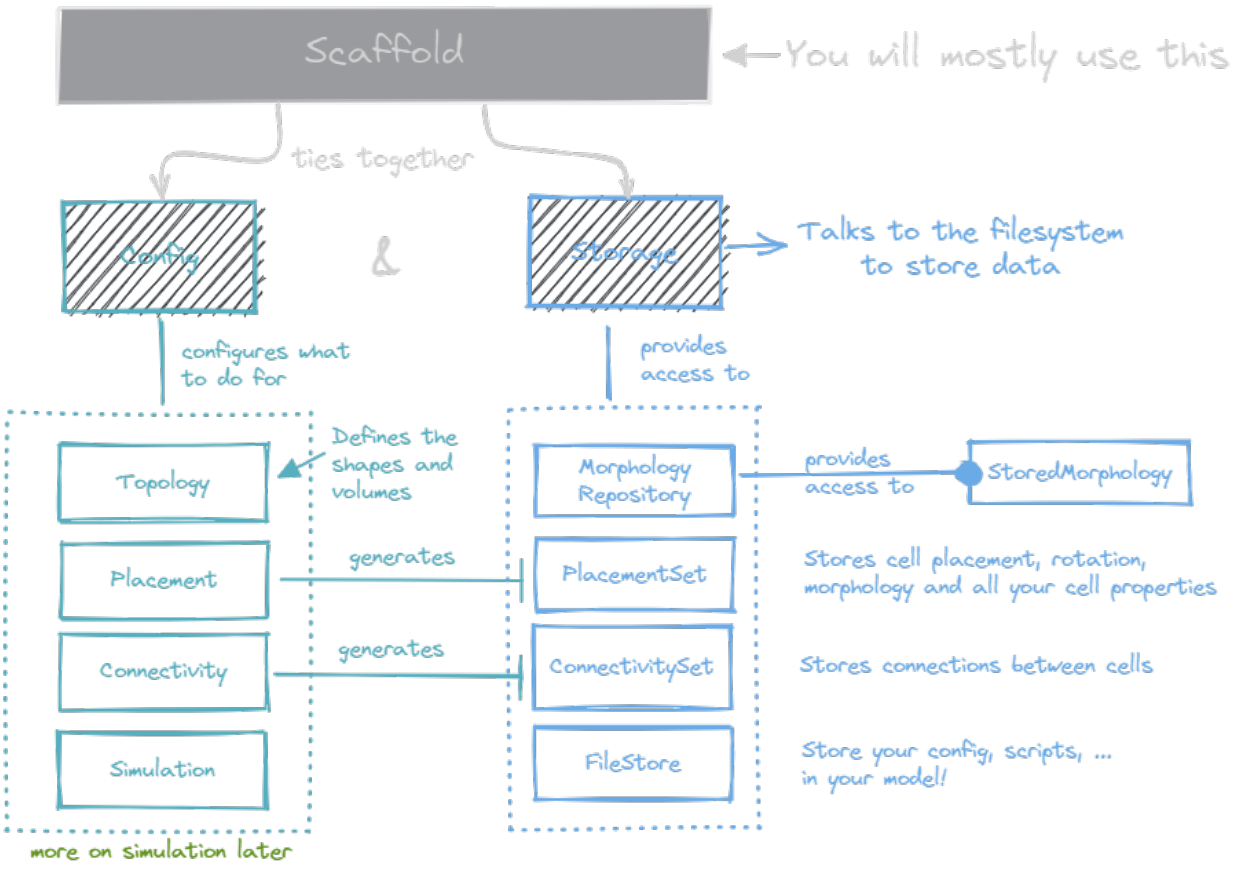
TOP LEVEL GUIDE



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 model description, while the storage contains your model data, like concrete cell positions or connections.

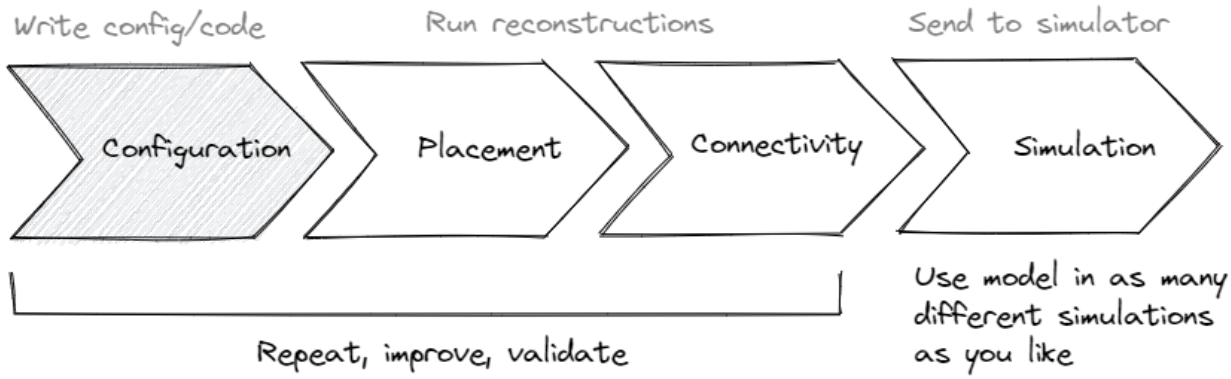
Using the scaffold object one can turn the abstract model configuration into a concrete storage object full of neuroscience. For it to do so, the configuration needs to describe which steps to take to place cells, called `Placement`, which steps to take to connect cells, called `Connectivity`, and what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

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*, and ConnectivitySets with *scaffold.get_connectivity_set*.

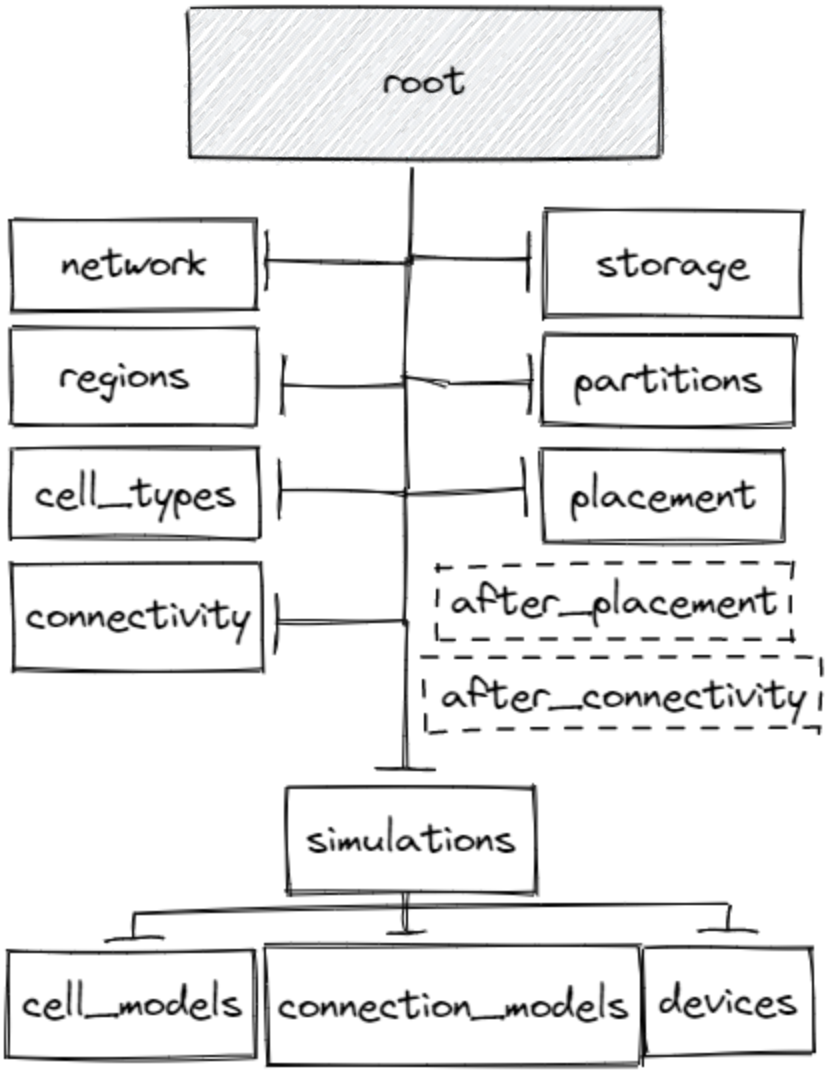

Scaffold
You will mostly use this
ties together
Config
&
Storage
Talks to the filesystem
to store data
configures what
to do for
provides
access to
Defines the
shapes and
volumes
Topology
Placement
Connectivity
Simulation
generates
generates
Morphology
Repository
provides
access to
StoredMorphology
PlacementSet
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

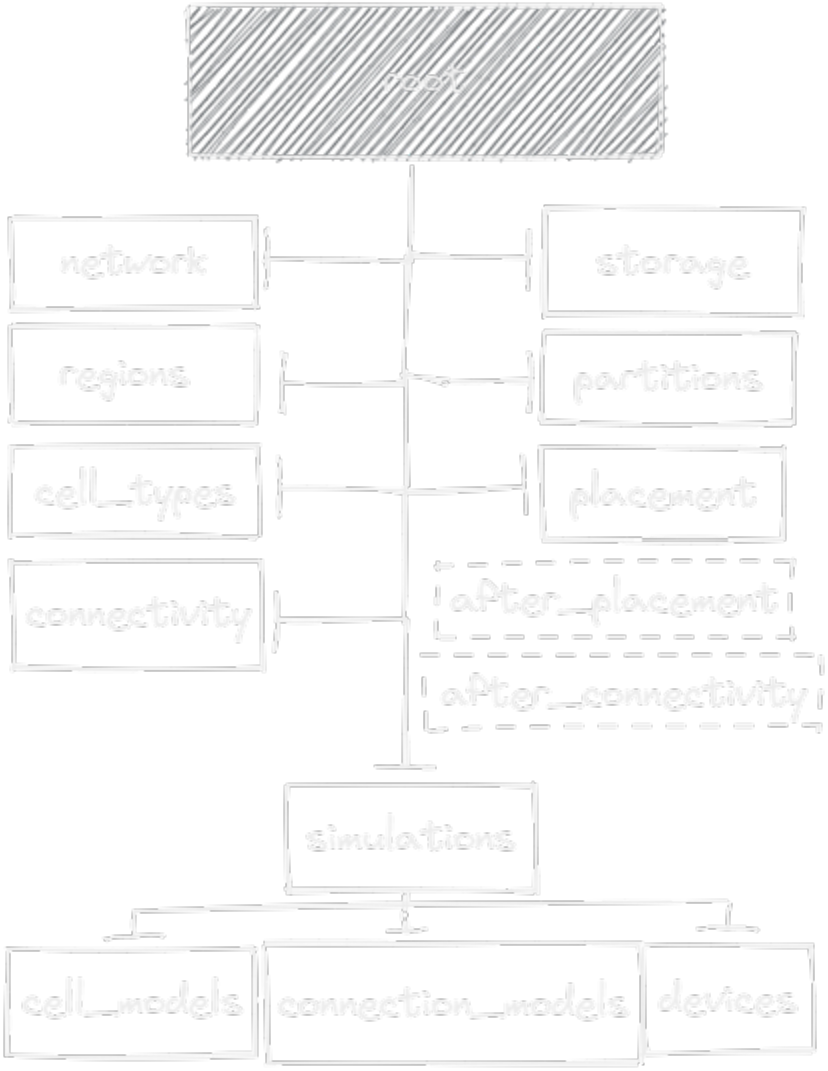
Ultimately this is the goal of the entire framework: To let you explicitly define every component and parameter that is a part of your model, and all its parameters, in such a way that a single CLI command, `bsb compile`, can turn your configuration into a reconstructed biophysically detailed large scale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

2.1 Workflow



2.2 Configuration


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


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

CHAPTER

THREE

GETTING STARTED

3.1 Your first network

Follow the *Installation*:

- 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.

The framework supports both declarative statements in configuration formats, or Python code. Be sure to take a quick look at each code tab to get a feel for the equivalent forms of configuration coding!

3.1.1 Create a project

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

```
bsb new my_first_model --quickstart --json
cd my_first_model
```

The project now contains a couple of important files:

- `network_configuration.json`: 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.

The configuration contains 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

```
pip install bsb-plotting
bsb compile --verbosity 3 --plot
```

PYTHON

```
from bsb_plot import plot_network

import bsb.options
from bsb import Scaffold, from_json

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

The `verbosity` flag increases the amount of output that is generated, to follow along or troubleshoot. The `plot` flags opens a plot .

3.1.2 Define starter components

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 second layer `top_layer`, and a region `brain_region` which will stack our layers on top of each other:

JSON

```
  "regions": {
    "brain_region": {
      "type": "stack",
      "children": ["base_layer", "top_layer"]
    }
  },
  "partitions": {
    "base_layer": {
      "type": "layer",
      "thickness": 100,
      "stack_index": 0
    },
    "top_layer": {
      "type": "layer",
      "thickness": 100,
```

(continues on next page)

(continued from previous page)

```
      "stack_index": 1
    }
  },
```

PYTHON

```
config.partitions.add("top_layer", thickness=100, stack_index=1)
config.regions.add(
    "brain_region",
    type="stack",
    children=[
        "base_layer",
        "top_layer",
    ],
)
```

The *type* of the `brain_region` is `stack`. This means it will place its children stacked on top of each other. The *type* of `base_layer` is `layer`. Layers specify their size in 1 dimension, and fill up the space in the other dimensions. See *Introduction* for more explanation on topology components.

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*:

JSON

```
  "cell_types": {
    "base_type": {
      "spatial": {
        "radius": 2,
        "density": 1e-3
      }
    },
    "top_type": {
      "spatial": {
        "radius": 7,
        "count": 10
      }
    }
  },
```

PYTHON

```
config.cell_types.add("top_type", spatial=dict(radius=7, count=10))
```

Placement

JSON

```
  "placement": {
    "base_placement": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["base_type"],
      "partitions": ["base_layer"]
    },
    "top_placement": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["top_type"],
      "partitions": ["top_layer"]
    }
  },
```

PYTHON

```
config.placement.add(
    "all_placement",
    strategy="bsb.placement.RandomPlacement",
    cell_types=["base_type", "top_type"],
    partitions=["base_layer"],
)
```

The `placement` blocks use the cell type indications to place cell types into partitions. You can use other *PlacementStrategies* by setting the *strategy* attribute. The BSB offers some strategies out of the box, or you can implement your own. The *RandomPlacement* places cells randomly in the assigned volume.

Take another look at your network:

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

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.

Warning: We pass the `--clear` flag to indicate that existing data may be overwritten. See *Storage flags* for more flags to deal with existing data.

Connectivity

JSON

```
  "connectivity": {
    "A_to_B": {
      "strategy": "bsb.connectivity.AllToAll",
      "presynaptic": {
        "cell_types": ["base_type"]
      },
      "postsynaptic": {
          "cell_types": ["top_type"]
      }
    }
  }
```

PYTHON

```
config.connectivity.add(
    "A_to_B",
    strategy="bsb.connectivity.AllToAll",
    presynaptic=dict(cell_types=["base_type"]),
    postsynaptic=dict(cell_types=["top_type"]),
)
```

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 *Simulating networks* stage.

What next?

Continue getting started Follow the next chapter and learn how to include morphologies.

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.

Recap

JSON

```
{
  "name": "Starting example",
  "storage": {
    "engine": "hdf5",
    "root": "network.hdf5"
  },
  "network": {
    "x": 400.0,
    "y": 600.0,
    "z": 400.0
  },
  "regions": {
    "brain_region": {
      "type": "stack",
      "children": ["base_layer", "top_layer"]
    }
  },
  "partitions": {
    "base_layer": {
      "type": "layer",
      "thickness": 100,
      "stack_index": 0
    },
    "top_layer": {
      "type": "layer",
      "thickness": 100,
      "stack_index": 1
    }
  },
  "cell_types": {
    "base_type": {
      "spatial": {
        "radius": 2,
        "density": 1e-3
      }
    },
    "top_type": {
      "spatial": {
        "radius": 7,
        "count": 10
      }
    }
  },
  "placement": {
    "base_placement": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["base_type"],
      "partitions": ["base_layer"]
    },
```

(continues on next page)

(continued from previous page)

```
    "top_placement": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["top_type"],
      "partitions": ["top_layer"]
    }
  },
  "connectivity": {
    "A_to_B": {
      "strategy": "bsb.connectivity.AllToAll",
      "presynaptic": {
        "cell_types": ["base_type"]
      },
      "postsynaptic": {
          "cell_types": ["top_type"]
      }
    }
  }
}
```

PYTHON

```
from bsb_plot import plot_network

import bsb.options
from bsb import Scaffold, from_json

bsb.options.verbosity = 3
config = from_json("network_configuration.json")

config.partitions.add("top_layer", thickness=100, stack_index=1)
config.regions.add(
    "brain_region",
    type="stack",
    children=[
        "base_layer",
        "top_layer",
    ],
)
config.cell_types.add("top_type", spatial=dict(radius=7, count=10))
config.placement.add(
    "all_placement",
    strategy="bsb.placement.RandomPlacement",
    cell_types=["base_type", "top_type"],
    partitions=["base_layer"],
)
config.connectivity.add(
    "A_to_B",
    strategy="bsb.connectivity.AllToAll",
    presynaptic=dict(cell_types=["base_type"]),
    postsynaptic=dict(cell_types=["top_type"]),
```

(continues on next page)

(continued from previous page)

```
)

network = Scaffold(config)
network.compile()
plot_network(network)
```

3.2 Adding morphologies

Note: This guide is a continuation of the *Getting Started guide*.

Hint: To follow along, download 2 morphologies from NeuroMorpho and save them as `neuron_A.swc` and `neuron2.swc` locally.

Previously we constructed a stacked double layer topology, with 2 cell types. We then connected them in an all-to-all fashion. The next step assigns *morphologies* to our cells, and connects the cells based on the intersection of their morphologies!

Morphologies can be loaded from local files or to fetch from remote sources, like NeuroMorpho.

3.2.1 Using local files

You can declare source morphologies in the root *morphologies* list:

YAML

```yaml
morphologies:
- neuron_A.swc
```

JSON

```
"morphologies": [
  "neuron_A"
],
```

PYTHON

```
    {"name": "neuron_B", "file": "neuron2.swc"},
]
config.cell_types.base_type.spatial.morphologies = ["neuron_A"]
```

In this case a morphology is created from `neuron_A.swc` and given the name `"neuron_A"`. As a second step, we associate this morphology to the `base_type` by referencing it by name in *cell_types.base_type.spatial.morphologies*:

YAML

```yaml
cell_types:
  base_type:
    spatial:
      radius: 2
      density: 0.001
      morphologies:
      - neuron_A
```

JSON

```json
  "cell_types": {
    "base_type": {
      "spatial": {
        "radius": 2,
        "density": 1e-3,
        "morphologies": [
          "neuron_A"
        ]
      }
    },
```

PYTHON

```
config.morphologies.append(
```

By default the name assigned to the morphology is the file name without its extension (here `.swc`). To change the name we can use a node with a *name* and *file*:

YAML

```yaml
morphologies:
- neuron_A.swc
- name: neuron_B
  file: neuron2.swc
```

JSON

```json
  "morphologies": [
    "neuron_A.swc",
    {
      "name": "neuron_B",
      "file": "neuron2.swc"
    },
```

PYTHON

```
    {"name": "neuron_B", "file": "neuron2.swc"},
]

config.cell_types.base_type.spatial.morphologies = ["neuron_A"]
```

It is also possible to add a pipeline to perform transformations on the loaded morphology. Pipelines can be added by adding a :guilabel`pipeline` list to the morphology node. Each item in the list may either be a string reference to an importable function or a method of the *Morphology* class. To pass parameters, use a node with the function reference placed in the guilabel:*func* attribute, and a *parameters* list. Here is an example what that would look like:

YAML

```
morphologies:
  - file: my_neuron.swc
    pipeline:
      - center
      - my_module.add_axon
      - func: rotate
        rotation: [20, 0, 20]
```

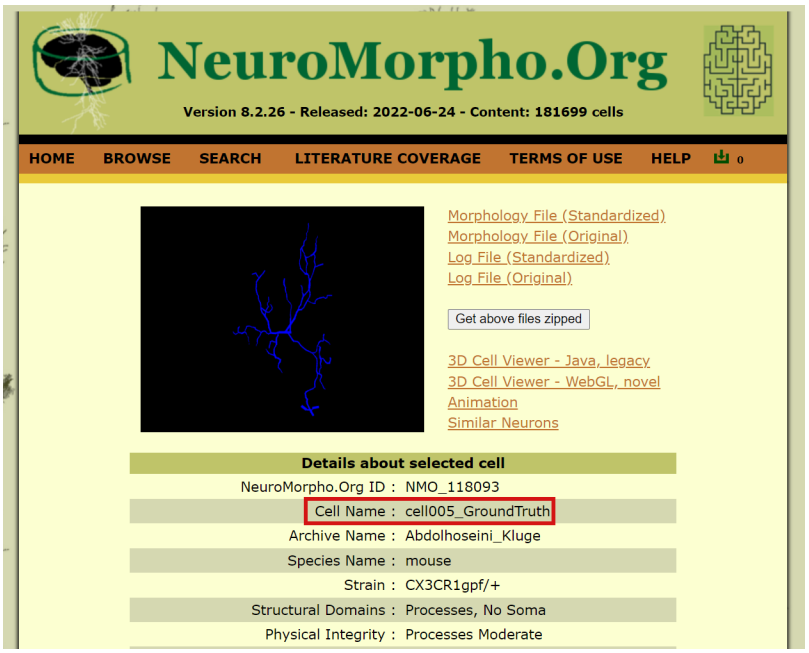
JSON

```
"morphologies": [
  {
    "file": "my_neuron.swc",
    "pipeline": [
      "center",
      "my_module.add_axon",
      {
        "func": "rotate",
        "rotation": [20, 0, 20]
      },
    ],
  }
]
```

Note: Any additional keys given in a pipeline step, such as *rotation* in the example, are passed to the function as keyword arguments.

3.2.2 Fetching with alternative URI schemes

The framework uses URI schemes to define the path of the sources that are loaded. By default it tries to load from the project local folder, using the``file`` URI scheme (`"file://"`). It is possible to fetch morphologies directly from neuromorpho.org using the NeuroMorpho scheme (`"nm://"`). You can refer to NeuroMorpho morphologies by their morphology name:



YAML

```yaml
morphologies:
- neuron_A.swc
- name: neuron_B
  file: neuron2.swc
```

JSON

```
  "morphologies": [
    "neuron_A.swc",
    {
      "name": "neuron_B",
      "file": "neuron2.swc"
    },
    {
       "name": "neuron_NM",
       "file": "nm://cell005_GroundTruth"
    }
  ],
  "cell_types": {
    "base_type": {
```

(continues on next page)

(continued from previous page)

```
      "spatial": {
        "radius": 2,
        "density": 1e-3,
        "morphologies": [
          "neuron_A"
        ]
      }
    },
    "top_type": {
      "spatial": {
        "radius": 7,
        "count": 10,
        "morphologies": [
          "neuron_B",
          "neuron_NM"
        ]
      }
    }
  },
```

PYTHON

```
)

config.cell_types.add(
    "top_type",
    spatial=dict(
        radius=7,
        count=10,
        morphologies=["neuron_B", "neuron_NM"],
    ),
)
config.placement.add(
    "all_placement",
```

3.2.3 Morphology intersection

Now that we have assigned morphologies to our cell types, we can use morphology-based connection strategies such as *VoxelIntersection*:

YAML

```yaml
connectivity:
  A_to_B:
    strategy: bsb.connectivity.VoxelIntersection
    presynaptic:
      cell_types:
      - base_type
    postsynaptic:
      cell_types:
      - top_type
```

JSON

```json
  "connectivity": {
    "A_to_B": {
      "strategy": "bsb.connectivity.VoxelIntersection",
      "presynaptic": {
        "cell_types": ["base_type"]
      },
      "postsynaptic": {
          "cell_types": ["top_type"]
      }
    }
  }
```

PYTHON

```python
    strategy="bsb.connectivity.VoxelIntersection",
    presynaptic=dict(cell_types=["base_type"]),
    postsynaptic=dict(cell_types=["top_type"]),
)

network = Scaffold(config)
```

Note: If there's multiple morphologies per cell type, they'll be assigned randomly, unless you specify a *MorphologyDistributor*.

3.2.4 Recap

YAML

```yaml
name: Starting example
storage:
  engine: hdf5
  root: network.hdf5
```

(continues on next page)

(continued from previous page)

```
network:
  x: 400
  y: 600
  z: 400
morphologies:
- neuron_A.swc
- name: neuron_B
  file: neuron2.swc
- name: neron_NM
  file: nm://cell005_GroundTruth
regions:
  brain_region:
    type: stack
    children:
    - base_layer
    - top_layer
partitions:
  base_layer:
    type: layer
    thickness: 100
    stack_index: 0
  top_layer:
    type: layer
    thickness: 100
    stack_index: 1
cell_types:
  base_type:
    spatial:
      radius: 2
      density: 0.001
      morphologies:
      - neuron_A
  top_type:
    spatial:
      radius: 7
      count: 10
      morphologies:
      - neuron_NM
placement:
  base_placement:
    strategy: bsb.placement.RandomPlacement
    cell_types:
    - base_type
    partitions:
    - base_layer
  top_placement:
    strategy: bsb.placement.RandomPlacement
    cell_types:
    - top_type
    partitions:
    - top_layer
connectivity:
```

(continues on next page)

(continued from previous page)

```
  A_to_B:
    strategy: bsb.connectivity.VoxelIntersection
    presynaptic:
      cell_types:
      - base_type
    postsynaptic:
      cell_types:
      - top_type
```

JSON

```
{
  "name": "Starting example",
  "storage": {
    "engine": "hdf5",
    "root": "network.hdf5"
  },
  "network": {
    "x": 400.0,
    "y": 600.0,
    "z": 400.0
  },
  "morphologies": [
    "neuron_A.swc",
    {
      "name": "neuron_B",
      "file": "neuron2.swc"
    },
    {
       "name": "neuron_NM",
       "file": "nm://cell005_GroundTruth"
    }
  ],
  "regions": {
    "brain_region": {
      "type": "stack",
      "children": ["base_layer", "top_layer"]
    }
  },
  "partitions": {
    "base_layer": {
      "type": "layer",
      "thickness": 100,
      "stack_index": 0
    },
    "top_layer": {
      "type": "layer",
      "thickness": 100,
      "stack_index": 1
    }
```

(continues on next page)

(continued from previous page)

```
  },
  "cell_types": {
    "base_type": {
      "spatial": {
        "radius": 2,
        "density": 1e-3,
        "morphologies": [
          "neuron_A"
        ]
      }
    },
    "top_type": {
      "spatial": {
        "radius": 7,
        "count": 10,
        "morphologies": [
          "neuron_B",
          "neuron_NM"
        ]
      }
    }
  },
  "placement": {
    "base_placement": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["base_type"],
      "partitions": ["base_layer"]
    },
    "top_placement": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["top_type"],
      "partitions": ["top_layer"]
    }
  },
  "connectivity": {
    "A_to_B": {
      "strategy": "bsb.connectivity.VoxelIntersection",
      "presynaptic": {
        "cell_types": ["base_type"]
      },
      "postsynaptic": {
          "cell_types": ["top_type"]
      }
    }
  }
}
```

PYTHON

```python
from bsb_plot import plot_network

import bsb.options
from bsb import Scaffold, Stack, from_json

bsb.options.verbosity = 3
config = from_json("network_configuration.json")

config.partitions.add("top_layer", thickness=100, stack_index=1)
config.regions["brain_region"] = Stack(
    children=[
        "base_layer",
        "top_layer",
    ]
)
config.morphologies = [
    "neuron_A.swc",
    {"name": "neuron_B", "file": "neuron2.swc"},
]

config.cell_types.base_type.spatial.morphologies = ["neuron_A"]

config.morphologies.append(
    {"name": "neuron_NM", "file": "nm://cell005_GroundTruth"},
)

config.cell_types.add(
    "top_type",
    spatial=dict(
        radius=7,
        count=10,
        morphologies=["neuron_B", "neuron_NM"],
    ),
)
config.placement.add(
    "all_placement",
    strategy="bsb.placement.RandomPlacement",
    cell_types=["base_type", "top_type"],
    partitions=["base_layer"],
)
config.connectivity.add(
    "A_to_B",
    strategy="bsb.connectivity.VoxelIntersection",
    presynaptic=dict(cell_types=["base_type"]),
    postsynaptic=dict(cell_types=["top_type"]),
)

network = Scaffold(config)

network.compile()
plot_network(network)
```

3.3 Managing network files

3.3.1 Creating networks

Default network

The default configuration contains a skeleton configuration, for an HDF5 storage, without any components in it. The file will be called something like `scaffold_network_2022_06_29_10_10_10.hdf5`, and will be created once you construct the *Scaffold* object:

```
from bsb import Scaffold

network = Scaffold()
network.compile()
```

Network from config

You can also first load or create a *Configuration* object, and create a network from it, by passing it to the *Scaffold*:

```
from bsb import Configuration, Scaffold

cfg = Configuration()
# Let's set a file name for the network
cfg.storage.root = "my_network.hdf5"
# And add a cell type
cfg.cell_types.add(
    "hero_cells",
    spatial=dict(
        radius=2,
        density=1e-3,
    ),
)

# After customizing your configuration, create a network from it.
network = Scaffold(cfg)
network.compile()
```

3.3.2 Loading a network from file

You can load a stored network from file using *bsb.core.from_storage()*:

```
from bsb import from_storage

network = from_storage("my_network.hdf5")
```

3.3.3 Accessing network data

Configuration

The configuration of a network is available as `network.configuration`, the root nodes such as `cell_types`, `placement` and others are available on `network` as well.

```
from bsb import from_storage

network = from_storage("network.hdf5")
print("My network was configured with", network.configuration)
print("My network has", len(network.configuration.cell_types), "cell types")
(
    # But to avoid some needless typing and repetition,
    network.cell_types is network.configuration.cell_types
    and network.placement is network.configuration.placement
    and "so on"
)
```

Placement data

The placement data is available through the *storage.interfaces.PlacementSet* interface. This example shows how to access the cell positions of each population:

```
import numpy as np

from bsb import from_storage

network = from_storage("network.hdf5")
for cell_type in network.cell_types:
    ps = cell_type.get_placement_set()
    pos = ps.load_positions()
    print(len(pos), cell_type.name, "placed")
    # The positions are an (Nx3) numpy array
    print("The median cell is located at", np.median(pos, axis=0))
```

See also:

load_morphologies(), *load_rotations()*.

Todo: Document best practices for the morphology data

Todo: Document best practices for the connectivity data

3.4 Projects

Projects help you keep your models organized, safe, and neat! A project is a folder containing:

- The `pyproject.toml` Python project settings file: This file uses the TOML syntax to set configuration values for the BSB and any other python tools your project uses.
- One or more configuration files.
- One or more network files.
- Your component code.

You can create projects using the *bsb.new* command.

3.4.1 Settings

Project settings are contained in the `pyproject.toml` file.

- `[tools.bsb]`: The root configuration section: You can set the values of any *Options* here.
  - `[tools.bsb.links]`: Contains the *file link* definitions.
  - `[tools.bsb.links."my_network.hdf5"]`: Storage specific file links In this example for a storage object called "my_network.hdf5"

```
[tools.bsb]
verbosity = 3

[tools.bsb.links]
config = "auto"

[tools.bsb.links."thalamus.hdf5"]
config = [ "sys", "thalamus.json", "always",]
```

3.4.2 File links

Storage objects can keep copies of configuration and morphologies. These copies might become outdated during development. To automatically update it, you can specify file links.

It is recommended that you only specify links for models that you are actively developing, to avoid overwriting and losing any unique configs or morphologies of a model.

Config links

Configuration links (`config =`) can be either *fixed* or *automatic*. Fixed config links will always overwrite the configuration of the model with the contents of the file, if it exists. Automatic config links do the same, but keep track of the path of the last saved config file, and stay linked with that file.

Syntax

The first argument is the *provider* of the link: `sys` for the filesystem (your folder) `fs` for the file store of the storage engine (storage engines may have their own way of storing files). The second argument is the path to the file, and the third argument is when to update, but is unused! For automatic config links you can simply pass the `"auto"` string.

Note: Links in `tools.bsb.links` are active for all models in your project! It's better to specify them on a per model basis using the `tools.bsb.links."my_model_name.hdf5"` section.

3.4.3 Component code

It's best practice to keep all of your component code in a subfolder with the same name as your model. For example, if you're modelling the cerebellum, create a folder called `cerebellum`. Inside place an `__init__.py` file, so that Python can import code from it. Then you best subdivide your code based on component type, e.g. keep placement strategies in a file called `placement.py`. That way, your placement components are available in your model as `cerebellum.placement.MyComponent`. It will also make it easy to distribute your code as a package!

3.4.4 Version control

An often overlooked aspect is version control! Version control helps you track every change you make as a version of your code, backs up your code, and lets you switch between versions. The `git` protocol is currently the most popular version control, combined with providers like GitHub or GitLab.

```
- This was my previous version
+ This is my new version
This line was not affected
```

This example shows how version control can track every change you make, to undo work, to try experimental changes, or to work on multiple conflicting features. Every change can be stored as a version, and backed up in the cloud.

Projects come with a `.gitignore` file, where you can exclude files from being backed up. Cloud providers won't let neuroscientists upload 100GB network files

CHAPTER

FOUR

GUIDES

4.1 Writing components

Todo:

- Write this skeleton out to a full guide.
- Start this out in a Getting Started style, where a toy problem is tackled.
- Then, for each possible component type, write an example that covers the interface and common problems and important to know things.

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 import config, MorphologySelector

@config.node
class MySizeSelector(MorphologySelector):
  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:

```
{
  "select": "select.MySizeSelector",
  "min_size": 30,
  "max_size": 50
}
```

Share your code with the whole world and become an author of a *plugin*!

4.1.1 Main components

Region

Partition

PlacementStrategy

ConnectivityStrategy

4.1.2 Placement components

MorphologySelector

MorphologyDistributor

RotationDistributor

Distributor

Indicator

4.2 BSB Packaging Guide

Todo: Well, writing this guide

CHAPTER

FIVE

EXAMPLES

5.1 Creating networks

5.1.1 Default network

The default configuration contains a skeleton configuration, for an HDF5 storage, without any components in it. The file will be called something like `scaffold_network_2022_06_29_10_10_10.hdf5`, and will be created once you construct the *Scaffold* object:

```
from bsb import Scaffold

network = Scaffold()
network.compile()
```

5.1.2 Network from config

You can also first load or create a *Configuration* object, and create a network from it, by passing it to the *Scaffold*:

```
from bsb import Configuration, Scaffold

cfg = Configuration()
# Let's set a file name for the network
cfg.storage.root = "my_network.hdf5"
# And add a cell type
cfg.cell_types.add(
    "hero_cells",
    spatial=dict(
        radius=2,
        density=1e-3,
    ),
)

# After customizing your configuration, create a network from it.
network = Scaffold(cfg)
network.compile()
```

5.2 Loading a network from file

You can load a stored network from file using *bsb.core.from_storage()*:

```
from bsb import from_storage

network = from_storage("my_network.hdf5")
```

5.3 Accessing network data

5.3.1 Configuration

The configuration of a network is available as `network.configuration`, the root nodes such as `cell_types`, `placement` and others are available on `network` as well.

```
from bsb import from_storage

network = from_storage("network.hdf5")
print("My network was configured with", network.configuration)
print("My network has", len(network.configuration.cell_types), "cell types")
(
    # But to avoid some needless typing and repetition,
    network.cell_types is network.configuration.cell_types
    and network.placement is network.configuration.placement
    and "so on"
)
```

5.3.2 Placement data

The placement data is available through the *storage.interfaces.PlacementSet* interface. This example shows how to access the cell positions of each population:

```
import numpy as np

from bsb import from_storage

network = from_storage("network.hdf5")
for cell_type in network.cell_types:
    ps = cell_type.get_placement_set()
    pos = ps.load_positions()
    print(len(pos), cell_type.name, "placed")
    # The positions are an (Nx3) numpy array
    print("The median cell is located at", np.median(pos, axis=0))
```

See also:

load_morphologies(), *load_rotations()*.

Todo: Document best practices for the morphology data

Todo: Document best practices for the connectivity data

5.4 Mouse brain atlas based placement

The BSB supports integration with cell atlases. All that's required is to implement a *Voxels* partition so that the atlas data can be converted from the atlas raster format, into a framework object. The framework has *Allen Mouse Brain Atlas integration* out of the box, and this example will use the *AllenStructure*.

After loading shapes from the atlas, we will use a local data file to assign density values to each voxel, and place cells accordingly.

We start by defining the basics: a region, an `allen` partition and a cell type:

```
"regions": {
  "brain": {"children": ["declive"]}
},
"partitions": {
  "declive": {
    "type": "allen",
    "struct_name": "DEC"
  }
},
"cell_types": {
  "my_cell": {
    "spatial": {
      "radius": 2.5,
      "density": 0.003
    }
  }
},
```

Here, the *mask_source* is not set so BSB will automatically download the 2017 version of the CCFv3 mouse brain annotation atlas volume from the Allen Institute website. Use *mask_source* to provide your own nrrd annotation volume.

The *struct_name* refers to the Allen mouse brain region acronym or name. You can also replace that with *struct_id*, if you're using the numeric identifiers. You can find the ids, acronyms and names in the Allen Brain Atlas brain region hierarchy file.

If we now place our `my_cell` in the `declive`, it will be placed with a fixed density of `0.003/m^3`:

```
"placement": {
  "example_placement": {
    "cls": "bsb.placement.RandomPlacement",
    "cell_types": ["my_cell"],
    "partitions": ["declive"]
  }
},
```

If however, we have data of the cell densities available, we can link our `declive` partition to it, by loading it as a *source* file:

```
  "partitions": {
    "declive": {
      "type": "allen",
      "source": "my_cell_density.nrrd",
      "keys": ["my_cell_density"],
      "struct_name": "DEC"
    }
  },
```

The *source* file will be loaded, and the values at the coordinates of the voxels that make up our partition are associated as a column of data. We use the *data_keys* to specify a name for the data column, so that in other places we can refer to it by name.

We need to select which data column we want to use for the density of `my_cell`, since we might need to load multiple densities for multiple cell types, or orientations, or other data. We can do this by specifying a *density_key*:

```
  "cell_types": {
    "my_cell": {
      "spatial": {
        "radius": 2.5,
        "density_key": "my_cell_density",
      }
    }
  },
```

That's it! If we compile the network, `my_cell` will be placed into `declive` with different densities in each voxel, according to the values provided in `my_cell_density.nrrd`.

CHAPTER

SIX

CONFIGURATION FILES

A configuration file describes the components of 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 is structured 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*, *nodes*, *dictionaries*, *lists*, and *references*. These configuration units can be declared through the config file, or programatically added.

6.1 Code

Most of the framework components pass the data on to Python classes, that determine the underlying code strategy of the component. In order to link your Python classes to the configuration file they should be an importable module. Here's an example of how the `MySpecialConnection` class in the local Python file `connectome.py` would be available to the configuration:

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

The framework will try to pass the additional keys `value1` and `thingy2` to the class. The class should be decorated as a configuration node for it to correctly receive and handle the values:

```
from bsb import config, ConnectionStrategy

@config.node
class MySpecialConnection(ConnectionStrategy):
  value1 = config.attr(type=int)
  thingy2 = config.list(type=int, size=2, required=True)
```

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

6.2 JSON

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.

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

6.2.1 JSON parser

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

- JSON references
- JSON imports

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:

```
{
  "template": {
    "A": "value",
    "B": "value"
  },
  "copy": {
    "$ref": "#/template"
  }
}
```

Will be parsed into:

```
{
  "template": {
    "A": "value",
    "B": "value"
  },
  "copy": {
    "A": "value",
    "B": "value"
  }
}
```

Note: Imported keys will not override keys that are already present. This way you can specify local data to customize what you import. If both keys are dictionaries, they are merged; with again priority for the local data.

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 for same document references. 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": {}
      }
    }
  }
}
```

In this document the deepest JSON path is `/walk/down/the/path`.

Warning: Pay attention to the initial / of the reference clause! Without it, you're making a reference relative to the current position. With an initial / you make a reference absolute to the root of the document.

JSON Imports

Imports are the bigger cousin of the reference. They can import multiple dictionaries from a common parent at the same time as siblings:

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

Will be parsed into:

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

Note: If you don't specify any *values* all nodes will be imported.

Note: The same merging rules apply as to the reference.

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 *values* key which lists which keys to import from the reference target.

6.3 Default configuration

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

```
{
  "storage": {
    "engine": "hdf5"
  },
  "network": {
    "x": 200, "y": 200, "z": 200
  },
  "partitions": {

  },
  "cell_types": {

  },
  "placement": {

  },
  "connectivity": {

  }
}
```

CHAPTER

SEVEN

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 in 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"
  }
}
```

7.1 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 `cls` attribute.

The value that is given to this attribute will be used to load the class of the node:

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

And in the configuration:

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

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

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

7.1.1 Configuring the dynamic attribute

The same keyword arguments can be passed to the `dynamic` decorator as to regular attributes to specify the properties of the dynamic attribute; As an example we specify a new attribute name with `attr_name="example_type"`, allow the dynamic attribute to be omitted `required=False`, and specify a fallback class with `default="Example"`:

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

@config.node
class Explicit(Example):
  purpose = config.attr(required=True)
```

`Example` can then be defined as either:

```
{
  "example_type": "Explicit",
  "purpose": "show explicit dynamic node"
}
```

or, because of the `default` kwarg, `Example` can be implicitly used by omitting the dynamic attribute:

```
{
  "purpose": "show implicit fallback"
}
```

7.1.2 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 Root node

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

7.3 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.

7.4 Node inheritance

Classes decorated with node decorators have their class and metaclass machinery rewritten. Basic inheritance works like this:

```
@config.node
class NodeA:
    pass

@config.node
class NodeB(NodeA):
    pass
```

However, when inheriting from more than one node class you will run into a metaclass conflict. To solve it, use *config.compose_nodes()*:

```
from bsb import config, compose_nodes

@config.node
class NodeA:
    pass

@config.node
class NodeB:
    pass

@config.node
class NodeC(compose_nodes(NodeA, NodeB)):
    pass
```

7.5 Configuration attributes

An attribute can refer to a singular value of a certain type, a dict, list, reference, or to a deeper node. You can use the *config.attr* in node decorated classes to define your attribute:

```
from bsb import config

@config.node
```

(continues on next page)

(continued from previous page)

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

```
{
  "count": 12,
  "candy": {
    "name": "Hardcandy",
    "sweetness": 4.5
  }
}
```

7.6 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"`.

7.7 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
      }
    }
  ]
}
```

7.8 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.

7.8.1 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.

7.8.2 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 `[]`.

7.8.3 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
    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`

7.9 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.

CHAPTER

EIGHT

TYPE VALIDATION

Configuration types convert given configuration values. Values incompatible with the type are rejected and the user is warned. 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.

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. Make sure that you use `config.attr(type=types.any_())`, as opposed to `config.attr(type=types.any_)`.

8.1 Examples

```
from bsb import config, 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
  list_of_numbers = config.attr(
    type=types.list(type=float)
  )
```

(continues on next page)

(continued from previous page)

```
  # 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")
```

CHAPTER

NINE

CONFIGURATION REFERENCE

9.1 Root nodes

PYTHON

```python
from bsb.config import Configuration

Configuration(
  name='example',
  components=[],
  packages=[
    {
  ],
  morphologies=[
    {
  ],
  storage={},
  network={},
  regions={},
  partitions={},
  cell_types={},
  placement={},
  after_placement={},
  connectivity={},
  after_connectivity={},
  simulations={},
)
```

JSON

```json
{
    "name": "example",
    "components": [],
    "packages": [
        [
            "u",
            "u"
        ]
```

(continues on next page)

(continued from previous page)

```
    ],
    "morphologies": [
        [
            "i",
            "i",
            "a",
            "a"
        ]
    ],
    "storage": {},
    "network": {},
    "regions": {},
    "partitions": {},
    "cell_types": {},
    "placement": {},
    "after_placement": {},
    "connectivity": {},
    "after_connectivity": {},
    "simulations": {}
}
```

YAML

```
after_connectivity: {}
after_placement: {}
cell_types: {}
components: []
connectivity: {}
morphologies:
- - i
  - i
  - a
  - a
name: example
network: {}
packages:
- - u
  - u
partitions: {}
placement: {}
regions: {}
simulations: {}
storage: {}
```

9.1.1 Storage

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

PYTHON

```
from bsb.storage.interfaces import StorageNode

StorageNode(
  root='example',
  engine='example',
)
```

JSON

```
{
    "root": "example",
    "engine": "example"
}
```

YAML

```
engine: example
root: example
```

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

9.1.2 Network

PYTHON

```
NetworkNode(
  x=3.14,
  y=3.14,
  z=3.14,
  origin=[0, 0, 0],
  chunk_size=[100.0, 100.0, 100.0],
)
```

JSON

```
{
    "x": 3.14,
    "y": 3.14,
    "z": 3.14,
    "origin": [
        0,
        0,
        0
    ],
    "chunk_size": [
        100.0,
        100.0,
        100.0
    ]
}
```

YAML

```
chunk_size:
- 100.0
- 100.0
- 100.0
origin:
- 0
- 0
- 0
x: 3.14
y: 3.14
z: 3.14
```

- *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. Can be a list of 3 floats for a rhomboid or 1 float for cubes.

9.1.3 Components

PYTHON

```
CodeDependencyNode(
  file=True,
  module='example',
  attr='example',
)
```

JSON

```
{
    "file": true,
    "module": "example",
    "attr": "example"
}
```

YAML

```
attr: example
file: true
module: example
```

9.1.4 Morphologies

PYTHON

```
from bsb.morphologies.parsers.parser import MorphologyParser

MorphologyDependencyNode(
  file='example',
  pipeline=[
    {
      'func': None,
      'parameters': None,
    },
  ],
  name='example',
  parser=MorphologyParser(
    cls='builtins.str',
    branch_cls='builtins.str',
    parser='bsb',
  ),
)
```

JSON

```
{
    "file": "example",
    "pipeline": [
        {
            "func": null,
            "parameters": null
        }
    ],
    "name": "example",
    "parser": {
```

(continues on next page)

(continued from previous page)

```
        "cls": "builtins.str",
        "branch_cls": "builtins.str",
        "parser": "bsb"
    }
}
```

YAML

```
file: example
name: example
parser:
  branch_cls: builtins.str
  cls: builtins.str
  parser: bsb
pipeline:
- func: null
  parameters: null
```

9.1.5 Regions

Note: Region nodes are components and can contain additional component-specific attributes.

PYTHON

```
from bsb.topology.region import Region

Region(
  name='example',
  children=[],
  type='group',
)
```

JSON

```
{
    "name": "example",
    "children": [],
    "type": "group"
}
```

YAML

```yaml
children: []
name: example
type: group
```

- *type*: Type of the region, determines what kind of structure it imposes on its children.
- *offset*: Offset of this region to its parent in the topology.

9.1.6 Partitions

Note: Partition nodes are components and can contain additional component-specific attributes.

PYTHON

```python
from bsb.topology.partition import Partition

Partition(
  name='example',
  type='layer',
)
```

JSON

```json
{
    "name": "example",
    "type": "layer"
}
```

YAML

```yaml
name: example
type: layer
```

- *type*: Name of the partition component, or its class.
- *region*: By-name reference to a region.

9.1.7 Cell types

PYTHON

```python
from bsb.cell_types import CellType, Plotting
from bsb.placement.indicator import PlacementIndications
from bsb.morphologies.selector import MorphologySelector

CellType(
  name='example',
  spatial=PlacementIndications(
    radius=3.14,
    density=3.14,
    planar_density=3.14,
    count_ratio=3.14,
    density_ratio=3.14,
    relative_to=None,
    count=42,
    geometry=True,
    morphologies=[
      {
        select='by_name',
      },
    ],
    density_key='example',
  ),
  plotting=Plotting(
    display_name='example',
    color='example',
    opacity=1.0,
  ),
  entity=False,
)
```

JSON

```json
{
    "name": "example",
    "spatial": {
        "radius": 3.14,
        "density": 3.14,
        "planar_density": 3.14,
        "count_ratio": 3.14,
        "density_ratio": 3.14,
        "relative_to": null,
        "count": 42,
        "geometry": {
            "name_of_the_thing": true
        },
        "morphologies": [
            {
```

(continues on next page)

(continued from previous page)

```
                "select": "by_name"
            }
        ],
        "density_key": "example"
    },
    "plotting": {
        "display_name": "example",
        "color": "example",
        "opacity": 1.0
    },
    "entity": false
}
```

YAML

```
entity: false
name: example
plotting:
  color: example
  display_name: example
  opacity: 1.0
spatial:
  count: 42
  count_ratio: 3.14
  density: 3.14
  density_key: example
  density_ratio: 3.14
  geometry:
    name_of_the_thing: true
  morphologies:
  - select: by_name
  planar_density: 3.14
  radius: 3.14
  relative_to: null
```

- *entity*: Indicates whether this cell type is an abstract entity, or a regular cell.
- *spatial*: Node for spatial information about the cell.
  - *radius*: Radius of the indicative cell soma (`m`).
  - *count*: Fixed number of cells to place.
  - *density*: Volumetric density of cells (`1/(m^3)`)
  - *planar_density*: Planar density of cells (`1/(m^2)`)
  - *density_key*: Key of the *data column* that holds the per voxel density information when this cell type is placed in a *voxel partition*.
  - *relative_to*: Reference to another cell type whose spatial information determines this cell type's number.
  - *density_ratio*: Ratio of densities to maintain with the related cell type.
  - *count_ratio*: Ratio of counts to maintain with the related cell type.

  - *geometry*: Node for geometric information about the cell. This node may contain arbitrary keys and values, useful for cascading custom placement strategy attributes.

- *morphologies*: List of morphology selectors.
- *plotting*:
  - *display_name*: Name used for this cell type when plotting it.
  - *color*: Color used for the cell type when plotting it.
  - *opacity*: Opacity (non-transparency) of the *color*

9.1.8 Placement

Note: Placement nodes are components and can contain additional component-specific attributes.

PYTHON

```
from bsb.placement.strategy import PlacementStrategy
from bsb.placement.indicator import PlacementIndications
from bsb.morphologies.selector import MorphologySelector
from bsb.placement.distributor import RotationDistributor, DistributorsNode,␣
↪MorphologyDistributor, Distributor

PlacementStrategy(
  name='example',
  cell_types=[],
  partitions=[],
  overrides=PlacementIndications(
    radius=3.14,
    density=3.14,
    planar_density=3.14,
    count_ratio=3.14,
    density_ratio=3.14,
    relative_to=None,
    count=42,
    geometry=True,
    morphologies=[
      {
        select='by_name',
      },
    ],
    density_key='example',
  ),
  depends_on=[],
  distribute=DistributorsNode(
    morphologies=MorphologyDistributor(
      strategy='random',
      may_be_empty=False,
    ),
    rotations=RotationDistributor(
```

(continues on next page)

(continued from previous page)

```
      strategy='none',
    ),
    properties=Distributor(
      strategy='example',
    ),
  ),
  strategy='example',
)
```

JSON

```
{
    "name": "example",
    "cell_types": [],
    "partitions": [],
    "overrides": {
        "name_of_the_thing": {
            "radius": 3.14,
            "density": 3.14,
            "planar_density": 3.14,
            "count_ratio": 3.14,
            "density_ratio": 3.14,
            "relative_to": null,
            "count": 42,
            "geometry": {
                "name_of_the_thing": true
            },
            "morphologies": [
                {
                    "select": "by_name"
                }
            ],
            "density_key": "example"
        }
    },
    "depends_on": [],
    "distribute": {
        "morphologies": {
            "strategy": "random",
            "may_be_empty": false
        },
        "rotations": {
            "strategy": "none"
        },
        "properties": {
            "strategy": "example"
        }
    },
    "strategy": "example"
}
```

YAML

```yaml
cell_types: []
depends_on: []
distribute:
  morphologies:
    may_be_empty: false
    strategy: random
  properties:
    strategy: example
  rotations:
    strategy: none
name: example
overrides:
  name_of_the_thing:
    count: 42
    count_ratio: 3.14
    density: 3.14
    density_key: example
    density_ratio: 3.14
    geometry:
      name_of_the_thing: true
    morphologies:
    - select: by_name
    planar_density: 3.14
    radius: 3.14
    relative_to: null
partitions: []
strategy: example
```

- *strategy*: Class name of the placement strategy algorithm to import.
- *cell_types*: List of cell type references. This list is used to gather placement indications for the underlying strategy. It is the underlying strategy that determines how they will interact, so check the component documentation. For most strategies, passing multiple cell types won't yield functional differences from having more cells in a single type.
- *partitions*: List of partitions to place the cell types in. Each strategy has their own way of dealing with partitions, but most will try to voxelize them (using *chunk_to_voxels()*), and combine the voxelsets of each partition. When using multiple partitions, you can save memory if all partitions voxelize into regular same-size voxelsets.
- *overrides*: Cell types define their own placement indications in the *spatial* node, but they might differ depending on the location they appear in. For this reason, each placement strategy may override the information per cell type. Specify the name of the cell types as the key, and provide a dictionary as value. Each key in the dictionary will override the corresponding cell type key.

9.1.9 Connectivity

Note: Connectivity nodes are components and can contain additional component-specific attributes.

PYTHON

```
from bsb.connectivity.strategy import Hemitype, ConnectionStrategy

ConnectionStrategy(
  name='example',
  presynaptic=Hemitype(
    cell_types=[],
    labels=[],
    morphology_labels=[],
    morpho_loader='bsb.connectivity.strategy.<lambda>',
  ),
  postsynaptic=Hemitype(
    cell_types=[],
    labels=[],
    morphology_labels=[],
    morpho_loader='bsb.connectivity.strategy.<lambda>',
  ),
  depends_on=[],
  output_naming={},
  strategy='example',
)
```

JSON

```
{
    "name": "example",
    "presynaptic": {
        "cell_types": [],
        "labels": [],
        "morphology_labels": [],
        "morpho_loader": "bsb.connectivity.strategy.<lambda>"
    },
    "postsynaptic": {
        "cell_types": [],
        "labels": [],
        "morphology_labels": [],
        "morpho_loader": "bsb.connectivity.strategy.<lambda>"
    },
    "depends_on": [],
    "output_naming": {},
    "strategy": "example"
}
```

YAML

```yaml
depends_on: []
name: example
output_naming: {}
postsynaptic:
  cell_types: []
  labels: []
  morpho_loader: bsb.connectivity.strategy.<lambda>
  morphology_labels: []
presynaptic:
  cell_types: []
  labels: []
  morpho_loader: bsb.connectivity.strategy.<lambda>
  morphology_labels: []
strategy: example
```

- *strategy*: Class name of the connectivity strategy algorithm to import.
- *presynaptic*/*postsynaptic*: Hemitype node specificatiosn for the pre/post synaptic side of the synapse.
  - *cell_types*: List of cell type references. It is the underlying strategy that determines how they will interact, so check the component documentation. For most strategies, all the presynaptic cell types will be cross combined with all the postsynaptic cell types.

9.1.10 Simulations

PYTHON

```python
from bsb.simulation.simulation import Simulation
from bsb.simulation.cell import CellModel
from bsb.simulation.parameter import Parameter, ParameterValue
from bsb.simulation.connection import ConnectionModel
from bsb.simulation.device import DeviceModel

Simulation(
  name='example',
  duration=3.14,
  cell_models=CellModel(
    name='example',
    cell_type=None,
    parameters=[
      {
        value=ParameterValue(
          type='example',
        ),
        type='example',
      },
    ],
  ),
  connection_models=ConnectionModel(
    name='example',
    tag='example',
```

(continues on next page)

(continued from previous page)

```
  ),
  devices=DeviceModel(
    name='example',
  ),
  post_prepare=[
    {
  ],
  simulator='example',
)
```

JSON

```
{
    "name": "example",
    "duration": 3.14,
    "cell_models": {
        "name": "example",
        "cell_type": null,
        "parameters": [
            {
                "value": {
                    "type": "example"
                },
                "type": "example"
            }
        ]
    },
    "connection_models": {
        "name": "example",
        "tag": "example"
    },
    "devices": {
        "name": "example"
    },
    "post_prepare": [
        null
    ],
    "simulator": "example"
}
```

YAML

```
cell_models:
  cell_type: null
  name: example
  parameters:
  - type: example
    value:
      type: example
```

(continues on next page)

(continued from previous page)

```
connection_models:
  name: example
  tag: example
devices:
  name: example
duration: 3.14
name: example
post_prepare:
- null
simulator: example
```

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 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

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*.

11.1 Create a project

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

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.
- `quickstart`: Generates an exemplary project with basic config that can be compiled.
- `exists`: With this flag, it is not an error for the `parent-folder` to exist.

11.2 Create 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.

11.3 Compiling a network

```
bsb [OPTIONS] compile [my-config.json] [COMPILE-FLAGS]
```

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.

Flags

- `-x`, `-y`, `-z`: Size hints of the 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.
- `-p`, `--plot`: Plot the created network.

Storage flags

These flags decide what to do with existing data.

- `-w`, `--clear`: Clear all data found in the storage object, and overwrite with new data.
- `-a`, `--append`: Append the new data to the existing data.
- `-r`, `--redo`: Clear all data that is involved in the strategies that are being executed, and replace it with the new data.

Phase flags

These flags control which phases and strategies to execute or ignore.

- `--np`, `--skip-placement`: Skip the placement phase.
- `--nap`, `--skip-after-placement`: Skip the after-placement phase.
- `--nc`, `--skip-connectivity`: Skip the connectivity phase.
- `--nac`, `--skip-after-connectivity`: Skip the after-connectivity phase.
- `--skip`: Name of a strategy to skip. You may pass this flag multiple times, or give a comma separated list of names.
- `--only`: Name of a strategy to run, skipping all other strategies. You may pass this flag multiple times, or give a comma separated list of names.

11.4 Run 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.

11.5 Check the global cache

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

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

CHAPTER

TWELVE

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.

12.1 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*.

12.2 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
```

12.3 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_option()*:

```
from bsb import store_option

store_option("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_option()*:

```
from bsb import read_option

link = read_option("networks.config_link")
```

12.4 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`.

12.5 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`

12.6 `pyproject.toml` structure

The BSB's project-wide settings are all stored in `pyproject.toml` under `tools.bsb`:

```
[tools.bsb]
config = "network_configuration.json"
```

12.6.1 Writing your own options

You can create your own options as a *plugin* by defining a class that inherits from *BsbOption*:

```
from bsb import BsbOption, 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*.

CHAPTER

THIRTEEN

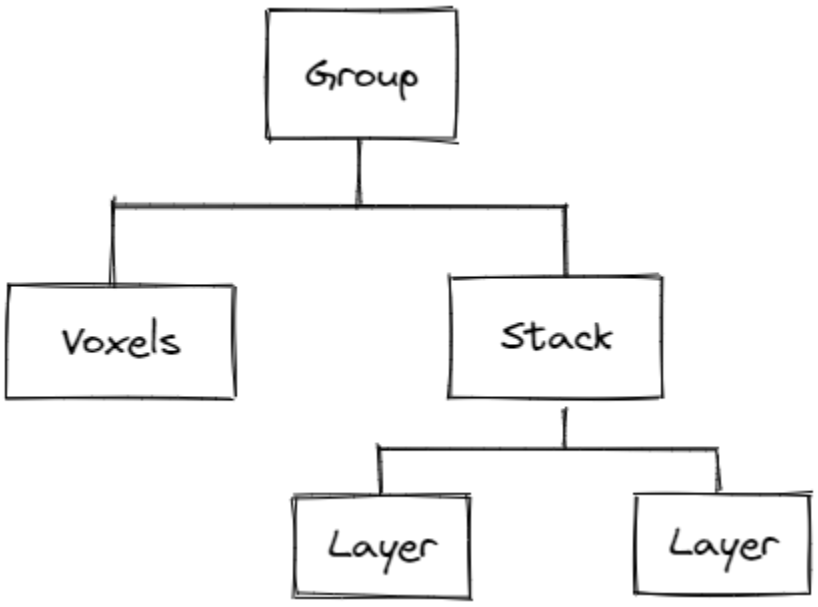
INTRODUCTION

13.1 Layouts

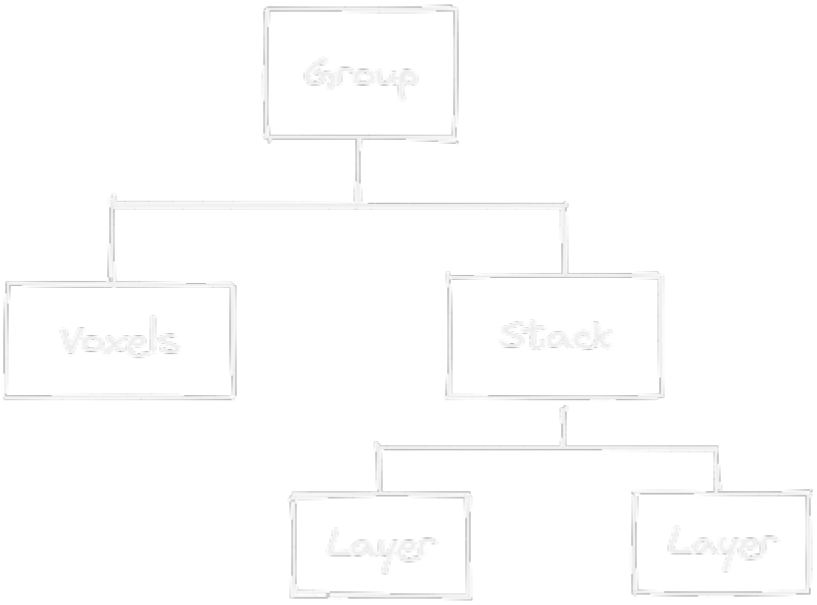
The topology module allows you to make abstract descriptions of the spatial layout of pieces of the region you are modelling. *Partitions* define shapes such as layers, cubes, spheres, and meshes. *Regions* put partitions together by arranging them hierarchically. The topology is formed as a tree of regions, that end downstream in a terminal set of partitions.

To initiate the topology, the network size hint is passed to the root region, which subdivides it for their children to make an initial attempt to lay themselves out. Once handed back the initial layouts of their children, parent regions can propose transformations to finalize the layout. If any required transformation proposals fail to meet the configured constraints, the layout process fails.

13.1.1 Example



The root *Group* receives the network X, Y, and Z. A `Group` is an inert region and simply passes the network boundaries on to its children. The *Voxels* loads its voxels, and positions them absolutely, ignoring the network boundaries. The *Stack* passes the volume on to the *Layers* who fill up the space and occupy their thickness. They return their layout up to the parent `Stack`, who in turn proposes translations to the layers in order to stack them on top of the other. The end



result is stack beginning from the network starting corner, with 2 layers as large as the network, with their respective thickness, and absolutely positioned voxels.

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*. 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 *NrrdVoxels*. By default it will load all the nonzero values in a source file:

JSON

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

PYTHON

```
from bsb import NrrdVoxels

my_voxel_partition = NrrdVoxels(source="data/my_nrrd_data.nrrd", voxel_size=25)
```

The nonzero values from the `data/my_nrrd_data.nrrd` file will be included in the *VoxelSet*, and their values will be stored on the voxelset as a *data column*. Data columns can be accessed through the *data* property:

```
voxels = NrrdVoxels(source="data/my_nrrd_data.nrrd", voxel_size=25)
vs = voxels.get_voxelset()
# Prints the information about the VoxelSet, like how many voxels there are etc.
print(vs)
```

(continues on next page)

(continued from previous page)

```
# Prints an (Nx1) array with one nonzero value for each voxel.
print(vs.data)
```

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": {
      "type": "nrrd",
      "mask_value": 55,
      "mask_source": "data/brain_structures.nrrd",
      "source": "data/whole_brain_cell_densities.nrrd",
      "voxel_size": 25
    }
  }
}
```

PYTHON

```
from bsb import NrrdVoxels

partition = NrrdVoxels(
  mask_value=55,
  mask_source="data/brain_structures.nrrd",
  source="data/whole_brain_cell_densities.nrrd",
  voxel_size=25,
)
vs = partition.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)
```

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": {
      "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 import NrrdVoxels

partition = NrrdVoxels(
  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 = partition.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)
```

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": {
      "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 import NrrdVoxels

partition = NrrdVoxels(
  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 = partition.get_voxelset()
# Access data columns as strings
print(vs.data[:, "type1"])
# Index multiple columns like this:
print(vs.data[:, "type1", "type3"])
```

15.1.2 Allen Mouse Brain Atlas integration

The Allen Institute for Brain Science (`AIBS`) gives free access, through their website, to thousands of datasets based on experiments on mice and humans.

For the mouse, these datasets are 3D-registered in a Common Coordinate Framework (CCF). The `AIBS` maintains the Allen Mouse Brain Atlas; a pair of files which defines a mouse brain region ontology, and its spatial segregation in the `CCF`:

- The brain region ontology takes the form of a hierarchical tree of brain region, with the root (top parent) region defining the borders of the mouse brain and the leafs its finest parcellations. It will be later be called `Allen Mouse Brain Region Hierarchy` (`AMBRH`) Each brain region in the `AMBRH` has a unique `id`, `name`, and `acronym` which can all be used to refer to the region.
- They also defined a mouse brain `Annotation volume` (NRRD file) which provides for each voxel of the `CCF` the id of the finest region it belongs to according to the brain region ontology.

With the BSB you can be seamlessly integrate any dataset registered in the Allen Mouse Brain CCF into your workflow using the *`AllenStructure`*. By default (*mask_volume* is not specified), the *`AllenStructure`* leverages the 2017 version of the `CCFv3 Annotation volume`, which it downloads directly from the Allen website. BSB will also automatically download the `AMBRH` that you can use to filter regions, providing any of the brain region id, name or acronym identifiers.

You can then download any Allen Atlas registered dataset as a local NRRD file, and associate it to the structure, by specifying it as a source file (through *source* or *sources*). The `Annotation volume` will be converted to a voxel mask, and the mask will be applied to your source files, thereby selecting the structure from the source files. Each source file will be converted into a data column on the voxelset:

JSON

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

PYTHON

```
from bsb import AllenStructure

partition = AllenStructure(
  # Loads the "ventroanterolateral thalamic nucleus" from the
  # Allen Mouse Brain Annotation volume
  struct_name="VAL",
  mask_source="data/brain_structures.nrrd",
  sources=[
```

(continues on next page)

(continued from previous page)

```
    "data/allen_gene_expression_25.nrrd",
  ],
  keys=["expression"],
)
print("Gene expression values per voxel:", partition.voxelset.expression)
```

CHAPTER

SIXTEEN

CELL TYPES

A cell types contains information about cell populations. There are 2 categories: cells, and entities. A cell has a position, while an entity does not. Cells can also have morphologies and orientations associated with them. On top of that, both cells and entities support additional arbitrary properties.

A cell type is an abstract description of the population. During placement, the concrete data is generated in the form of a *PlacementSet*. These can then be connected together into *ConnectivitySets*. Furthermore, during simulation, cell types are represented by **cell models**.

Basic configuration

The *radius* and *density* are the 2 most basic *placement indicators*, they specify how large and dense the cells in the population generally are. The *plotting* block allows you to specify formatting details.

```
{
  "cell_types": {
    "my_cell_type": {
      "spatial": {
        "radius": 10.0,
        "density": 3e-9
      },
      "plotting": {
        "display_name": "My Cell Type",
        "color": "pink",
        "opacity": 1.0
      }
    }
  }
}
```

Specifying spatial density

You can set the spatial distribution for each cell type present in a *NrrdVoxels* partition.

To do so, you should first attach your nrrd volumetric density file(s) to the partition with either the *source* or *sources* blocks. Then, label the file(s) with the *keys* list block and refer to the *keys* in the *cell_types* with *density_key*:

```
{
  "partitions": {
    "declive": {
      "type": "nrrd",
```

(continues on next page)

(continued from previous page)

```
      "sources": ["first_cell_type_density.nrrd",
                  "second_cell_type_density.nrrd"],
      "keys": ["first_cell_type_density",
               "second_cell_type_density"]
      "voxel_size": 25,
    }
  }
  "cell_types": {
    "first_cell_type": {
      "spatial": {
        "radius": 10.0,
        "density_key": "first_cell_type_density"
      },
      "plotting": {
        "display_name": "First Cell Type",
        "color": "pink",
        "opacity": 1.0
      }
    },
    "first_cell_type": {
      "spatial": {
        "radius": 5.0,
        "density_key": "second_cell_type_density"
      },
      "plotting": {
        "display_name": "Second Cell Type",
        "color": "#0000FF",
        "opacity": 0.5
      }
    }
  }
}
```

The nrrd files should contain voxel based volumetric density in unit of cells / voxel volume, where the voxel volume is in cubic unit of *voxel_size*. i.e., if *voxel_size* is in µm then the density file is in cells/µm^3.

Specifying morphologies

If the cell type is represented by morphologies, you can list multiple `selectors` to fetch them from the *Morphology repositories*.

```
{
  "cell_types": {
    "my_cell_type": {
      "spatial": {
        "radius": 10.0,
        "density": 3e-9,
        "morphologies": [
          {
            "select": "by_name",
            "names": ["cells_A_*", "cell_B_2"]
```

(continues on next page)

(continued from previous page)

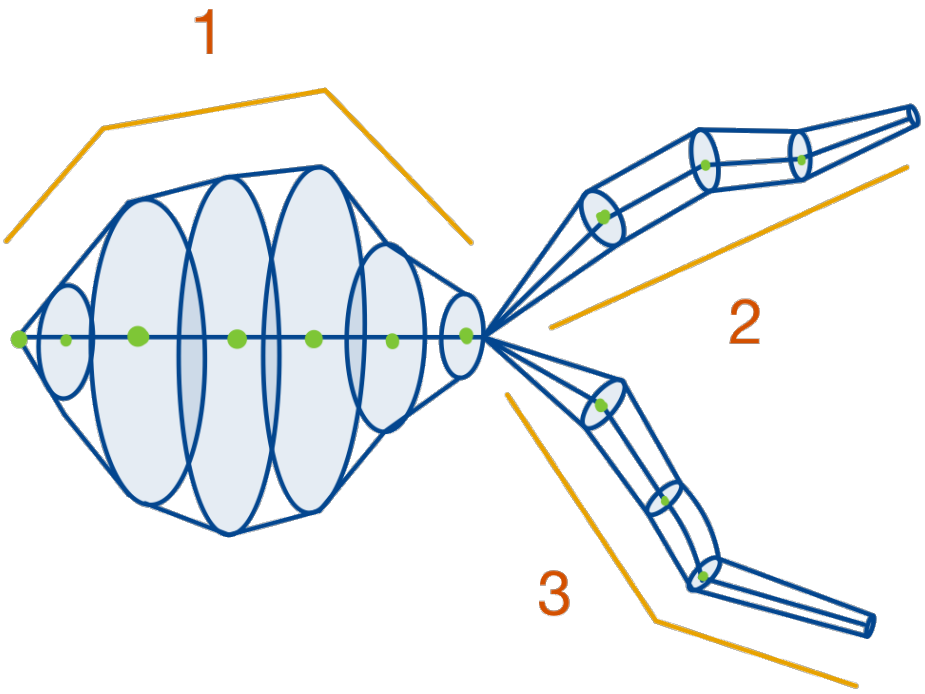
```
          }
        ]
      },
      "plotting": {
        "display_name": "My Cell Type",
        "color": "pink",
        "opacity": 1.0
      }
    }
  }
}
```

CHAPTER

SEVENTEEN

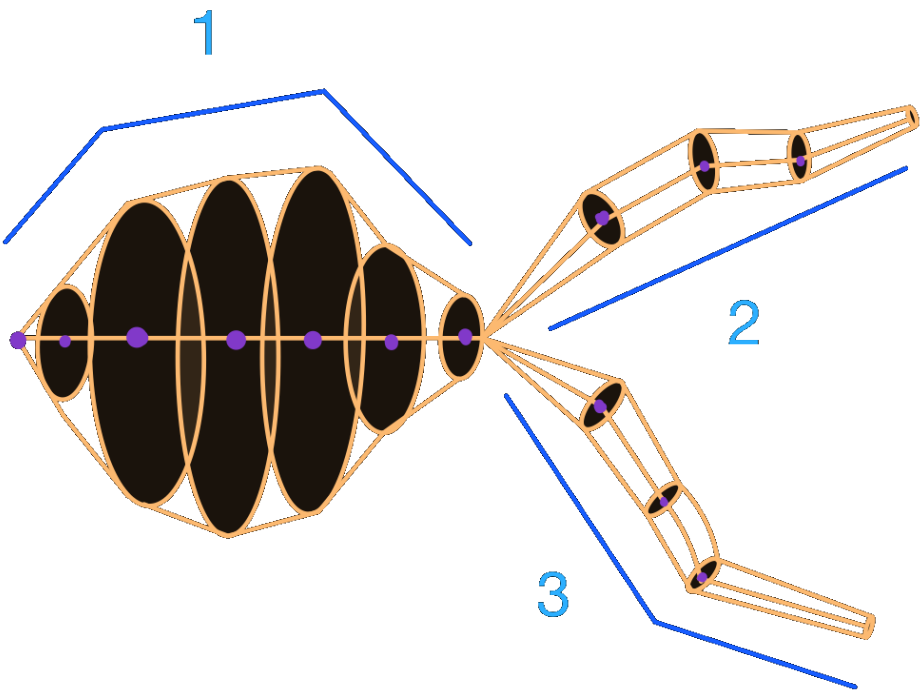
MORPHOLOGIES

Morphologies are the 3D representation of a cell. A morphology consists of head-to-tail connected branches, and branches consist of a series of points with radii. Points can be labelled and can have multiple user-defined properties per point.



1. The root branch, shaped like a soma because of its radii.
2. A child branch of the root branch.
3. Another child branch of the root branch.

Network configurations can contain a *morphologies* key to define the morphologies that should be processed and assigned to cells. See *Adding morphologies* for a guide on the possibilities. Morphologies can be stored in a network in


1
2
3

the *MorphologyRepository*.

17.1 Parsing morphologies

A morphology file can be parsed with *parse_morphology_file()*, if you already have the content of a file you can pass that directly into *parse_morphology_content()*:

```
from bsb import parse_morphology_file

morpho = parse_morphology_file("./my_file.swc")
```

Important: The default parser only supports SWC files. Use the `morphio` parser for ASC files.

There are many different formats and even multiple conventions per format for parsing morphologies. To support these diverse approaches the framework provides configurable *Morphology parsers*. You can pass the type of parser and additional arguments:

```
from bsb import parse_morphology_file

morpho = parse_morphology_file("./my_file.swc", parser="morphio", flags=["no_duplicates
↪"])
```

Once we have our *Morphology* object we can save it in *Storage*; storages and networks have a `morphologies` attribute that links to a *MorphologyRepository* that can save and load morphologies:

```
from bsb import Storage

store = Storage("hdf5", "morphologies.hdf5")
store.morphologies.save("MyCell", morpho)
```

17.2 Constructing morphologies

Create your branches, attach them in a parent-child relationship, and provide the roots to the *Morphology* constructor:

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

root = Branch(
  # XYZ
  np.array([
    [0, 1, 2],
    [0, 1, 2],
    [0, 1, 2],
  ]),
  # radius
  np.array([1, 1, 1]),
)
child_branch = Branch(
  np.array([
```

(continues on next page)

(continued from previous page)

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

17.3 Basic use

Morphologies and branches contain spatial data in the `points` and `radii` attributes. Points can be individually labelled with arbitrary strings, and additional properties for each point can be assigned to morphologies/branches:

```
from bsb import from_storage

# Load the morphology
network = from_storage("network.hdf5")
morpho = network.morphologies.load("my_morphology")
print(f"Has {len(morpho)} points and {len(morpho.branches)} branches.")
```

Once loaded we can do *transformations*, label or assign properties on the morphology:

```
# Take a branch
special_branch = morpho.branches[3]
# Assign some labels to the whole branch
special_branch.label(["axon", "special"])
# Assign labels only to the first quarter of the branch
first_quarter = np.arange(len(special_branch)) < len(special_branch) / 4
special_branch.label(["initial_segment"], first_quarter)
# Assign random data as the `random_data` property to the branch
special_branch.set_property(random_data=np.random.random(len(special_branch)))
print(f"Random data for each point:", special_branch.random_data)
```

Once you're done with the morphology you can save it again:

```
network.morphologies.save("processed_morphology", morpho)
```

Note: You can assign as many labels as you like (2^64 combinations max)! Labels'll cost you almost no memory or disk space! You can also add as many properties as you like, but they'll cost you memory and disk space per point on the morphology.

Labels

Branches or points can be labelled, and pieces of the morphology can be selected by their label. Labels are also useful targets to insert biophysical mechanisms into parts of the cell later on in simulation.

```
import numpy as np

from bsb import from_storage

# Load the morphology
network = from_storage("network.hdf5")
morpho = network.morphologies.load("my_morphology")

# Filter branches
big_branches = [b for b in morpho.branches if np.any(b.radii > 2)]
for b in big_branches:
    # Label all points on the branch as a `big_branch` point
    b.label(["big_branch"])
    if b.is_terminal:
        # Label the last point on terminal branches as a `tip`
        b.label(["tip"], [-1])

network.morphologies.save("labelled_morphology", morpho)
```

Properties

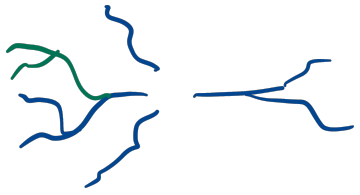
Branches and morphologies can be given additional properties. The basic properties are `x`, `y`, `z`, `radii` and `labels`. You can pass additional properties to the `properties` argument of the *Branch* constructor. They will be automatically joined on the morphology.

17.4 Subtree transformations

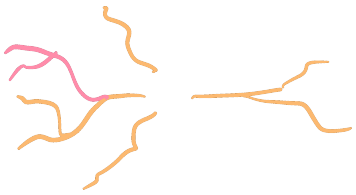
A subtree is a (sub)set of a morphology defined by a set of *roots* and all of its downstream branches (i.e. the branches *emanating* from a set of roots). A subtree with roots equal to the roots of the morphology is equal to the entire morphology, and all transformations valid on a subtree are also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

17.4.1 Creating subtrees

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

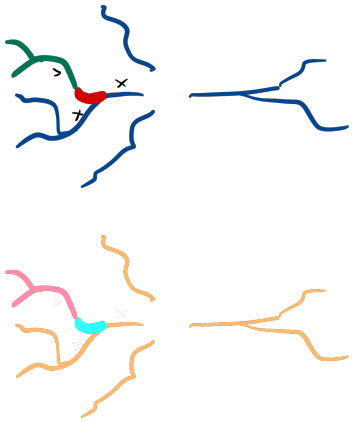


```
axon = morfo.subtree("axon")
# Multiple labels can be given
hybrid = morfo.subtree("proximal", "distal")
```



Warning: Branches will be selected as soon as they have one or more points labelled with a selected label.

Selections will always include all the branches emanating (downtree) from the selection as well:



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

17.4.2 Translation

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

17.4.3 Centering

Subtrees may *center()* themselves so that the point `(0, 0, 0)` becomes the geometric mean of the roots.

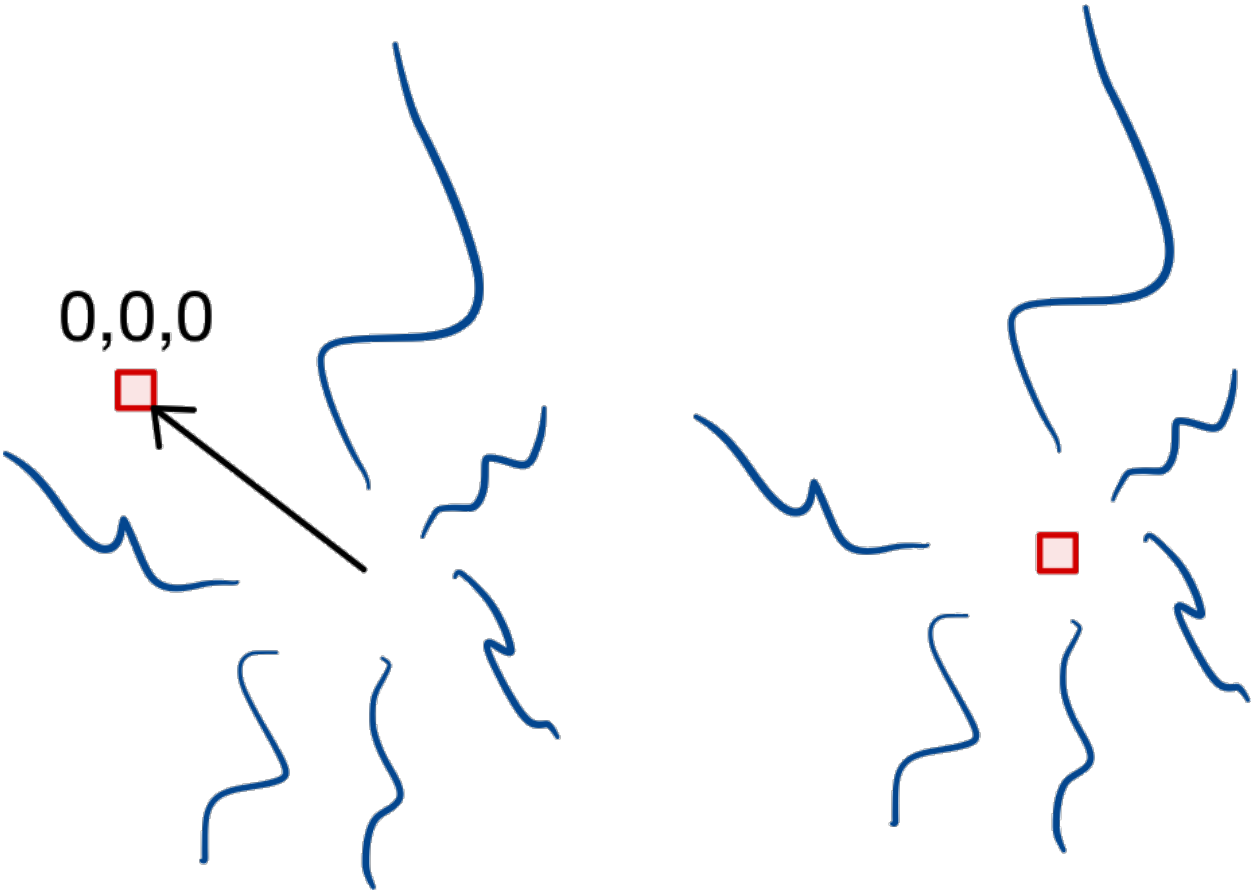
17.4.4 Rotation

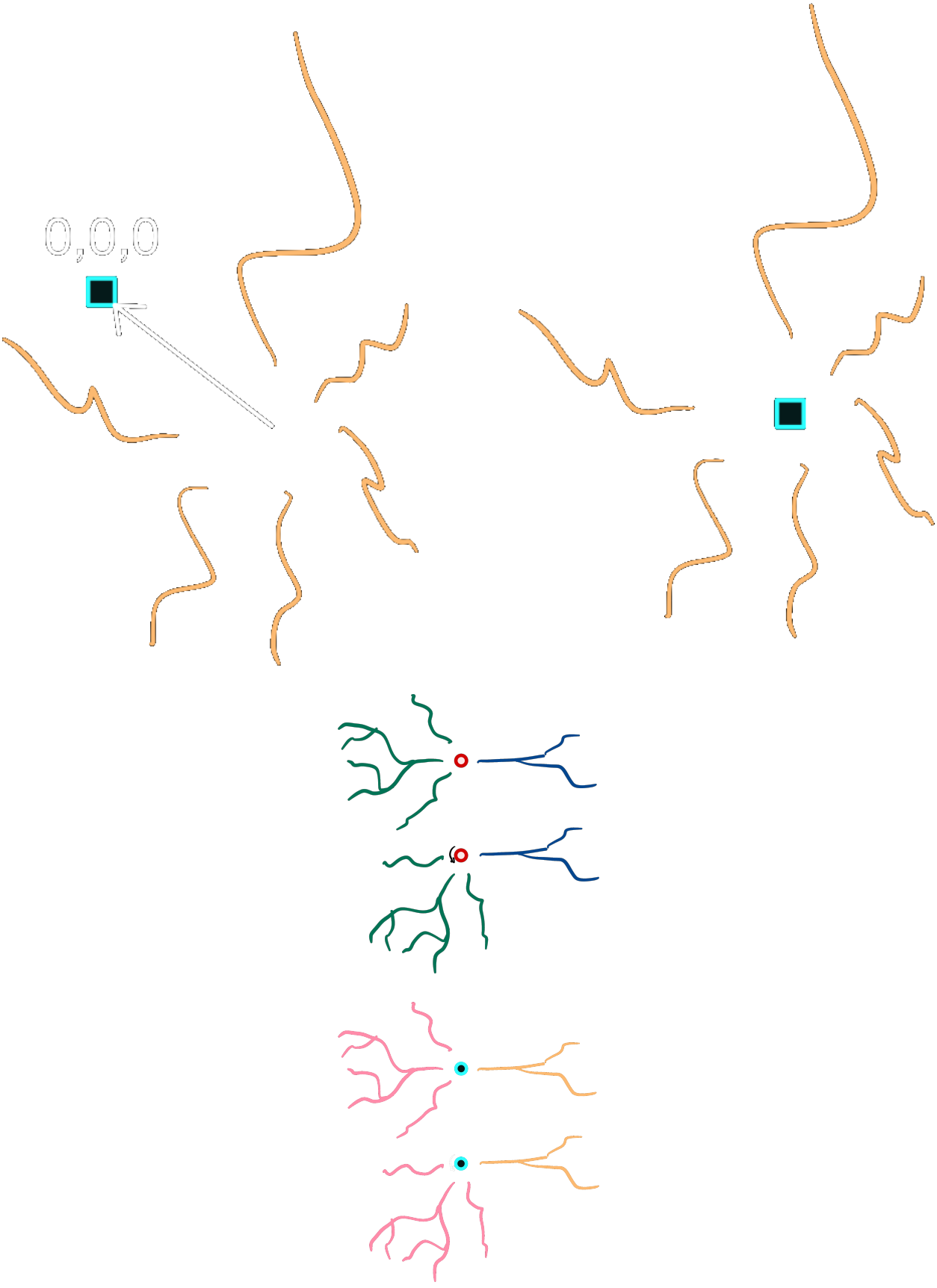
Subtrees may be *rotated* around a singular point, by giving a `Rotation` (and a center, by default 0):

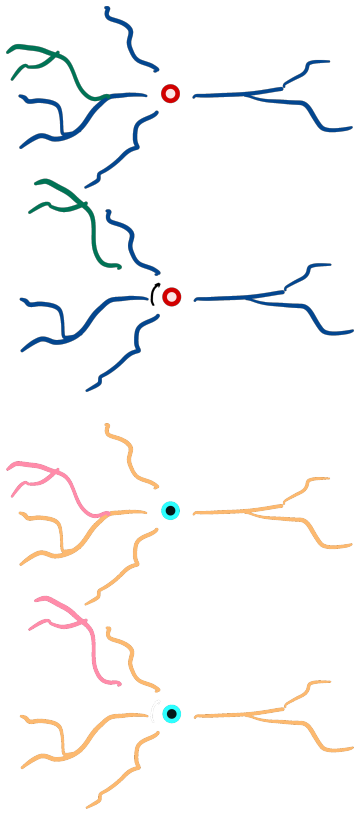
```
from scipy.spatial.transform import Rotation

r = Rotation.from_euler("xy", [90, 90], degrees=True)
dendrites.rotate(r)
```

```
dendrite.rotate(r)
```

Note that this creates a gap, because we are rotating around the center, root-rotation might be preferred here.

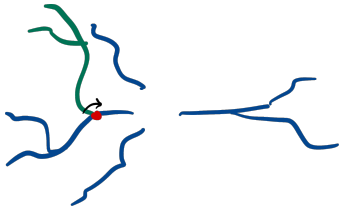

0,0,0


0,0,0



17.4.5 Root-rotation

Subtrees may be *root-rotated* around each respective root in the tree:



```
dendrite.root_rotate(r)
```

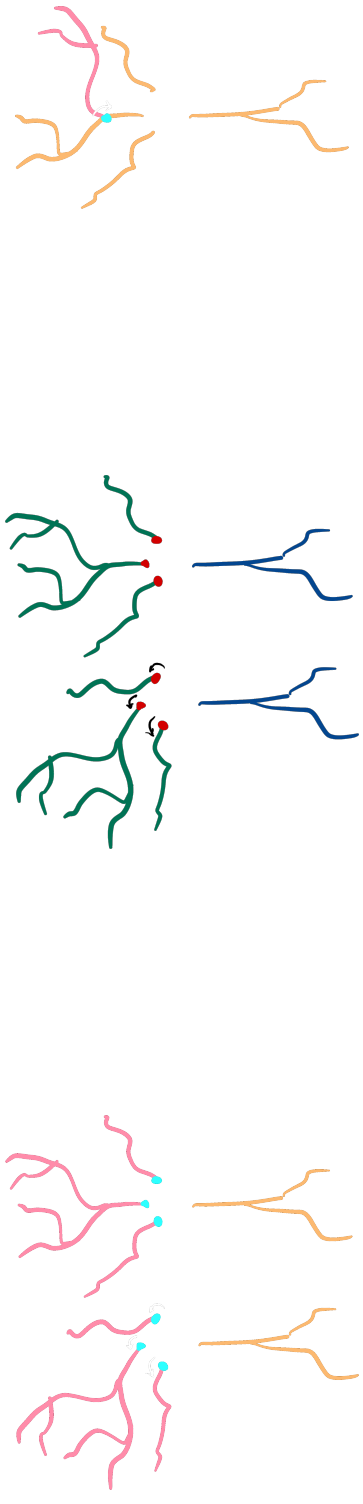
```
dendrites.root_rotate(r)
```

Additionally, you can *root-rotate* from a point of the subtree instead of its root. In this case, points starting from the point selected will be rotated.

To do so, set the *downstream_of* parameter with the index of the point of your interest.

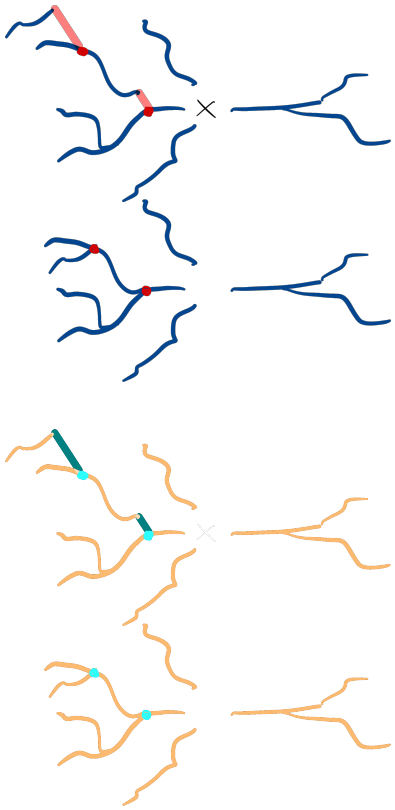
```
# rotate all points after the second point in the subtree
# i.e.: points at index 0 and 1 will not be rotated.
dendrites.root_rotate(r, downstream_of=2)
```

Note: This feature can only be applied to subtrees with a single root



17.4.6 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. Gaps _between_ roots are never collapsed.

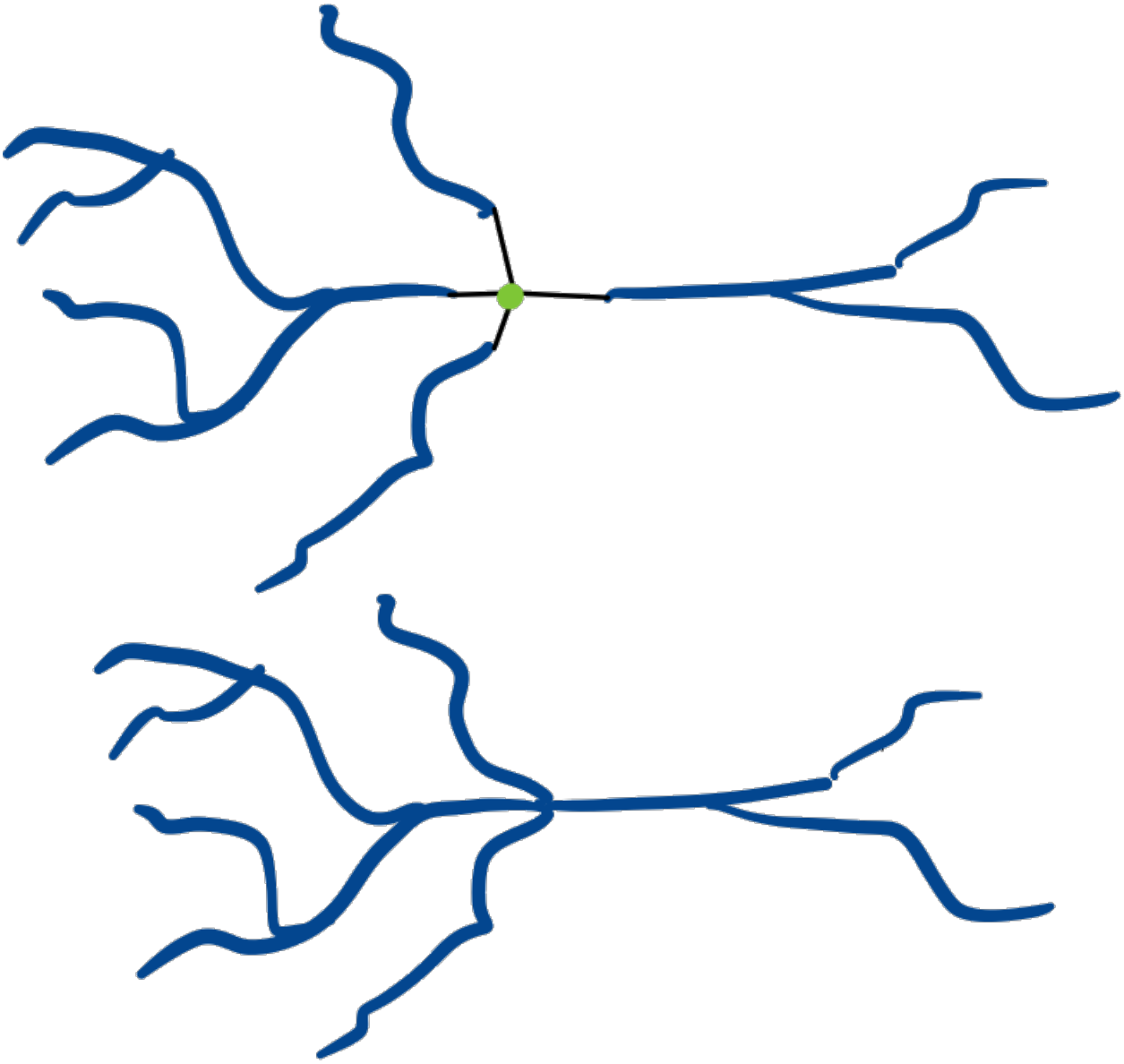
See also:

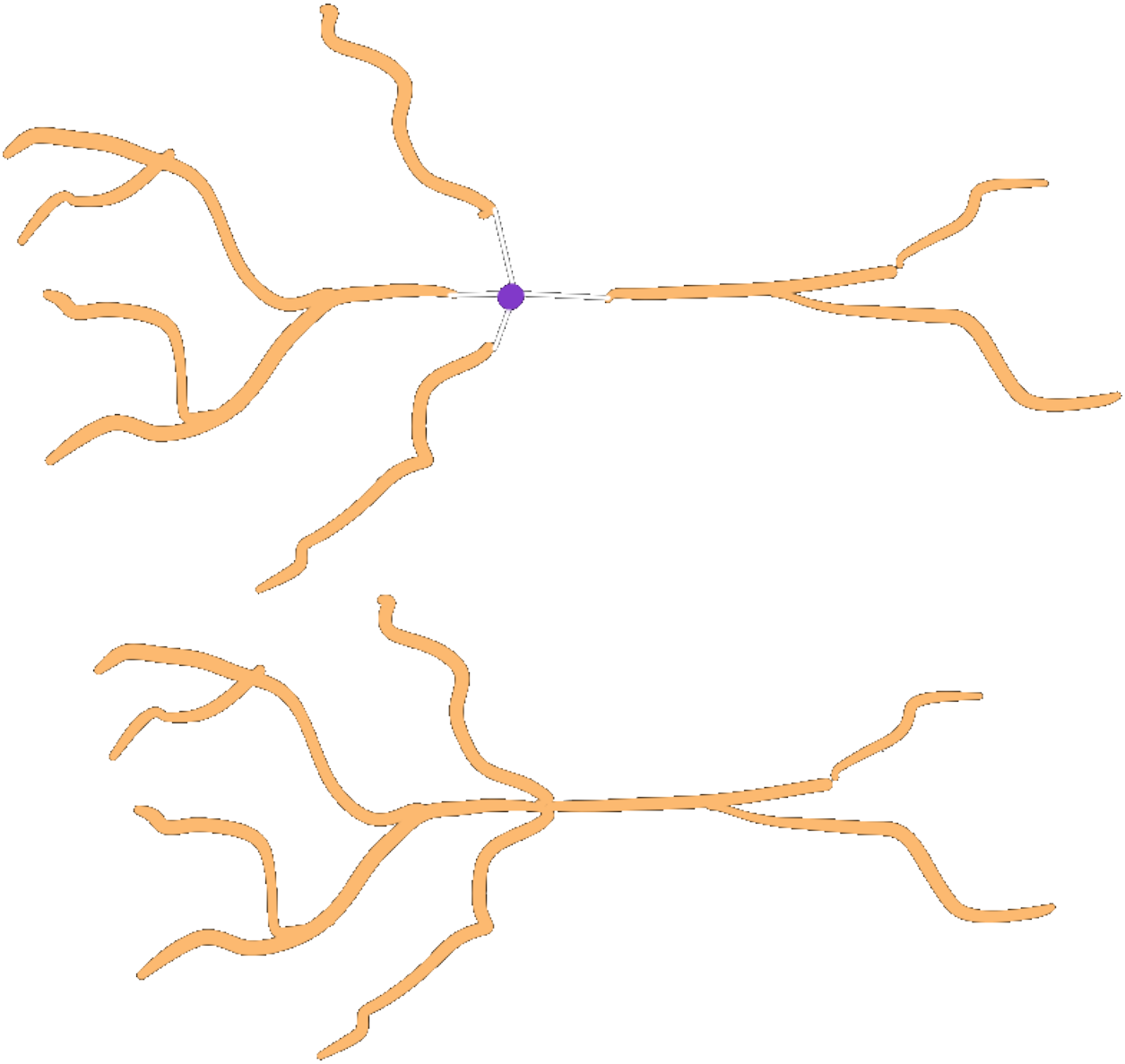
Collapsing

17.4.7 Collapsing

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

```
roots.collapse()
```





Call chaining

Calls to any of the above functions can be chained together:

```
dendrites.close_gaps().center().rotate(r)
```

17.5 Advanced features

17.5.1 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 *load()* method to load the *Morphology* object from storage and a *get_meta()* method to return the metadata.

17.5.2 Morphology selectors

The most common way of telling the framework which morphologies to use is through *MorphologySelectors*. Currently you can select morphologies `by_name` or `from_neuromorpho`:

JSON

```
"morphologies": [
  {
    "select": "by_name",
    "names": ["my_morpho_1", "all_other_*"]
  },
  {
    "select": "from_neuromorpho",
    "names": ["H17-03-013-11-08-04_692297214_m", "cell010_GroundTruth"]
  }
]
```

If you want to make your own selector, you should implement the *validate()* and *pick()* methods.

`validate` can be used to assert that all the required morphologies and metadata are present, while `pick` needs to return `True`/`False` to include a morphology in the selection. Both methods are handed *StoredMorphology* objects. Only *load()* morphologies if it is impossible to determine the outcome from the metadata alone.

The following example creates a morphology selector selects morphologies based on the presence of a user defined metadata `"size"`:

```
from bsb import config, MorphologySelector

@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):
```

(continues on next page)

(continued from previous page)

```
      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
```

After installing your morphology selector as a plugin, you can use `by_size` as selector:

JSON

```
{
  "cell_type_A": {
    "spatial": {
      "morphologies": [
        {
          "select": "by_size",
          "min_size": 35
        }
      ]
    }
  }
}
```

PYTHON

```
network.cell_types.cell_type_A.spatial.morphologies = [MySizeSelector(min_size=35)]
```

17.5.3 Morphology metadata

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

17.5.4 Morphology distributors

A *MorphologyDistributor* is a special type of *Distributor* that is called after positions have been generated by a *PlacementStrategy* to assign morphologies, and optionally rotations. The *distribute()* method is called with the partitions, the indicators for the cell type and the positions; the method has to return a *MorphologySet* or a tuple together with a *RotationSet*.

Warning: The rotations returned by a morphology distributor may be overruled when a *RotationDistributor* is defined for the same placement block.

Distributor configuration

Each *placement* block may contain a *DistributorsNode*, which can specify the morphology and/or rotation distributors, and any other property distributor:

JSON

```
{
  "placement": {
    "placement_A": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["cell_A"],
      "partitions": ["layer_A"],
      "distribute": {
        "morphologies": {
          "strategy": "roundrobin"
        }
      }
    }
  }
}
```

PYTHON

```
from bsb import RoundRobinMorphologies

network.placement.placement_A.distribute.morphologies = RoundRobinMorphologies()
```

Distributor interface

The generic interface has a single function: `distribute(positions, context)`. The `context` contains `.partitions` and `.indicator` for additional placement context. The distributor must return a dataset of N floats, where N is the number of `positions` you've been given, so that it can be stored as an additional property on the cell type.

The morphology distributors have a slightly different interface, and receive an additional `morphologies` argument: `distribute(positions, morphologies, context)`. The morphologies are a list of *StoredMorphology*, that the user has configured to use for the cell type under consideration and that the distributor should consider the input, or template morphologies for the operation.

The morphology distributor is supposed to return an array of N integers, where each integer refers to an index in the list of morphologies. e.g.: if there are 3 morphologies, putting a `0` on the n-th index means that cell N will be assigned morphology `0` (which is the first morphology in the list). `1` and `2` refer to the 2nd and 3rd morphology, and returning any other values would be an error.

If you need to break out of the morphologies that were handed to you, morphology distributors are also allowed to return their own *MorphologySet*. Since you're free to pass any list of morphology loaders to create a morphology set, you can put and assign any morphology you like.

Tip: *MorphologySets* work on *StoredMorphologies*! This means that it is your job to save the morphologies into your network first, and to use the returned values of the save operation as input to the morphology set:

```
def distribute(self, positions, morphologies, context):
  # We're ignoring what is given, and make our own morphologies
  morphologies = [Morphology(...) for p in positions]
  # If we pass the `morphologies` to the `MorphologySet`, we create an error.
  # So we save the morphologies, and use the stored morphologies instead.
  loaders = [
    self.scaffold.morphologies.save(f"morpho_{i}", m)
    for i, m in enumerate(morphologies)
  ]
  return MorphologySet(loaders, np.arange(len(loaders)))
```

This is cumbersome, so if you plan on generating new morphologies, use a *morphology generator* instead.

Finally, each morphology distributor is allowed to return an additional argument to assign rotations to each cell as well. The return value must be a *RotationSet*.

Warning: The rotations returned from a morphology distributor may be ignored and replaced by the values of the rotation distributor, if the user configures one.

The following example creates a distributor that selects smaller morphologies the closer the position is to the top of the partition:

```
import numpy as np
from scipy.stats.distributions import norm

from bsb import MorphologyDistributor


class SmallerTopMorphologies(MorphologyDistributor, classmap_entry="small_top"):
    def distribute(self, positions, morphologies, context):
        # Get the maximum Y coordinate of all the partitions boundaries
        top_of_layers = np.maximum([p.data.mdc[1] for p in context.partitions])
        depths = top_of_layers - positions[:, 1]
        # Get all the heights of the morphologies, by peeking into the morphology␣
↪metadata
        msizes = [
            loader.get_meta()["mdc"][1] - loader.get_meta()["ldc"][1]
            for loader in morphologies
        ]
        # Pick deeper positions for bigger morphologies.
        weights = np.column_stack(
            [norm(loc=size, scale=20).pdf(depths) for size in msizes]
        )
        # The columns are the morphology ids, so make an arr from 0 to n morphologies.
        picker = np.arange(weights.shape[1])
        # An array to store the picked weights
        picked = np.empty(weights.shape[0], dtype=int)
        rng = np.default_rng()
        for i, p in enumerate(weights):
            # Pick a value from 0 to n, based on the weights.
            picked[i] = rng.choice(picker, p=p)
```

(continues on next page)

(continued from previous page)

```
        # Return the picked morphologies for each position.
        return picked
```

Then, after installing your distributor as a plugin, you can use `small_top`:

JSON

```
{
  "placement": {
    "placement_A": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["cell_A"],
      "partitions": ["layer_A"],
      "distribute": {
        "morphologies": {
          "strategy": "small_top"
        }
      }
    }
  }
}
```

PYTHON

```
network.placement.placement_A.distribute.morphologies = SmallerTopMorphologies()
```

Morphology generators

Continuing on the morphology distributor, one can also make a specialized generator of morphologies. The generator takes the same arguments as a distributor, but returns a list of *Morphology* objects, and the morphology indices to make use of them. It can also return rotations as a 3rd return value.

This example is a morphology generator that generates a simple stick that drops down to the origin for each position:

```
import numpy as np

from bsb import Branch, Morphology, MorphologyGenerator


class TouchTheBottomMorphologies(MorphologyGenerator, classmap_entry="touchdown"):
    def generate(self, positions, morphologies, context):
        return [
            Morphology([Branch([pos, [pos[1], 0, pos[2]]], [1, 1])]) for pos in positions
        ], np.arange(len(positions))
```

Then, after installing your generator as a plugin, you can use `touchdown`:

JSON

```
{
  "placement": {
    "placement_A": {
      "strategy": "bsb.placement.RandomPlacement",
      "cell_types": ["cell_A"],
      "partitions": ["layer_A"],
      "distribute": {
        "morphologies": {
          "strategy": "touchdown"
        }
      }
    }
  }
}
```

PYTHON

```
network.placement.placement_A.distribute.morphologies = TouchTheBottomMorphologies()
```

17.5.5 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)
```

17.6 Reference

Morphology module

class bsb.morphologies.**Branch**(*points*, *radii*, *labels=None*, *properties=None*, *children=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

`attach_child`(*branch*)

Attach a branch as a child to this branch.

Parameters
branch (*Branch*) – Child branch

`cached_voxelize`(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids and cache the result.

Return type
bsb.voxels.VoxelSet

`ceil_arc_point`(*arc*)

Get the index of the nearest distal arc point.

`center`()

Center the morphology on the origin

`property children`

Collection of the child branches of this branch.

Returns
list of *Branches*

Return type
list

`close_gaps`()

Close any head-to-tail gaps between parent and child branches.

`collapse`(*on=None*)

Collapse all the roots of the morphology or subtree onto a single point.

Parameters
on (*int*) – Index of the root to collapse on. Collapses onto the origin by default.

`contains_labels`(*labels*)

Check if this branch contains any points labelled with any of the given labels.

Parameters
labels (*List[str]*) – The labels to check for.

Return type
bool

`copy`(*branch_class=None*)

Return a parentless and childless copy of the branch.

Parameters
branch_class (*type*) – Custom branch creation class

Returns
A branch, or *branch_class* if given, without parents or children.

Return type
bsb.morphologies.Branch

`delete_point`(*index*)

Remove a point from the branch

Parameters
index (*int*) – index position of the point to remove

Returns
the branch where the point has been removed

Return type
bsb.morphologies.Branch

`detach`()

Detach the branch from its parent, if one exists.

`detach_child`(*branch*)

Remove a branch as a child from this branch.

Parameters
branch (*Branch*) – Child branch

`property end`

Return the spatial coordinates of the terminal point of this branch.

`property euclidean_dist`

Return the Euclidean distance from the start to the terminal point of this branch.

`find_closest_point`(*coord*)

Return the index of the closest on this branch to a desired coordinate.

Parameters
coord – The coordinate to find the nearest point to

Type
numpy.ndarray

`flatten`()

Return the flattened points of the morphology or subtree.

Return type
numpy.ndarray

`flatten_labels`()

Return the flattened labels of the morphology or subtree.

Return type
numpy.ndarray

`flatten_properties`()

Return the flattened properties of the morphology or subtree.

Return type
numpy.ndarray

`flatten_radii`()

Return the flattened radii of the morphology or subtree.

Return type
numpy.ndarray

floor_arc_point(*arc*)

Get the index of the nearest proximal arc point.

property fractal_dim

Return the fractal dimension of this branch, computed as the coefficient of the line fitting the log-log plot of path vs euclidean distances of its points.

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_axial_distances(*idx_start=0*, *idx_end=-1*, *return_max=False*)

Return the displacements or its max value of a subset of branch points from its axis vector. :param idx_start = 0: index of the first point of the subset. :param idx_end = -1: index of the last point of the subset. :param return_max = False: if True the function only returns the max value of displacements, otherwise the entire array.

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_label_mask(*labels*)

Return a mask for the specified labels

Parameters

labels (*List[str] | numpy.ndarray[str]*) – The labels to check for.

Returns

A boolean mask that selects out the points that match the label.

Return type

List[numpy.ndarray]

get_points_labelled(*labels*)

Filter out all points with certain labels

Parameters

labels (*List[str] | numpy.ndarray[str]*) – The labels to check for.

Returns

All points with the labels.

Return type
List[numpy.ndarray]

`insert_branch`(*branch*, *index*)

Split this branch and insert the given `branch` at the specified `index`.

Parameters

- **branch** (*Branch*) – Branch to be attached
- **index** – Index or coordinates of the cutpoint; if coordinates are given, the closest point to the coordinates is used.

Type
Union[numpy.ndarray, int]

`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`(*labels*, *points=None*)

Add labels to the branch.

Parameters

- **labels** (*List[str]*) – Label(s) for the branch
- **points** – An integer or boolean mask to select the points to label.

property labels

Return the labels of the points on this branch. Labels are represented as a number that is associated to a set of labels. See *Labels* for more info.

property labelsets

Return the sets of labels associated to each numerical label.

list_labels()

Return a list of labels present on the branch.

property max_displacement

Return the max displacement of the branch points from its axis vector.

property path_length

Return the sum of the euclidean distances between the points on the branch.

property point_vectors

Return the individual vectors between consecutive points on this branch.

property points

Return the spatial coordinates of the points on this branch.

property radii

Return the radii of the points on this branch.

root_rotate(*rot*, *downstream_of=0*)

Rotate the subtree emanating from each root around the start of that root If downstream_of is provided, will rotate points starting from the index provided (only for subtrees with a single root).

Parameters

- **rot** (*scipy.spatial.transform.Rotation*) – Scipy rotation to apply to the subtree.
- **downstream_of** – index of the point in the subtree from which the rotation should be applied. This feature works only when the subtree has only one root branch.

Returns
rotated Morphology

Return type
bsb.morphologies.SubTree

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

Point rotation

Parameters

- **rot** – Scipy rotation
- **center** (*numpy.ndarray*) – rotation offset point.

Type
Union[scipy.spatial.transform.Rotation, List[float,float,float]]

property segments

Return the start and end points of vectors between consecutive points on this branch.

simplify(*epsilon*, *idx_start=0*, *idx_end=-1*)

Apply Ramer–Douglas–Peucker algorithm to all points or a subset of points of the branch. :param epsilon: Epsilon to be used in the algorithm. :param idx_start = 0: Index of the first element of the subset of points to be reduced. :param epsilon = -1: Index of the last element of the subset of points to be reduced.

simplify_branches(*epsilon*)

Apply Ramer–Douglas–Peucker algorithm to all points of all branches of the SubTree. :param epsilon: Epsilon to be used in the algorithm.

property size

Returns the amount of points on this branch

Returns
Number of points on the branch.

Return type
int

property start

Return the spatial coordinates of the starting point of this branch.

translate(*point*)

Translate the subtree by a 3D vector.

Parameters
point (*numpy.ndarray*) – 3D vector to translate the subtree.

Returns
the translated subtree

Return type
bsb.morphologies.SubTree

property vector

Return the vector of the axis connecting the start and terminal points.

property versor

Return the normalized vector of the axis connecting the start and terminal points.

voxelize(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids.

Return type
bsb.voxels.VoxelSet

walk()

Iterate over the points in the branch.

class bsb.morphologies.**Morphology**(*roots*, *meta=None*, *shared_buffers=None*, *sanitize=False*)

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.

property adjacency_dictionary

Return a dictonary associating to each key (branch index) a list of adjacent branch indices

as_filtered(*labels=None*)

Return a filtered copy of the morphology that includes only points that match the current label filter, or the specified labels.

copy()

Copy the morphology.

get_label_mask(*labels*)

Get a mask corresponding to all the points labelled with 1 or more of the given labels

property labelsets

Return the sets of labels associated to each numerical label.

list_labels()

Return a list of labels present on the morphology.

set_label_filter(*labels*)

Set a label filter, so that *as_filtered* returns copies filtered by these labels.

to_graph_array()

Create a SWC-like numpy array from a Morphology.

> **Warning:** Custom SWC tags (above 3) won't work and throw an error

Returns
a numpy array with columns storing the standard SWC attributes

Return type
numpy.ndarray

to_swc(*file*)

Create a SWC file from a Morphology. :param file: path to write to

class bsb.morphologies.**MorphologySet**(*loaders*, *m_indices=None*, */*, *labels=None*)

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 branch_adjacency

Return a dictonary containing mapping the id of the branch to its children.

property branches

Return a depth-first flattened array of all branches.

cached_voxelize(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids and cache the result.

Return type
bsb.voxels.VoxelSet

center()

Center the morphology on the origin

close_gaps()

Close any head-to-tail gaps between parent and child branches.

collapse(*on=None*)

Collapse all the roots of the morphology or subtree onto a single point.

Parameters
on (*int*) – Index of the root to collapse on. Collapses onto the origin by default.

flatten()

Return the flattened points of the morphology or subtree.

Return type
numpy.ndarray

flatten_labels()

Return the flattened labels of the morphology or subtree.

Return type
numpy.ndarray

flatten_properties()

Return the flattened properties of the morphology or subtree.

Return type
numpy.ndarray

flatten_radii()

Return the flattened radii of the morphology or subtree.

Return type
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

label(*labels*, *points=None*)

Add labels to the morphology or subtree.

Parameters

- **labels** (*list[str]*) – Labels to add to the subtree.
- **points** (*numpy.ndarray*) – Optional boolean or integer mask for the points to be labelled.

property path_length

Return the total path length as the sum of the euclidian distances between consecutive points.

root_rotate(*rot*, *downstream_of=0*)

Rotate the subtree emanating from each root around the start of that root If downstream_of is provided, will rotate points starting from the index provided (only for subtrees with a single root).

Parameters

- **rot** (*scipy.spatial.transform.Rotation*) – Scipy rotation to apply to the subtree.
- **downstream_of** – index of the point in the subtree from which the rotation should be applied. This feature works only when the subtree has only one root branch.

Returns
rotated Morphology

Return type
bsb.morphologies.SubTree

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

Point rotation

Parameters

- **rot** – Scipy rotation
- **center** (*numpy.ndarray*) – rotation offset point.

Type
Union[scipy.spatial.transform.Rotation, List[float,float,float]]

simplify_branches(*epsilon*)

Apply Ramer–Douglas–Peucker algorithm to all points of all branches of the SubTree. :param epsilon: Epsilon to be used in the algorithm.

translate(*point*)

Translate the subtree by a 3D vector.

Parameters
point (*numpy.ndarray*) – 3D vector to translate the subtree.

Returns
the translated subtree

Return type
bsb.morphologies.SubTree

voxelize(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids.

Return type
bsb.voxels.VoxelSet

bsb.morphologies.**branch_iter**(*branch*)

Iterate over a branch and all of its children depth first.

CHAPTER

EIGHTEEN

MORPHOLOGY PARSERS

18.1 BsbParser

This is the default parser. It only parses SWC files, but it allows you to create as many different SWC tags as you want, and can assign labels based on these tags. This allows you to create a rich annotation system on top of the SWC format to denote dendritic spines, axonal myelination with saltatory conduction, or any other morphological feature.

PYTHON

```
from bsb.morphologies.parsers.parser import BsbParser

BsbParser(
  cls='builtins.str',
  branch_cls='builtins.str',
  parser='bsb',
  tags={},
  skip_boundary_labels=[],
)
```

JSON

```
{
    "cls": "builtins.str",
    "branch_cls": "builtins.str",
    "parser": "bsb",
    "tags": {},
    "skip_boundary_labels": []
}
```

YAML

```yaml
branch_cls: builtins.str
cls: builtins.str
parser: bsb
skip_boundary_labels: []
tags: {}
```

The default SWC tags are 1 for `soma`, 2 for `axon`, and 3 for `dendrites`. You can add/overwrite tags by setting the *tags* attribute:

```
{
  "parser": "bsb",
  "tags": {
    4: ["dendrites", "apical_dendrites"],
    5: ["dendrites", "basal_dendrites"],
    6: ["axon", "axon_initial_segment"],
    7: ["axon", "axon_hillock"],
    8: ["axon", "myelin"],
    9: ["axon", "myelin", "node_of_ranvier"]
  }
}
```

The soma is usually approximated as a stack of cylinders, this means that

18.2 MorphIOParser

PYTHON

```python
from bsb.morphologies.parsers.parser import MorphIOParser

MorphIOParser(
  cls='builtins.str',
  branch_cls='builtins.str',
  parser='bsb',
  flags=[],
)
```

JSON

```
{
    "cls": "builtins.str",
    "branch_cls": "builtins.str",
    "parser": "bsb",
    "flags": []
}
```

YAML

```yaml
branch_cls: builtins.str
cls: builtins.str
flags: []
parser: bsb
```

CHAPTER

NINETEEN

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.

To access an MR, a *Storage* object is required:

```
from bsb import Storage

store = Storage("hdf5", "morphologies.hdf5")
mr = store.morphologies
print(mr.all())
```

Similarly, the built-in MR of a network is accessible as `network.morphologies`:

```
from bsb import from_storage

network = from_hdf("my_existing_model.hdf5")
mr = network.morphologies
```

You can use the *save()* method to store *Morphologies*. If you don't immediately need the whole morphology, you can *preload()* it, otherwise you can load the entire thing with *load()*.

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

abstract all()

Fetch all of the stored morphologies.

Returns
List of the stored morphologies.

Return type
List[StoredMorphology]

abstract get_all_meta()

Get the metadata of all stored morphologies. :returns: Metadata dictionary :rtype: dict

abstract get_meta(*name*)

Get the metadata of a stored morphology.

Parameters
name (*str*) – Key of the stored morphology.

Returns
Metadata dictionary

Return type
dict

abstract has(*name*)

Check whether a morphology under the given name exists

Parameters
name (*str*) – Key of the stored morphology.

Returns
Whether the key exists in the repo.

Return type
bool

list()

List all the names of the morphologies in the repository.

abstract load(*name*)

Load a stored morphology as a constructed morphology object.

Parameters
name (*str*) – Key of the stored morphology.

Returns
A morphology

Return type
Morphology

abstract preload(*name*)

Load a stored morphology as a morphology loader.

Parameters
name (*str*) – Key of the stored morphology.

Returns
The stored morphology

Return type
StoredMorphology

abstract save(*name*, *morphology*, *overwrite=False*)

Store a morphology

Parameters

- **name** (*str*) – Key to store the morphology under.
- **morphology** (bsb.morphologies.Morphology) – Morphology to store
- **overwrite** (*bool*) – Overwrite any stored morphology that already exists under that name

Returns
The stored morphology

Return type
StoredMorphology

abstract select(**selectors*)

Select stored morphologies.

Parameters
selectors (*List[*bsb.morphologies.selector.MorphologySelector*]*) – Any number of morphology selectors.

Returns
All stored morphologies that match at least one selector.

Return type
List[StoredMorphology]

abstract set_all_meta(*all_meta*)

Set the metadata of all stored morphologies. :param all_meta: Metadata dictionary. :type all_meta: dict

abstract update_all_meta(*meta*)

Update the metadata of stored morphologies with the provided key values

Parameters
meta (*str*) – Metadata dictionary.

CHAPTER

TWENTY

MORPHOLOGYSET

20.1 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 import from_storage

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

CHAPTER

TWENTYONE

LIST OF PLACEMENT STRATEGIES

21.1 RandomPlacement

Class: *bsb.placement.RandomPlacement*

21.2 ParallelArrayPlacement

Class: *bsb.placement.ParallelArrayPlacement*

21.3 FixedPositions

Class: *bsb.placement.FixedPositions*

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

- `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]]
        }
    },
  }
}
```

CHAPTER

TWENTYTWO

PLACEMENT SETS

PlacementSets are constructed from the *Storage* and can be used to retrieve the positions, morphologies, rotations and additional datasets.

Note: Loading datasets from storage is an expensive operation. Store a local reference to the data you retrieve instead of making multiple calls.

22.1 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 import from_storage

network = from_storage("my_network.hdf5")
ps = network.get_placement_set("my_cell")
# Alternatives to obtain the same placement set:
ps = network.get_placement_set(network.cell_types.my_cell)
ps = network.cell_types.my_cell.get_placement_set()
ps = network.storage.get_placement_set(network.cell_types.my_cell)
```

22.2 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 so will the n-th rotation.

22.3 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)
```

22.4 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**.

22.5 Rotations

The positions of the cells can be retrieved using the *load_rotations()* method.

22.6 Additional datasets

Not implemented yet.

CHAPTER

TWENTYTHREE

DEFINING CONNECTIONS

23.1 Adding a connection type

Connections are defined in the configuration under the `connectivity` block:

```
{
  "connectivity": {
    "type_A_to_type_B": {
      "strategy": "bsb.connectivity.VoxelIntersection",
      "presynaptic": {
        "cell_types": ["type_A"]
      },
      "postsynaptic": {
          "cell_types": ["type_B"]
      }
    }
  }
}
```

- *strategy*: Which *ConnectionStrategy* to load.
- *pre*/*post*: The pre/post-synaptic *hemitypes*:
  - *cell_types*: A list of cell types.
  - *labels*: (optional) a list of labels to filter the cells by
  - *morphology_labels*: (optional) a list of labels that filter which pieces of the morphology to consider when forming connections (such as `axon`, `dendrites`, or any other label you've created)

What each connection type does depends entirely on the selec

The framework will load the specified *strategy*, and will ask the strategy to determine the regions of interest, and will queue up one parallel job per region of interest. In each parallel job, the data generated during the placement step is used to determine presynaptic to postsynaptic connection locations.

23.2 Targetting subpopulations using cell labels

Each hemitype (*presynaptic* and *postsynaptic*) accepts an additional list of labels to filter the cell populations by. This can be used to connect subpopulations of cells that are labelled with any of the given labels:

```
{
  "components": ["my_module.py"],
  "connectivity": {
    "type_A_to_type_B": {
      "class": "my_module.ConnectBetween",
      "min": 10,
      "max": 15.5,
      "presynaptic": {
        "cell_types": ["type_A"],
        "labels": ["subgroup1", "example2"]
      },
      "postsynaptic": {
        "cell_types": ["type_B"]
      }
    }
  }
}
```

This snippet would connect only the cells of `type_A` that are labelled with either `subgroup1` or `example2`, to all of the cells of `type_B`, within 10 to 15.5 micrometer distance of each other.

23.3 Specifying subcellular regions using morphology labels

You can also specify which regions on a morphology you're interested in connecting. By default axodendritic contacts are enabled, but by specifying different *morphology_labels* you can alter this behavior. This example lets you form dendrodendritic contacts:

```
{
  "components": ["my_module.py"],
  "connectivity": {
    "type_A_to_type_B": {
      "class": "my_module.ConnectBetween",
      "min": 10,
      "max": 15.5,
      "presynaptic": {
        "cell_types": ["type_A"],
        "morphology_labels": ["dendrites"]
      },
      "postsynaptic": {
        "cell_types": ["type_B"],
        "morphology_labels": ["dendrites"]
      }
    }
  }
}
```

In general this works with any label that is present on the morphology. You could process your morphologies to add as many labels as you want, and then create different connectivity targets.

CHAPTER

TWENTYFOUR

WRITING A COMPONENT

New to components? Write your first one with *our guide*

You can create custom connectivity patterns by creating a Python file in your project root (e.g. `my_module.py`) with inside a class inheriting from *`ConnectionStrategy`*.

First we'll discuss the parts of the interface to implement, followed by an example, some notes, and use cases.

24.1 Interface

24.1.1 `connect()`

- `pre_set`/`post_set`: The pre/post-synaptic placement sets you used to perform the calculations.
- `src_locs`/`dest_locs`:
  - **A matrix with 3 columns with on each row the cell id, branch id, and** point id.
  - **Each row of the `src_locs` matrix will be connected to the same row in the** `dest_locs` matrix
- **`tag`**
  [a tag describing the connection (optional, defaults to the strategy name, or] *f"{name}_{pre}_to_{post}"* when multiple cell types are combined). Use this when you wish to create multiple distinct sets between the same cell types.

For example, if `src_locs` and `dest_locs` are the following matrices:

Table 1: src_locs

Index of the cell in pre_pos array	Index of the branch at which the connection starts	Index of the point on the branch at which the connection starts.
2	0	6
10	0	2

Table 2: dest_locs

Index of the cell in post_pos array	Index of the branch at which the connecion ends.	Index of the point on the branch at which the connection ends.
5	1	3
7	1	4

then two connections are formed:

- **The first connection is formed between the presynaptic cell whose index in `pre_pos`** is `2` and the postsynaptic cell whose index in `post_pos` is `10`.

Furthermore, the connection begins at the point with id `6` on the branch whose id is `0` on the presynaptic cell and ends on the points with id `3` on the branch whose id is `1` on the postsynaptic cell.

- **The second connection is formed between the presynaptic cell whose index in `pre_pos`** is `10` and the postsynaptic cell whose index in `post_pos` is `7`. Furthermore, the connection begins at the point with id `3` on the branch whose id is `0` on the presynaptic cell and ends on the points with id `4` on the branch whose id is `1` on the postsynaptic cell.

Note: If the exact location of a synaptic connection is not needed, then in both `src_locs` and `dest_locs` the indices of the branches and of the point on the branch can be set to `-1`.

24.1.2 `get_region_of_interest()`

This is an optional part of the interface. Using a region of interest (RoI) can speed up algorithms when it is possible to know for a given presynaptic chunk, which postsynaptic chunks might contain useful cell candidates.

Chunks are identified by a set of coordinates on a regular grid. E.g., for a network with chunk size (100, 100, 100), the chunk (3, -2, 1) is the rhomboid region between its least dominant corner at (300, -200, 100), and its most dominant corner at (200, -100, 0).

`get_region_of_interest(chunk)` receives the presynaptic chunk and should return a list of postsynaptic chunks.

24.2 Example

The example connects cells that are near each other, between a *min* and *max* distance:

```
from bsb import ConnectionStrategy, ConfigurationError, 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 connect(self, pre, post):
    # The `connect` function is responsible for deciding which cells get connected.
    # Use each hemitype's `.placement` to get a dictionary of `PlacementSet`s to connect

    # Cross-combine each presynaptic placement set ...
    for presyn_data in pre.placement:
      from_pos = presyn_data.load_positions()
      # ... with each postsynaptic placement set
      for postsyn_data in post.placement:
        to_pos = postsyn_data.load_positions()
        # Calculate the NxM pairwise distances between the cells
```

(continues on next page)

(continued from previous page)

```
        pairw_dist = dist.cdist(from_pos, to_pos)
        # Find those that match the distance criteria
        m_pre, m_post = np.nonzero((pairw_dist <= max) & (pairw_dist >= min))
        # Construct the Kx3 connection matrices
        pre_locs = np.full((len(m_pre), 3), -1)
        post_locs = np.full((len(m_pre), 3), -1)
        # The first columns are the cell ids, the other columns are padded with -1
        # to ignore subcellular precision and form point neuron connections.
        pre_locs[:, 0] = m_pre
        post_locs[:, 0] = m_post
        # Call `self.connect_cells` to store the connections you found
        self.connect_cells(presyn_data, postsyn_data, pre_locs, post_locs)

  # Optional, you can leave this off to focus on `connect` first.
  def get_region_of_interest(self, chunk):
    # Find all postsynaptic chunks that are within the search radius away from us.
    return [
      c
      for c in self.get_all_post_chunks()
      if dist.euclidean(c.ldc, chunk.ldc) < self.max + chunk.dimensions
    ]

  # Optional, you can add extra checks and preparation of your component here
  def __init__(self, **kwargs):
    # Check if the configured max and min distance values make sense.
    if self.max < self.min:
      raise ConfigurationError("Max distance should be larger than min distance.")
```

And an example configuration using this strategy:

```
{
  "components": ["my_module.py"],
  "connectivity": {
    "type_A_to_type_B": {
      "class": "my_module.ConnectBetween",
      "min": 10,
      "max": 15.5,
      "presynaptic": {
        "cell_types": ["type_A"]
      },
      "postsynaptic": {
        "cell_types": ["type_B"]
      }
    }
  }
}
```

24.2.1 Notes

Setting up the class

We need to inherit from *ConnectionStrategy* to create a connection component and decorate our class with the `config.node` decorator to integrate it with the configuration system. For specifics on configuration, see *Nodes*.

Accessing configuration values during `connect`

Any `config.attr` or similar attributes that you define on the class will be populated with data from the network configuration, and will be available on `self` in the methods of the component.

In this example *min* is an optional float that defaults to 0, and *max* is a required float.

Accessing placement data during `connect`

The `connect` function is handed the placement information as the `pre` and `post` parameters. The `.placement` attribute contains a dictionary with as keys the *cell_types.CellType* and as value the *PlacementSets*.

Note: The placement sets in the parameters are scoped to the data of the parallel job that is being executed. If you want to remove this scope and access to the global data, you can create a fresh placement set from the cell type with `cell_type.get_placement_set()`.

Creating connections

Connections are stored in a presynaptic and postsynaptic matrix. Each matrix contains 3 columns: the cell id, branch id, and point id. If your cells have no morphologies, use -1 as a filler for the branch and point ids.

Call `self.scaffold.connect_cells(from_type, to_type, from_locs, to_locs)` to connect the cells. If you are creating multiple different connections between the same pair of cell types, you can pass an optional `tag` keyword argument to give them a unique name and separate them.

Use regions of interest

Using a region of interest (RoI) can speed up algorithms when it is possible to know, when given a presynaptic chunk, which postsynaptic chunks might contain useful cell candidates.

Chunks are identified by a set of coordinates on a regular grid. E.g., for a network with chunk size (100, 100, 100), the chunk (3, -2, 1) is the rhomboid region between its least dominant corner at (300, -200, 100), and its most dominant corner at (200, -100, 0).

Using the same example, for every presynaptic chunk, we know that we will only form connections with cells less than `max` distance away, so why check cells in chunks more than `max` distance away?

If you implement `get_region_of_interest(chunk)`, you can return the list of chunks that should be loaded for the parallel job that processes that `chunk`:

```
def get_region_of_interest(self, chunk):
  return [
    c
    for c in self.get_all_post_chunks()
```

(continues on next page)

(continued from previous page)

```
    if dist.euclidean(c.ldc, chunk.ldc) < self.max + chunk.dimensions
  ]
```

24.3 Connecting point-like cells

Suppose we want to connect Golgi cells and granule cells, without storing information about the exact positions of the synapses (we may want to consider cells as point-like objects, as in NEST). We want to write a class called `ConnectomeGolgiGranule` that connects a Golgi cell to a granule cell if their distance is less than 100 micrometers (see the configuration block above).

First we define the class `ConnectomeGolgiGlomerulus` and we specify that we require to be configured with a *radius* and *divergence* attribute.

```
@config.node
class ConnectomeGolgiGlomerulus(ConnectionStrategy):
    # Read vars from the configuration file
    radius = config.attr(type=int, required=True)
    divergence = config.attr(type=int, required=True)
```

Now we need to write the `get_region_of_interest` method. For a given chunk we want all the neighbouring chunks in which we can find the presynaptic cells at less than 50 micrometers. Such cells are contained for sure in the chunks which are less than 50 micrometers away from the current chunk.

```
def get_region_of_interest(self, chunk):
  # We get the ConnectivitySet of golgi_to_granule
  cs = self.network.get_connectivity_set(tag="golgi_to_granule")
  # We get the coordinates of all the chunks
  chunks = ct.get_placement_set().get_all_chunks()
  # We define an empty list in which we shall add the chunks of interest
  selected_chunks = []
  # We look for chunks which are less than radius away from the current one
  for c in chunks:
    dist = np.sqrt(
      np.power((chunk[0] - c[0]) * chunk.dimensions[0], 2)
        + np.power((chunk[1]  - c[1]) * chunk.dimensions[1], 2)
        + np.power((chunk[2]  - c[2]) * chunk.dimensions[2], 2)
    )
    # We select only the chunks satisfying the condition
    if (dist < self.radius):
        selected_chunks.append(Chunk([c[0], c[1], c[2]], chunk.dimensions))
  return selected_chunks
```

Now we're ready to write the `connect` method:

```
def connect(self, pre, post):
  # This strategy connects every combination pair of the configured presynaptic to␣
↪postsynaptic cell types.
  # We will tackle each pair's connectivity inside of our own `_connect_type` helper␣
↪method.
  for pre_ps in pre.placement:
      for post_ps in post.placement:
```

(continues on next page)

(continued from previous page)

```
        # The hemitype collection's `placement` is a dictionary mapping each cell type␣
↪to a placement set with all
        # cells being processed in this parallel job. So call our own `_connect_type`␣
↪method with each pre-post combination
        self._connect_type(pre_ps, post_ps)

  def _connect_type(self, pre_ps, post_ps):
    # This is the inner function that calculates the connectivity matrix for a pre-post␣
↪cell type pair
    # We start by loading the cell position matrices (Nx3)
    golgi_pos = pre_ps.load_positions()
    granule_pos = post_ps.load_positions()
    n_glomeruli = len(glomeruli_pos)
    n_golgi = len(golgi_pos)
    n_conn = n_glomeruli * n_golgi
    # For the sake of speed we define two arrays pre_locs and post_locs of length n_conn
    # (the maximum number of connections which can be made) to store the connections␣
↪information,
    # even if we will not use all the entries of arrays.
    # We keep track of how many entries we actually employ, namely how many connection
    # we made, using the variable ptr. For example if we formed 4 connections the useful
    # data lie in the first 4 elements
    pre_locs = np.full((n_conn, 3), -1, dtype=int)
    post_locs = np.full((n_conn, 3), -1, dtype=int)
    ptr = 0
    # We select the cells to connect according to our connection rule.
    for i, golgi in enumerate(golgi_pos):
      # We compute the distance between the current Golgi cell and all the granule cells␣
↪in the region of interest.
      dist = np.sqrt(
                  np.power(golgi[0] - granule_pos[0], 2)
                  + np.power(golgi[1] - granule_pos[1], 2)
                  + np.power(golgi[2] - granule_pos[2], 2)
              )
      # We select all the granule cells which are less than 100 micrometers away up to␣
↪the divergence value.
      # For the sake of simplicity in this example we assume to find at least 40␣
↪candidates satisfying the condition.
      granule_close_enough = dist < self.radius

      # We find the indices of the 40 closest granule cells
      to_connect_ids = np.argsort(granule_close_enough)[0:self.divergence]

      # Since we are interested in connecting point-like cells, we do not need to store
      # info about the precise position on the dendrites or axons;
      # It is enough to store which presynaptic cell is connected to
      # certain postsynaptic cells, namely the first entry of both `pre_set` and `post_
↪set`.

      # The index of the presynaptic cell in the `golgi_pos` array is `i`
      pre_set[ptr:ptr+self.divergence,0] = i
      # We store in post_set the indices of the postsynaptic cells we selected before.
```

(continues on next page)

(continued from previous page)

```
      post_set[ptr:ptr+self.divergence,0] = to_connect_ids
      ptr += to_be_connected

    # Now we connect the cells according to the information stored in `src_locs` and␣
↪`dest_locs`
    # calling the `connect_cells` method.
    self.connect_cells(pre_set, post_set, src_locs, dest_locs)
```

24.4 Connections between a detailed cell and a point-like cell

If we have a detailed morphology of the pre- or postsynaptic cells we can specify where to form the connection. Suppose we want to connect Golgi cells to glomeruli specifying the position of the connection on the Golgi cell axon. In this example we form a connection on the closest point to a glomerulus. First, we need to specify the type of neurites that we want to consider on the morphologies when forming synapses. We can do this in the configuration file, using the:guilabel:*morphology_labels* attribute on the `connectivity.*.postsynaptic` (or `presynaptic`) node:

```
"golgi_to_granule": {
      "strategy": "cerebellum.connectome.golgi_granule.ConnectomeGolgiGranule",
      "radius": 100,
      "convergence": 40,
      "presynaptic": {
        "cell_types": ["glomerulus"]
      },
      "postsynaptic": {
        "cell_types": ["golgi_cell"],
        "morphology_labels" : ["basal_dendrites"]
      }
    }
```

The *get_region_of_interest()* is analogous to the previous example, so we focus only on the *connect()* method.

```
def connect(self, pre, post):
  for pre_ps in pre.placement:
      for post_ps in post.placement:
          self._connect_type(pre_ps, post_ps)

  def _connect_type(self, pre_ps, post_ps):
    # We store the positions of the pre and post synaptic cells.
    golgi_pos = pre_ps.load_positions()
    glomeruli_pos = post_ps.load_positions()
    n_glomeruli = len(glomeruli_pos)
    n_golgi = len(golgi_pos)
    max_conn = n_glomeruli * n_golgi
    # We define two arrays of length `max_conn ` to store the connections,
    # even if we will not use all the entries of arrays, for the sake of speed.
    pre_locs = np.full((max_conn , 3), -1, dtype=int)
    post_locs = np.full((max_conn , 3), -1, dtype=int)
    # `ptr` keeps track of how many connections we've made so far.
    ptr = 0
```

(continues on next page)

(continued from previous page)

```
    # Cache morphologies and generate the morphologies iterator.
    morpho_set = post_ps.load_morphologies()
    golgi_morphos = morpho_set.iter_morphologies(cache=True, hard_cache=True)

    # Loop through all the Golgi cells
    for i, golgi, morpho in zip(itertools.count(), golgi_pos, golgi_morphos):

        # We compute the distance between the current Golgi cell and all the glomeruli,
        # then select the good ones.
        dist = np.sqrt(
            np.power(golgi[0] - glomeruli_pos[:, 0], 2)
            + np.power(golgi[1] - glomeruli_pos[:, 1], 2)
            + np.power(golgi[2] - glomeruli_pos[:, 2], 2)
        )

        to_connect_bool = dist < self.radius
        to_connect_idx = np.nonzero(to_connect_bool)[0]
        connected_gloms = len(to_connect_idx)

        # We assign the indices of the Golgi cell and the granule cells to connect
        pre_locs[ptr : (ptr + connected_gloms), 0] = to_connect_idx
        post_locs[ptr : (ptr + connected_gloms), 0] = i

        # Get the branches corresponding to basal dendrites.
        # `morpho` contains only the branches tagged as specified
        # in the configuration file.
        basal_dendrides_branches = morpho.get_branches()

        # Get the starting branch id of the denridic branches
        first_dendride_id = morpho.branches.index(basal_dendrides_branches[0])

        # Find terminal points on branches
        terminal_ids = np.full(len(basal_dendrides_branches), 0, dtype=int)
        for i,b in enumerate(basal_dendrides_branches):
            if b.is_terminal:
                terminal_ids[i] = 1
        terminal_branches_ids = np.nonzero(terminal_ids)[0]

        # Keep only terminal branches
        basal_dendrides_branches = np.take(basal_dendrides_branches, terminal_branches_
↪ids, axis=0)
        terminal_branches_ids = terminal_branches_ids + first_dendride_id

        # Find the point-on-branch ids of the tips
        tips_coordinates = np.full((len(basal_dendrides_branches),3), 0, dtype=float)
        for i,branch in enumerate(basal_dendrides_branches):
            tips_coordinates[i] = branch.points[-1]

        # Choose randomly the branch where the synapse is made
        # favouring the branches closer to the glomerulus.
        rolls = exp_dist.rvs(size=len(basal_dendrides_branches))
```

(continues on next page)

(continued from previous page)

```
        # Compute the distance between terminal points of basal dendrites
        # and the soma of the avaiable glomeruli
        for id_g,glom_p in enumerate(glomeruli_pos):
            pts_dist = np.sqrt(np.power(tips_coordinates[:,0] + golgi[0] - glom_p[0], 2)
                    + np.power(tips_coordinates[:,1] + golgi[1] - glom_p[1], 2)
                    + np.power(tips_coordinates[:,2] + golgi[2] - glom_p[2], 2)
                )

            sorted_pts_ids = np.argsort(pts_dist)
            # Pick the point in which we form a synapse according to a exponential␣
↪distribution mapped
            # through the distance indices: high chance to pick closeby points.
            pt_idx = sorted_pts_ids[int(len(basal_dendrides_branches)*rolls[np.random.
↪randint(0,len(rolls))])]

            # The id of the branch is the id of the terminal_branches plus the id of the␣
↪first dendritic branch
            post_locs[ptr+id_g,1] = terminal_branches_ids[pt_idx]
            # We connect the tip of the branch
            post_locs[ptr+id_g,2] = len(basal_dendrides_branches[pt_idx].points)-1
        ptr += connected_gloms

    # Now we connect the cells
    self.connect_cells(pre_ps, post_ps, pre_locs[:ptr], post_locs[:ptr])
```

CHAPTER

TWENTYFIVE

LIST OF STRATEGIES

25.1 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.

CHAPTER

TWENTYSIX

SIMULATING NETWORKS

Simulations can be run through the CLI:

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

or through the `bsb` library for more control. When using the CLI, the framework sets up a "hands off" simulation workflow:

- 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

When you use the library, you can set up more complex workflows, such as parameter sweeps:

```
# A module to read HDF5 data

# A module to run NEURON simulations in isolation
import nrnsub

from bsb import from_storage


# This decorator runs each call to the function in isolation
# on a separate process. Does not work with MPI.
@nrnsub.isolate
def sweep(param):
    # Open the network file
    network = from_storage("my_network.hdf5")

    # Here you can set whatever simulation parameter you want.
    network.simulations.my_sim.devices.my_stim.rate = param

    # Run the simulation
    results = network.run_simulation("my_sim")
    # Tjese are the recorded spiketrains and signals
    print(results.spiketrains)
    print(results.analogsignals)
```

(continues on next page)

(continued from previous page)

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

Parallel simulations

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

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

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

26.1 Configuration

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 is covered per simulator below.

26.2 Arbor

26.2.1 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.

26.2.2 Connection models

todo: doc

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

26.2.3 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

26.3 NEST

Additional root attributes:

- `modules`: list of NEST extension modules to be installed.

```
{
  "simulations": {
    "first_simulation": {
      "simulator": "nest",
      "duration": 1000,
      "resolution": 1.0,
      "modules": ["cerebmodule"],

      "cell_models": {

      },
      "connection_models": {

      },
      "devices": {

      }
    },
    "second_simulation": {

    }
  }
}
```

26.3.1 Cell models

In the `cell_models` block, you specify the simulator representation for each cell type. Each key in the block can have the following attributes:

- `model`: NEST neuron model, See the available models in the NEST documentation
- `constants`: 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]

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": {
      "constants": {
        "t_ref": 2.0,
        "C_m": 145.0,
        "V_th": -55.0,
        "V_reset": -75.0,
        "E_L": -62.0
      }
    }
  }
}
```

26.3.2 Connection models

26.3.3 Devices

26.4 NEURON

26.4.1 Cell models

By default the NEURON adapter uses an `ArborizedCellModel`, which loads external `arborize` definition to instantiate cells.

```
{
  "cell_models": {
    "cell_type_A": {
      "model": "dbbs_models.GranuleCell"
```

(continues on next page)

(continued from previous page)

```
    },
    "cell_type_B": {
      "model": "dbbs_models.PurkinjeCell"
    }
  }
}
```

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.

26.4.2 Connection models

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"]
    }
  }
}
```

26.4.3 Devices

Devices send input, or record output.

Here we'll place a spike generator and a voltage clamp:

```
{
  "devices": {
    "stimulus": {
      "device": "spike_generator",
      "targetting": {
          "strategy": "cell_model",
          "cell_models": ["cell_type_A"]
      },
      "synapses": ["AMPA"],
      "start": 500,
      "number": 10,
      "interval": 10,
      "noise": true
    },
    "voltage_clamp": {
      "device": "voltage_clamp",
      "targetting": {
          "strategy": "cell_model",
          "cell_models": ["cell_type_A"]
          "count": 1,
      },
      "location": {
          "strategy": "soma"
```

(continues on next page)

(continued from previous page)

```
      },
      "parameters": {
        "delay": 0,
        "duration": 1000,
        "after": 0,
        "voltage": -63
      }
    }
  }
}
```

CHAPTER

TWENTYSEVEN

BSB

27.1 bsb package

27.1.1 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: *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.**BsbCommand**

Bases: object

add_to_parser()

handler(*context*)

class bsb.cli.commands.**RootCommand**

Bases: *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

Module contents

class bsb.config.parsers.**ConfigurationParser**

bsb.config.parsers.**get_configuration_parser**(*parser*, ***kwargs*)

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.

Submodules

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.

class bsb.config.refs.**Reference**

Bases: object

up(*here*, *to=None*)

bsb.config.types module

class bsb.config.types.**PackageRequirement**

Bases: *TypeHandler*

class bsb.config.types.**TypeHandler**

Bases: 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.

class bsb.config.types.**WeakInverter**(**args*, ***kwargs*)

Bases: object

store_value(*value*, *result*)

bsb.config.types.**any_**()

class bsb.config.types.**class_**(*module_path=None*)

Bases: *object_*

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

class bsb.config.types.**deg_to_radian**

Bases: *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

class bsb.config.types.**distribution**

Bases: *TypeHandler*

Type validator. Type casts the value or node to a distribution.

class bsb.config.types.**evaluation**

Bases: *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

class bsb.config.types.**function_**(*module_path=None*)

Bases: *object_*

Type validator. Attempts to import the value, absolute, or relative to the *module_path* entries, and verifies that it is callable.

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.**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.**key**()

Type handler for keys in configuration trees. Keys can be either int indices of a config list, or string keys of a config dict.

Returns
Type validator function

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

class bsb.config.types.**method**(*class_name*)

Bases: *function_*

class bsb.config.types.**method_shortcut**(*class_name*)

Bases: *method*, *function_*

bsb.config.types.**mut_excl**(**mutuals*, *required=True*, *max=1*, *shortform=False*)

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.
- **shortform** (*bool*) – Allow the short form alternative.

Returns
Requirement function

Return type
Callable

class bsb.config.types.**ndarray**

Bases: *TypeHandler*

Type validator numpy arrays.

Returns
Type validator function

Return type
Callable

bsb.config.types.**none**()

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

class bsb.config.types.**object_**(*module_path=None*)

Bases: *TypeHandler*

Type validator. Attempts to import the value, absolute, or relative to the *module_path* entries.

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.**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.**shortform**()

bsb.config.types.**str**(*strip=False*, *lower=False*, *upper=False*, *safe=True*)

Type validator. Attempts to cast the value to a str, optionally with some sanitation.

Parameters

- **strip** (*bool*) – Trim whitespaces
- **lower** (*bool*) – Convert value to lowercase
- **upper** (*bool*) – Convert value to uppercase
- **safe** (*bool*) – If False, checks that the type of value is string before cast.

Returns
Type validator function

Raises
TypeError when value can't be cast.

Return type
Callable

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

Module contents

bsb.config module

Contains the dynamic attribute system; Use `@bsb.config.root/node/dynamic/pluggable` to decorate your classes and add class attributes using `x = config.attr/dict/list/ref/reflist` to populate your classes with powerful attributes.

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

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

classmethod default(***kwargs*)

class bsb.config.**ConfigurationAttribute**(*type=None*, *default=None*, *call_default=None*, *required=False*, *key=False*, *unset=False*, *hint=<object object>*)

Bases: object

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

flag_dirty(*instance*)

flag_pristine(*instance*)

fset(*instance*, *value*)

get_default()

get_hint()

get_node_name(*instance*)

get_type()

is_dirty(*instance*)

is_node_type()

should_call_default()

tree(*instance*)

tree_of(*value*)

class bsb.config.**Distribution**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

distribution: str

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

draw(*n*)

get_node_name()

parameters: dict[str, Any]

scaffold: *Scaffold*

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 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). Use this to prevent mutable default values.
- **key** – If set, the key of the parent is stored on this attribute.

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.**compose_nodes**(**node_classes*)

Create a composite mixin class of the given classes. Inherit from the returned class to inherit from more than one node class.

bsb.config.**copy_configuration_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.**file**(***kwargs*)

Create a file dependency attribute.

bsb.config.**format_configuration_content**(*parser_name: str*, *config:* Configuration, ***kwargs*)

Convert a configuration object to a string using the given parser.

bsb.config.**get_config_attributes**(*cls*)

bsb.config.**get_config_path**()

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.**make_configuration_diagram**(*config*)

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.**parse_configuration_content**(*content*, *parser=None*, *path=None*, ***kwargs*)

bsb.config.**parse_configuration_file**(*file*, *parser=None*, *path=None*, ***kwargs*)

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*, */*, *type=None*, ***kwargs*)

Create a configuration property attribute. You may provide a value or a callable. Call *setter* on the return value as you would with a regular property.

bsb.config.**provide**(*value*)

Provide a value for a parent class' attribute. Can be a value or a callable, a readonly configuration property 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

Dev

class bsb.config._config.**NetworkNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

class bsb.config._attrs.**cfgdict**

Extension of the builtin dictionary to manipulate dicts of configuration nodes.

class bsb.config._attrs.**cfglist**(*iterable=()*, */*)

Extension of the builtin list to manipulate lists of configuration nodes.

bsb.topology package

Submodules

bsb.topology.partition module

Module for the Partition configuration nodes and its dependencies.

class bsb.topology.partition.**AllenStructure**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *NrrdVoxels*

Partition based on the Allen Institute for Brain Science mouse brain region ontology, later referred as Allen Mouse Brain Region Hierarchy (AMBRH)

classmethod find_structure(*id*)

Find an Allen structure by name, acronym or ID.

Parameters
id (*Union[str, int, float]*) – Query for the name, acronym or ID of the Allen structure.

Returns
Allen structure node of the Allen ontology tree.

Return type
dict

Raises
NodeNotFoundError

get_node_name()

classmethod get_structure_idset(*find*)

Return the set of IDs that make up the requested Allen structure.

Parameters
find (*Union[str, int]*) – Acronym or ID of the Allen structure.

Returns
Set of IDs

Return type
numpy.ndarray

classmethod get_structure_mask(*find*)

Returns the mask data delineated by the Allen structure.

Parameters
find (*Union[str, int]*) – Acronym, Name or ID of the Allen structure.

Returns
A boolean of the mask filtered based on the Allen structure.

Return type
Callable[numpy.ndarray]

classmethod get_structure_mask_condition(*find*)

Return a lambda that when applied to the mask data, returns a mask that delineates the Allen structure.

Parameters
find (*Union[str, int]*) – Acronym, Name or ID of the Allen structure.

Returns
Masking lambda

Return type
Callable[numpy.ndarray]

mask_only: bool

Flag to indicate no voxel data needs to be stored

mask_source: *NrrdDependencyNode*

Path to the NRRD file containing the volumetric annotation data of the Partition.

struct_id: int

Id of the region to filter within the annotation volume according to the AMBRH. If struct_id is set, then struct_name should not be set.

struct_name: str

Name or acronym of the region to filter within the annotation volume according to the AMBRH. If struct_name is set, then struct_id should not be set.

voxel_size: int

Size of each voxel.

class bsb.topology.partition.**Layer**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Rhomboid*

axis: Literal['x'] | Literal['y'] | Literal['z']

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

get_layout(*hint*)

Given a Layout as hint to begin from, create a Layout object that describes how this partition would like to be laid out.

Parameters
hint (bsb.topology._layout.Layout) – The layout space that this partition should place itself in.

Returns
The layout describing the space this partition takes up.

Return type
bsb.topology._layout.Layout

get_node_name()

stack_index: float

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

thickness: float

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

volume_scale: list[float]

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

class bsb.topology.partition.**NrrdVoxels**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Voxels*

Voxel partition whose voxelset is loaded from an NRRD file. By default it includes all the nonzero voxels in the file, but other masking conditions can be specified. Additionally, data can be associated to each voxel by inclusion of (multiple) source NRRD files.

get_mask()

Get the mask to apply on the sources' data of the partition.

Returns
A tuple of arrays, one for each dimension of the mask, containing the indices of the non-zero elements in that dimension.

get_node_name()

get_voxelset()

Creates a VoxelSet of the sources of the Partition that matches its mask.

Returns
VoxelSet of the Partition sources.

keys: list[str]

List of names to assign to each source of the Partition.

mask_only: bool

Flag to indicate no voxel data needs to be stored

mask_source: *NrrdDependencyNode*

Path to the NRRD file containing the volumetric annotation data of the Partition.

mask_value: int

Integer value to filter in mask_source (if it is set, otherwise sources/source) to create a mask of the voxel set(s) used as input.

source: *NrrdDependencyNode*

Path to the NRRD file containing volumetric data to associate with the partition. If source is set, then sources should not be set.

sources: *NrrdDependencyNode*

List of paths to NRRD files containing volumetric data to associate with the Partition. If sources is set, then source should not be set.

sparse: bool

Boolean flag to expect a sparse or dense mask. If the mask selects most voxels, use `dense`, otherwise use `sparse`.

strict: bool

Boolean flag to check the sources and the mask sizes. When the flag is True, sources and mask should have exactly the same sizes; otherwise, sources sizes should be greater than mask sizes.

voxel_size: int

Size of each voxel.

class bsb.topology.partition.**Partition**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: ABC

abstract chunk_to_voxels(*chunk*)

Voxelize the partition's occupation in this chunk. Required to fill the partition with cells by the placement module.

Parameters
chunk (bsb.storage._chunks.Chunk) – The chunk to calculate voxels for.

Returns
The set of voxels that together make up the shape of this partition in this chunk.

Return type
bsb.voxels.VoxelSet

property data

abstract get_layout(*hint*)

Given a Layout as hint to begin from, create a Layout object that describes how this partition would like to be laid out.

Parameters
hint (bsb.topology._layout.Layout) – The layout space that this partition should place itself in.

Returns
The layout describing the space this partition takes up.

Return type
bsb.topology._layout.Layout

get_node_name()

name: str

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

property region

abstract rotate(*rotation*)

Rotate the partition by the given rotation object.

Parameters
rotation (*scipy.spatial.transform.Rotation*) – Rotation object.

Raises
exceptions.LayoutError if the rotation needs to be rejected.

scaffold: *Scaffold*

abstract scale(*factors*)

Scale up/down the partition according to the given factors.

Parameters
factors (*numpy.ndarray*) – Scaling factors, XYZ.

Raises
exceptions.LayoutError if the scaling needs to be rejected.

abstract surface(*chunk=None*)

Calculate the surface of the partition in m^2.

Parameters
chunk (bsb.storage._chunks.Chunk) – If given, limit the surface of the partition inside of the chunk.

Returns
Surface of the partition (in the chunk)

Return type
float

abstract to_chunks(*chunk_size*)

Calculate all the chunks this partition occupies when cut into `chunk_sized` pieces.

Parameters
chunk_size (*numpy.ndarray*) – Size per chunk (in m). The slicing always starts at [0, 0, 0].

Returns
Chunks occupied by this partition

Return type
List[*bsb.storage._chunks.Chunk*]

abstract translate(*offset*)

Translate the partition by the given offset.

Parameters
offset (*numpy.ndarray*) – Offset, XYZ.

Raises
exceptions.LayoutError if the translation needs to be rejected.

type

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

abstract volume(*chunk=None*)

Calculate the volume of the partition in m^3.

Parameters
chunk (bsb.storage._chunks.Chunk) – If given, limit the volume of the partition inside of the chunk.

Returns
Volume of the partition (in the chunk)

Return type
float

class bsb.topology.partition.**Rhomboid**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Partition*

can_move: bool

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

can_rotate: bool

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

can_scale: bool

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

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 data.

dimensions: list[float]

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

get_dependencies()

Return other partitions or regions that need to be laid out before this.

get_layout(*hint*)

Given a Layout as hint to begin from, create a Layout object that describes how this partition would like to be laid out.

Parameters
hint (bsb.topology._layout.Layout) – The layout space that this partition should place itself in.

Returns
The layout describing the space this partition takes up.

Return type
bsb.topology._layout.Layout

get_node_name()

property ldc

property mdc

orientation: list[float]

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

origin: list[float]

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

rotate(*rot*)

Rotate the partition by the given rotation object.

Parameters
rotation (*scipy.spatial.transform.Rotation*) – Rotation object.

Raises
exceptions.LayoutError if the rotation needs to be rejected.

scale(*factors*)

Scale up/down the partition according to the given factors.

Parameters
factors (*numpy.ndarray*) – Scaling factors, XYZ.

Raises
exceptions.LayoutError if the scaling needs to be rejected.

surface(*chunk=None*)

Calculate the surface of the partition in m^2.

Parameters
chunk (bsb.storage._chunks.Chunk) – If given, limit the surface of the partition inside of the chunk.

Returns
Surface of the partition (in the chunk)

Return type
float

to_chunks(*chunk_size*)

Calculate all the chunks this partition occupies when cut into `chunk_sized` pieces.

Parameters
chunk_size (*numpy.ndarray*) – Size per chunk (in m). The slicing always starts at [0, 0, 0].

Returns
Chunks occupied by this partition

Return type
List[*bsb.storage._chunks.Chunk*]

`translate`(*translation*)

Translate the partition by the given offset.

Parameters
offset (*numpy.ndarray*) – Offset, XYZ.

Raises
exceptions.LayoutError if the translation needs to be rejected.

`volume`(*chunk=None*)

Calculate the volume of the partition in m^3.

Parameters
chunk (bsb.storage._chunks.Chunk) – If given, limit the volume of the partition inside of the chunk.

Returns
Volume of the partition (in the chunk)

Return type
float

`class bsb.topology.partition.Voxels`(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Partition*, ABC

Partition based on a set of voxels.

`chunk_to_voxels`(*chunk*)

Voxelize the partition's occupation in this chunk. Required to fill the partition with cells by the placement module.

Parameters
chunk (bsb.storage._chunks.Chunk) – The chunk to calculate voxels for.

Returns
The set of voxels that together make up the shape of this partition in this chunk.

Return type
bsb.voxels.VoxelSet

`get_layout`(*hint*)

Given a Layout as hint to begin from, create a Layout object that describes how this partition would like to be laid out.

Parameters
hint (bsb.topology._layout.Layout) – The layout space that this partition should place itself in.

Returns
The layout describing the space this partition takes up.

Return type
bsb.topology._layout.Layout

`get_node_name`()

`abstract get_voxelset`()

`rotate`(*rotation*)

Rotate the partition by the given rotation object.

Parameters
rotation (*scipy.spatial.transform.Rotation*) – Rotation object.

Raises
exceptions.LayoutError if the rotation needs to be rejected.

`scale`(*factor*)

Scale up/down the partition according to the given factors.

Parameters
factors (*numpy.ndarray*) – Scaling factors, XYZ.

Raises
exceptions.LayoutError if the scaling needs to be rejected.

`surface`(*chunk=None*)

Calculate the surface of the partition in m^2.

Parameters
chunk (bsb.storage._chunks.Chunk) – If given, limit the surface of the partition inside of the chunk.

Returns
Surface of the partition (in the chunk)

Return type
float

`to_chunks`(*chunk_size*)

Calculate all the chunks this partition occupies when cut into `chunk_sized` pieces.

Parameters
chunk_size (*numpy.ndarray*) – Size per chunk (in m). The slicing always starts at [0, 0, 0].

Returns
Chunks occupied by this partition

Return type
List[*bsb.storage._chunks.Chunk*]

`translate`(*offset*)

Translate the partition by the given offset.

Parameters
offset (*numpy.ndarray*) – Offset, XYZ.

Raises
exceptions.LayoutError if the translation needs to be rejected.

volume(*chunk=None*)

Calculate the volume of the partition in m^3.

Parameters
chunk (bsb.storage._chunks.Chunk) – If given, limit the volume of the partition inside of the chunk.

Returns
Volume of the partition (in the chunk)

Return type
float

property voxelset

bsb.topology.region module

Module for the Region types.

class bsb.topology.region.**Region**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: ABC

Base region.

When arranging will simply call arrange/layout on its children but won't cause any changes itself.

children: list[*Region* | *Partition*]

property data

do_layout(*hint*)

get_dependencies()

get_layout(*hint*)

get_node_name()

name: str

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

abstract rotate(*rotation*)

scaffold: *Scaffold*

abstract scale(*factors*)

abstract translate(*offset*)

type

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

class bsb.topology.region.**RegionGroup**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Region*

get_node_name()

rotate(*rotation*)

scale(*factors*)

translate(*offset*)

class bsb.topology.region.**Stack**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *RegionGroup*

Stack components on top of each other based on their `stack_index` and adjust its own height accordingly.

axis: `Literal['x'] | Literal['y'] | Literal['z']`

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

get_layout(*hint*)

get_node_name()

rotate(*rotation*)

scale(*factors*)

translate(*offset*)

Module contents

Topology module

bsb.topology.**box_layout**(*ldc*, *mdc*)

bsb.topology.**create_topology**(*regions*, *ldc*, *mdc*)

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.
- **ldc** – Least dominant corner of the topology. Forms the suggested outer bounds of the topology together with the *mdc*.
- **mdc** – Most dominant corner of the topology. Forms the suggested outer bounds of the topology together with the *mdc*.

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.

bsb.topology.**is_partition**(*obj*)

Checks if an object is a partition.

bsb.topology.**is_region**(*obj*)

Checks if an object is a region.

bsb.morphologies package

Subpackages

bsb.morphologies.parsers package

Module contents

bsb.morphologies.parsers.**parse_morphology_content**(*content: str | bytes*, *parser='bsb'*, ***kwargs*)

bsb.morphologies.parsers.**parse_morphology_file**(*file: str | PathLike*, *parser='bsb'*, ***kwargs*)

class bsb.morphologies.parsers.parser.**BsbParser**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *MorphologyParser*

get_node_name()

parse(*file:* FileDependency | *str*)

Parse the morphology

parse_content(*content: str*)

skip_boundary_labels: list[str]

A set of labels that is used to create gaps in a morphology at certain boundaries. No point will be inferred between a child branch of a branch labelled with the given labels; usually used to skip points between the soma and its child branches.

tags: dict[str | list[str]]

Dictionary mapping SWC tags to sets of morphology labels.

class bsb.morphologies.parsers.parser.**MorphIOParser**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *MorphologyParser*

flags

get_node_name()

parse(*file:* FileDependency | *str*) → *Morphology*

Parse the morphology

class bsb.morphologies.parsers.parser.**MorphologyParser**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

branch_cls: type

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

cls: type

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

get_node_name()

abstract parse(*file:* FileDependency | *str*) → *Morphology*

Parse the morphology

parser

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

Module contents

Morphology module

class bsb.morphologies.**Branch**(*points*, *radii*, *labels=None*, *properties=None*, *children=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

attach_child(*branch*)

Attach a branch as a child to this branch.

Parameters
branch (*Branch*) – Child branch

cached_voxelize(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids and cache the result.

Return type
bsb.voxels.VoxelSet

ceil_arc_point(*arc*)

Get the index of the nearest distal arc point.

center()

Center the morphology on the origin

property children

Collection of the child branches of this branch.

Returns
list of *Branches*

Return type
list

close_gaps()

Close any head-to-tail gaps between parent and child branches.

collapse(*on=None*)

Collapse all the roots of the morphology or subtree onto a single point.

Parameters
on (*int*) – Index of the root to collapse on. Collapses onto the origin by default.

contains_labels(*labels*)

Check if this branch contains any points labelled with any of the given labels.

Parameters
labels (*List[str]*) – The labels to check for.

Return type
bool

copy(*branch_class=None*)

Return a parentless and childless copy of the branch.

Parameters
branch_class (*type*) – Custom branch creation class

Returns
A branch, or *branch_class* if given, without parents or children.

Return type
bsb.morphologies.Branch

delete_point(*index*)

Remove a point from the branch

Parameters
index (*int*) – index position of the point to remove

Returns
the branch where the point has been removed

Return type
bsb.morphologies.Branch

detach()

Detach the branch from its parent, if one exists.

detach_child(*branch*)

Remove a branch as a child from this branch.

Parameters
branch (*Branch*) – Child branch

property end

Return the spatial coordinates of the terminal point of this branch.

property euclidean_dist

Return the Euclidean distance from the start to the terminal point of this branch.

find_closest_point(*coord*)

Return the index of the closest on this branch to a desired coordinate.

Parameters
coord – The coordinate to find the nearest point to

Type
numpy.ndarray

flatten()

Return the flattened points of the morphology or subtree.

Return type
numpy.ndarray

`flatten_labels()`

Return the flattened labels of the morphology or subtree.

Return type
numpy.ndarray

`flatten_properties()`

Return the flattened properties of the morphology or subtree.

Return type
numpy.ndarray

`flatten_radii()`

Return the flattened radii of the morphology or subtree.

Return type
numpy.ndarray

`floor_arc_point(arc)`

Get the index of the nearest proximal arc point.

`property fractal_dim`

Return the fractal dimension of this branch, computed as the coefficient of the line fitting the log-log plot of path vs euclidean distances of its points.

`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_axial_distances(idx_start=0, idx_end=-1, return_max=False)`

Return the displacements or its max value of a subset of branch points from its axis vector. :param idx_start = 0: index of the first point of the subset. :param idx_end = -1: index of the last point of the subset. :param return_max = False: if True the function only returns the max value of displacements, otherwise the entire array.

`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_label_mask(labels)`

Return a mask for the specified labels

Parameters
labels (*List[str] | numpy.ndarray[str]*) – The labels to check for.

Returns
A boolean mask that selects out the points that match the label.

Return type
List[numpy.ndarray]

get_points_labelled(*labels*)

Filter out all points with certain labels

Parameters
labels (*List[str] | numpy.ndarray[str]*) – The labels to check for.

Returns
All points with the labels.

Return type
List[numpy.ndarray]

insert_branch(*branch*, *index*)

Split this branch and insert the given `branch` at the specified `index`.

Parameters

- **branch** (*Branch*) – Branch to be attached
- **index** – Index or coordinates of the cutpoint; if coordinates are given, the closest point to the coordinates is used.

Type
Union[numpy.ndarray, int]

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(*labels*, *points=None*)

Add labels to the branch.

Parameters

- **labels** (*List[str]*) – Label(s) for the branch
- **points** – An integer or boolean mask to select the points to label.

property labels

Return the labels of the points on this branch. Labels are represented as a number that is associated to a set of labels. See *Labels* for more info.

property labelsets

Return the sets of labels associated to each numerical label.

list_labels()

Return a list of labels present on the branch.

property max_displacement

Return the max displacement of the branch points from its axis vector.

property parent

property path_length

Return the sum of the euclidean distances between the points on the branch.

property point_vectors

Return the individual vectors between consecutive points on this branch.

property points

Return the spatial coordinates of the points on this branch.

property radii

Return the radii of the points on this branch.

root_rotate(*rot*, *downstream_of=0*)

Rotate the subtree emanating from each root around the start of that root If downstream_of is provided, will rotate points starting from the index provided (only for subtrees with a single root).

Parameters

- **rot** (*scipy.spatial.transform.Rotation*) – Scipy rotation to apply to the subtree.
- **downstream_of** – index of the point in the subtree from which the rotation should be applied. This feature works only when the subtree has only one root branch.

Returns
rotated Morphology

Return type
bsb.morphologies.SubTree

`rotate`(*rotation*, *center=None*)

Point rotation

Parameters

- **rot** – Scipy rotation
- **center** (*numpy.ndarray*) – rotation offset point.

Type
Union[scipy.spatial.transform.Rotation, List[float,float,float]]

`property segments`

Return the start and end points of vectors between consecutive points on this branch.

`set_properties`(***kwargs*)

`simplify`(*epsilon*, *idx_start=0*, *idx_end=-1*)

Apply Ramer–Douglas–Peucker algorithm to all points or a subset of points of the branch. :param epsilon: Epsilon to be used in the algorithm. :param idx_start = 0: Index of the first element of the subset of points to be reduced. :param epsilon = -1: Index of the last element of the subset of points to be reduced.

`simplify_branches`(*epsilon*)

Apply Ramer–Douglas–Peucker algorithm to all points of all branches of the SubTree. :param epsilon: Epsilon to be used in the algorithm.

`property size`

Returns the amount of points on this branch

Returns
Number of points on the branch.

Return type
int

`property start`

Return the spatial coordinates of the starting point of this branch.

`subtree`(*labels=None*)

`translate`(*point*)

Translate the subtree by a 3D vector.

Parameters
point (*numpy.ndarray*) – 3D vector to translate the subtree.

Returns
the translated subtree

Return type
bsb.morphologies.SubTree

`property vector`

Return the vector of the axis connecting the start and terminal points.

`property versor`

Return the normalized vector of the axis connecting the start and terminal points.

voxelize(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids.

Return type
bsb.voxels.VoxelSet

walk()

Iterate over the points in the branch.

class bsb.morphologies.**Morphology**(*roots*, *meta=None*, *shared_buffers=None*, *sanitize=False*)

Bases: *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.

property adjacency_dictionary

Return a dictonary associating to each key (branch index) a list of adjacent branch indices

as_filtered(*labels=None*)

Return a filtered copy of the morphology that includes only points that match the current label filter, or the specified labels.

copy()

Copy the morphology.

classmethod empty()

get_label_mask(*labels*)

Get a mask corresponding to all the points labelled with 1 or more of the given labels

property is_optimized

property labelsets

Return the sets of labels associated to each numerical label.

list_labels()

Return a list of labels present on the morphology.

property meta

optimize(*force=False*)

set_label_filter(*labels*)

Set a label filter, so that *as_filtered* returns copies filtered by these labels.

simplify(**args*, *optimize=True*, ***kwargs*)

to_graph_array()

Create a SWC-like numpy array from a Morphology.

> **Warning:** Custom SWC tags (above 3) won't work and throw an error

Returns
a numpy array with columns storing the standard SWC attributes

Return type
numpy.ndarray

to_swc(*file*)

Create a SWC file from a Morphology. :param file: path to write to

class bsb.morphologies.**MorphologySet**(*loaders*, *m_indices=None*, */*, *labels=None*)

Bases: object

Associates a set of *StoredMorphologies* to cells

clear_soft_cache()

count_morphologies()

count_unique()

classmethod empty()

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

get_indices(*copy=True*)

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*)

property names

set_label_filter(*labels*)

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 branch_adjacency

Return a dictonary containing mapping the id of the branch to its children.

property branches

Return a depth-first flattened array of all branches.

`cached_voxelize`(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids and cache the result.

Return type
bsb.voxels.VoxelSet

`center`()

Center the morphology on the origin

`close_gaps`()

Close any head-to-tail gaps between parent and child branches.

`collapse`(*on=None*)

Collapse all the roots of the morphology or subtree onto a single point.

Parameters
on (*int*) – Index of the root to collapse on. Collapses onto the origin by default.

`flatten`()

Return the flattened points of the morphology or subtree.

Return type
numpy.ndarray

`flatten_labels`()

Return the flattened labels of the morphology or subtree.

Return type
numpy.ndarray

`flatten_properties`()

Return the flattened properties of the morphology or subtree.

Return type
numpy.ndarray

`flatten_radii`()

Return the flattened radii of the morphology or subtree.

Return type
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

`label`(*labels*, *points=None*)

Add labels to the morphology or subtree.

Parameters

- **labels** (*list[str]*) – Labels to add to the subtree.
- **points** (*numpy.ndarray*) – Optional boolean or integer mask for the points to be labelled.

property labels

property origin

property path_length

Return the total path length as the sum of the euclidian distances between consecutive points.

property points

property properties

property radii

root_rotate(*rot*, *downstream_of=0*)

Rotate the subtree emanating from each root around the start of that root If downstream_of is provided, will rotate points starting from the index provided (only for subtrees with a single root).

Parameters

- **rot** (*scipy.spatial.transform.Rotation*) – Scipy rotation to apply to the subtree.
- **downstream_of** – index of the point in the subtree from which the rotation should be applied. This feature works only when the subtree has only one root branch.

Returns
rotated Morphology

Return type
bsb.morphologies.SubTree

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

Point rotation

Parameters

- **rot** – Scipy rotation
- **center** (*numpy.ndarray*) – rotation offset point.

Type
Union[scipy.spatial.transform.Rotation, List[float,float,float]]

simplify_branches(*epsilon*)

Apply Ramer–Douglas–Peucker algorithm to all points of all branches of the SubTree. :param epsilon: Epsilon to be used in the algorithm.

property size

subtree(*labels=None*)

translate(*point*)

Translate the subtree by a 3D vector.

Parameters
point (*numpy.ndarray*) – 3D vector to translate the subtree.

Returns
the translated subtree

Return type
bsb.morphologies.SubTree

voxelize(*N*)

Turn the morphology or subtree into an approximating set of axis-aligned cuboids.

Return type
bsb.voxels.VoxelSet

bsb.morphologies.**branch_iter**(*branch*)

Iterate over a branch and all of its children depth first.

bsb.morphologies.**parse_morphology_file**(*file*, ***kwargs*)

class bsb.morphologies.selector.**MorphologySelector**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: ABC

get_node_name()

abstract pick(*morphology*)

scaffold: *Scaffold*

select

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

abstract validate(*all_morphos*)

class bsb.morphologies.selector.**NameSelector**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *MorphologySelector*

get_node_name()

names: *cfglist*[**str**]

pick(*morphology*)

validate(*all_morphos*)

class bsb.morphologies.selector.**NeuroMorphoSelector**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *NameSelector*

get_node_name()

bsb.placement package

Submodules

bsb.placement.arrays module

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

Implementation of the placement of cells in parallel arrays.

angle: float

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.

queue(*pool:* JobPool, *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.

spacing_x: float

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

bsb.placement.distributor module

class bsb.placement.distributor.**DistributionContext**(*indicator:* bsb.placement.indicator.PlacementIndications, *partitions:* *List[*bsb.topology.partition.Partition*]*)

indicator: *PlacementIndications*

partitions: List[*Partition*]

class bsb.placement.distributor.**Distributor**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

abstract distribute(*positions*, *context*)

Is called to distribute cell properties.

Parameters
partitions – The partitions the cells were placed in.

Returns
An array with the property data

Return type
numpy.ndarray

get_node_name()

strategy

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

class bsb.placement.distributor.**DistributorsNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

get_node_name()

morphologies: *MorphologyDistributor*

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

properties: dict[*Distributor*]

rotations: *RotationDistributor*

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

class bsb.placement.distributor.**ExplicitNoRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

distribute(*positions*, *context*)

Is called to distribute cell properties.

Parameters
partitions – The partitions the cells were placed in.

Returns
An array with the property data

Return type
numpy.ndarray

get_node_name()

class bsb.placement.distributor.**Implicit**

class bsb.placement.distributor.**ImplicitNoRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

distribute(*positions*, *context*)

Is called to distribute cell properties.

Parameters
partitions – The partitions the cells were placed in.

Returns
An array with the property data

Return type
numpy.ndarray

get_node_name()

class bsb.placement.distributor.**MorphologyDistributor**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

abstract distribute(*positions*, *morphologies*, *context*)

Is called to distribute cell morphologies and optionally rotations.

Parameters

- **positions** (*numpy.ndarray*) – Placed positions under consideration
- **morphologies** – The template morphology loaders. You can decide to use them and/or generate new ones in the MorphologySet that you produce. If you produce any new morphologies, don't forget to encapsulate them in a *StoredMorphology* loader, or better yet, use the *MorphologyGenerator*.
- **context** (DistributionContext) – The placement indicator and partitions.

Returns
A MorphologySet with assigned morphologies, and optionally a RotationSet

Return type
Union[MorphologySet, Tuple[MorphologySet, RotationSet]]

get_node_name()

may_be_empty

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

strategy

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

class bsb.placement.distributor.**MorphologyGenerator**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Special case of the morphology distributor that provides extra convenience when generating new morphologies.

distribute(*positions*, *morphologies*, *context*)

Is called to distribute cell morphologies and optionally rotations.

Parameters

- **positions** (*numpy.ndarray*) – Placed positions under consideration

- **morphologies** – The template morphology loaders. You can decide to use them and/or generate new ones in the MorphologySet that you produce. If you produce any new morphologies, don't forget to encapsulate them in a *StoredMorphology* loader, or better yet, use the *MorphologyGenerator*.
- **context** (DistributionContext) – The placement indicator and partitions.

Returns
A MorphologySet with assigned morphologies, and optionally a RotationSet

Return type
Union[MorphologySet, Tuple[MorphologySet, RotationSet]]

abstract generate(*positions*, *morphologies*, *context*)

get_node_name()

may_be_empty

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

class bsb.placement.distributor.**RandomMorphologies**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Distributes selected morphologies randomly without rotating them.

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

distribute(*positions*, *morphologies*, *context*)

Uses the morphology selection indicators to select morphologies and returns a MorphologySet of randomly assigned morphologies

get_node_name()

may_be_empty

class bsb.placement.distributor.**RandomRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

distribute(*positions*, *context*)

Is called to distribute cell properties.

Parameters
partitions – The partitions the cells were placed in.

Returns
An array with the property data

Return type
numpy.ndarray

get_node_name()

class bsb.placement.distributor.**RotationDistributor**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Rotates everything by nothing!

abstract distribute(*positions*, *context*)

Is called to distribute cell properties.

Parameters
partitions – The partitions the cells were placed in.

Returns
An array with the property data

Return type
numpy.ndarray

get_node_name()

strategy

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

class bsb.placement.distributor.**RoundRobinMorphologies**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Distributes selected morphologies round robin, values are looped and assigned one by one in order.

```
{ "placement": { "place_XY": {
  "distribute": {
      "morphologies": {"strategy": "roundrobin"}
  }
}}}
```

distribute(*positions*, *morphologies*, *context*)

Is called to distribute cell morphologies and optionally rotations.

Parameters

- **positions** (*numpy.ndarray*) – Placed positions under consideration
- **morphologies** – The template morphology loaders. You can decide to use them and/or generate new ones in the MorphologySet that you produce. If you produce any new morphologies, don't forget to encapsulate them in a *StoredMorphology* loader, or better yet, use the *MorphologyGenerator*.
- **context** (DistributionContext) – The placement indicator and partitions.

Returns
A MorphologySet with assigned morphologies, and optionally a RotationSet

Return type
Union[MorphologySet, Tuple[MorphologySet, RotationSet]]

get_node_name()

may_be_empty

class bsb.placement.distributor.**VolumetricRotations**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

default_vector

Default orientation vector of each position.

distribute(*positions*, *context*)

Rotates according to a volumetric orientation field of specific resolution. For each position, find the equivalent voxel in the volumetric orientation field and apply the rotation from the default_vector to the corresponding orientation vector. Positions outside the orientation field will not be rotated.

Parameters

- **positions** – Placed positions under consideration. Its shape is (N, 3) where N is the number of positions.
- **context** (DistributionContext) – The placement indicator and partitions.

Returns
A RotationSet object containing the 3D Euler angles in degrees for the rotation of each position.

Return type
RotationSet

`get_node_name()`

`orientation_path`
Path to the nrrd file containing the volumetric orientation field. It provides a rotation for each voxel considered. Its shape should be (3, L, W, D) where L, W and D are the sizes of the field.

`orientation_resolution`
Voxel size resolution of the orientation field.

`space_origin`
Origin point for the orientation field.

bsb.placement.indicator module

`class bsb.placement.indicator.PlacementIndicator(strat, cell_type)`

`assert_indication(attr)`

`property cell_type`

`get_radius()`

`guess(chunk=None, voxels=None)`
Estimate the count of cell to place based on the cell_type's PlacementIndications. Float estimates are converted to int using an acceptance-rejection method.

Parameters

- **chunk** (bsb.storage._chunks.Chunk) – if provided, will estimate the number of cell within the Chunk.
- **voxels** (bsb.voxels.VoxelSet) – if provided, will estimate the number of cell within the VoxelSet. Only for cells with the indication "density_key" set or with the indication "relative_to" set and the target cell has the indication "density_key" set.

Returns
Cell counts for each chunk or voxel.

Return type
numpy.ndarray[int]

`indication(attr)`

`use_morphologies()`

bsb.placement.random module

class bsb.placement.random.**RandomPlacement**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Place cells in random positions.

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.

Parameters

- **chunk** (bsb.storage._chunks.Chunk) – Chunk to fill
- **indicators** (*Mapping[str, bsb.placement.indicator.PlacementIndicator]*) – Dictionary of each cell type to its PlacementIndicator

bsb.placement.strategy module

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

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.

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.

Parameters

- **chunk** (bsb.storage._chunks.Chunk) – Chunk to fill
- **indicators** (*Mapping[str, bsb.placement.indicator.PlacementIndicator]*) – Dictionary of each cell type to its PlacementIndicator

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.**FixedPositions**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

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.

Parameters

- **chunk** (bsb.storage._chunks.Chunk) – Chunk to fill
- **indicators** (*Mapping[str, bsb.placement.indicator.PlacementIndicator]*) – Dictionary of each cell type to its PlacementIndicator

positions: *ndarray*

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

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.**PlacementStrategy**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

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

cell_types: *list[CellType]*

depends_on: *list[PlacementStrategy]*

distribute: *DistributorsNode*

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

get_deps()

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()

guess_cell_count()

indicator_class

alias of *PlacementIndicator*

is_entities()

name: *str*

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

overrides: *cfgdict[PlacementIndications]*

partitions: *list[Partition]*

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.

Parameters

- **chunk** (bsb.storage._chunks.Chunk) – Chunk to fill
- **indicators** (*Mapping[str, bsb.placement.indicator.PlacementIndicator]*) – Dictionary of each cell type to its PlacementIndicator

place_cells(*indicator*, *positions*, *chunk*, *additional=None*)

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.

scaffold: *Scaffold*

strategy

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

Module contents

bsb.connectivity package

Subpackages

bsb.connectivity.detailed package

Submodules

bsb.connectivity.detailed.shared module

class bsb.connectivity.detailed.shared.**Intersectional**

Bases: object

affinity

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

candidate_intersection(*target_coll*, *candidate_coll*)

get_region_of_interest(*chunk*)

bsb.connectivity.detailed.voxel_intersection module

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

Bases: *Intersectional*, *ConnectionStrategy*

This strategy finds overlap between voxelized morphologies.

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()

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: *ConnectionStrategy*

All to all connectivity between two neural populations

connect(*pre*, *post*)

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

Bases: *ConnectionStrategy*

Connect cells based on a convergence distribution, i.e. by connecting each source cell to X target cells.

connect()

convergence: *Distribution*

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

get_node_name()

class bsb.connectivity.general.**FixedIndegree**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *InvertedRoI*, *ConnectionStrategy*

Connect a group of postsynaptic cell types to `indegree` uniformly random presynaptic cells from all the presynaptic cell types.

connect(*pre*, *post*)

get_node_name()

indegree: int

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

bsb.connectivity.strategy module

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

Bases: ABC, *HasDependencies*

abstract connect(*presyn_collection*, *postsyn_collection*)

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

depends_on: list[*ConnectionStrategy*]

The list of strategies that must run before this one

get_all_post_chunks()

get_all_pre_chunks()

get_cell_types()

get_deps()

get_node_name()

get_output_names(*pre=None*, *post=None*)

get_region_of_interest(*chunk*)

name: str

Name used to refer to the connectivity strategy

output_naming: str | None | dict[str, dict[str, str, None, list[str]]]

Specifies how to name the output ConnectivitySets in which the connections between cell type pairs are stored.

postsynaptic: *Hemitype*

Postsynaptic (target) neuron population

presynaptic: *Hemitype*

Presynaptic (source) neuron population

queue(*pool:* JobPool)

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.

scaffold: *Scaffold*

strategy

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

class bsb.connectivity.strategy.**Hemitype**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

Class used to represent one (pre- or postsynaptic) side of a connection rule.

cell_types: list[*CellType*]

List of cell types to use in connection.

get_node_name()

labels: list[str]

List of labels to filter the placement set by.

morpho_loader: Callable[[*PlacementSet*], *MorphologySet*]

Function to load the morphologies (MorphologySet) from a PlacementSet. This override can allow temporary dynamic morphology generation during the connectivity phase, from a much smaller, or empty, MorphologySet. It is useful for example when the task would take too much disk space or time otherwise.

morphology_labels: list[str]

List of labels to filter the morphologies by.

scaffold: *Scaffold*

class bsb.connectivity.strategy.**HemitypeCollection**(*hemitype*, *roi*)

Bases: object

property placement

Module contents

bsb.simulation package

Submodules

bsb.simulation.simulation module

class bsb.simulation.simulation.**ProgressEvent**(*progression*, *duration*, *time*)

Bases: object

class bsb.simulation.simulation.**Simulation**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

cell_models: *cfgdict*[*CellModel*]

connection_models: *cfgdict*[*ConnectionModel*]

devices: *cfgdict*[*DeviceModel*]

duration: float

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

get_connectivity_sets() → Mapping[*ConnectionModel*, *ConnectivitySet*]

get_model_of(*type:* CellType | ConnectionStrategy) → *CellModel* | *ConnectionModel* | None

get_node_name()

name: str

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

post_prepare: *cfglist*[Callable[[*Simulation*, Any], None]]

scaffold: *Scaffold*

simulator: str

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

bsb.simulation.adapter module

class bsb.simulation.adapter.**AdapterProgress**(*duration*)

Bases: object

complete()

steps(*step=1*)

tick(*step*)

Report simulation progress.

class bsb.simulation.adapter.**SimulationData**(*simulation:* Simulation, *result=None*)

Bases: object

class bsb.simulation.adapter.**SimulatorAdapter**

Bases: ABC

add_progress_listener(*listener*)

collect(*simulation*, *simdata*, *simresult*, *comm=None*)

Collect the output of a simulation that completed

abstract prepare(*simulation*, *comm=None*)

Reset the simulation backend and prepare for the given simulation.

Parameters

- **simulation** (Simulation) – The simulation configuration to prepare.
- **comm** – The mpi4py MPI communicator to use. Only nodes in the communicator will participate in the simulation. The first node will idle as the main node.

abstract run(**simulations*, *comm=None*)

Fire up the prepared adapter.

simulate(**simulations*, *post_prepare=None*, *comm=None*)

Simulate the given simulations.

bsb.simulation.cell module

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

Bases: *SimulationComponent*

Cell models are simulator specific representations of a cell type.

cell_type: *CellType*

The cell type that this model represents

get_node_name()

get_placement_set(*chunks=None*)

parameters: *cfglist*[*Parameter*]

The parameters of the model.

bsb.simulation.component module

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

Bases: ABC

get_node_name()

name: str

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

property simulation

bsb.simulation.connection module

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

Bases: *SimulationComponent*

get_connectivity_set()

get_node_name()

tag: str

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

bsb.simulation.device module

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

Bases: *SimulationComponent*

get_node_name()

implement(*adapter*, *simulation*, *simdata*)

bsb.simulation.parameter module

class bsb.simulation.parameter.**Parameter**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

get_node_name()

type

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

value: *ParameterValue*

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

class bsb.simulation.parameter.**ParameterValue**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

get_node_name()

type

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

bsb.simulation.results module

class bsb.simulation.results.**SimulationRecorder**

Bases: object

flush(*segment: neo.core.Segment*)

class bsb.simulation.results.**SimulationResult**(*simulation*)

Bases: object

add(*recorder*)

property analogsignals

create_recorder(*flush: Callable[[neo.core.Segment], None]*)

flush()

property spiketrains

write(*filename*, *mode*)

bsb.simulation.targetting module

class bsb.simulation.targetting.**BranchLocTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *LabelTargetting*

get_locations(*cell*)

get_node_name()

x

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

class bsb.simulation.targetting.**ByIdTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *FractionFilter*, *CellTargetting*

Targets all given identifiers.

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

ids: dict[str, list[int]]

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

class bsb.simulation.targetting.**ByLabelTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *CellModelFilter*, *FractionFilter*, *CellTargetting*

Targets all given labels.

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

labels: list[str]

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

class bsb.simulation.targetting.**CellModelFilter**

Bases: object

cell_models: list[*CellModel*]

get_targets(*adapter*, *simulation*, *simdata*)

class bsb.simulation.targetting.**CellModelTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *CellModelFilter*, *FractionFilter*, *CellTargetting*

Targets all cells of certain cell models.

cell_models: list[*CellModel*]

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

class bsb.simulation.targeting.**CellTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Targetting*

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

type: Literal['cell'] | Literal['connection']

class bsb.simulation.targeting.**ConnectionTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *Targetting*

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

type: Literal['cell'] | Literal['connection']

class bsb.simulation.targeting.**CylindricalTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *CellModelFilter*, *FractionFilter*, *CellTargetting*

Targets all cells in a cylinder along specified axis.

axis: Literal['x'] | Literal['y'] | Literal['z']

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

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

Target all or certain cells within a cylinder of specified radius.

origin: list[float]

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

radius: float

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

class bsb.simulation.targeting.**FractionFilter**

Bases: object

count

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

static filter(*f*)

fraction

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

satisfy_fractions(*targets*)

class bsb.simulation.targeting.**LabelTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *LocationTargetting*

get_locations(*cell*)

get_node_name()

labels

class bsb.simulation.targetting.**LocationTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

get_locations(*cell*)

get_node_name()

strategy

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: *CellModelFilter*, *FractionFilter*, *CellTargetting*

Targets all identifiers of certain cell types.

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

n: int

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

class bsb.simulation.targetting.**SomaTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *LocationTargetting*

get_locations(*cell*)

get_node_name()

class bsb.simulation.targetting.**SphericalTargetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *CellModelFilter*, *FractionFilter*, *CellTargetting*

Targets all cells in a sphere.

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

Target all or certain cells within a sphere of specified radius.

origin: list[float]

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

radius: float

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

class bsb.simulation.targetting.**Targetting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

get_node_name()

get_targets(*adapter*, *simulation*, *simdata*)

strategy

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

type: Literal['cell'] | Literal['connection']

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

Module contents

class bsb.simulation.**SimulationBackendPlugin**(*Adapter:* bsb.simulation.adapter.SimulatorAdapter, *Simulation:* bsb.simulation.simulation.Simulation)

Bases: object

Adapter: *SimulatorAdapter*

Simulation: *Simulation*

bsb.simulation.**get_simulation_adapter**(*name: str*)

bsb.services package

Submodules

bsb.services.mpi module

class bsb.services.mpi.**MPIModule**(*module*)

Bases: *MockModule*

Module provider of the MPI interface.

property COMM_WORLD

class bsb.services.mpi.**MPIService**

Bases: object

abort(*errorcode=1*)

allgather(*obj*)

barrier()

bcast(*obj*, *root=0*)

gather(*obj*, *root=0*)

get_communicator()

get_rank()

get_size()

bsb.services.mpilock module

class bsb.services.mpilock.**Fence**(*access*)

Bases: object

collect()

guard()

share(*obj*)

exception bsb.services.mpilock.**FencedSignal**

Bases: Exception

class bsb.services.mpilock.**MPILockModule**(*module*)

Bases: *MockModule*

sync(*comm=None*, *master=0*)

class bsb.services.mpilock.**MockedWindowController**(*comm=None*, *master=0*)

Bases: object

close()

property closed

property master

property rank

read()

single_write(*handle=None*, *rank=None*)

write()

bsb.services.pool module

Job pooling module.

Jobs derive from the base *Job* class which can be put on the queue of a *JobPool*. In order to submit themselves to the pool Jobs will *serialize()* themselves into a predefined set of variables:

```
job.serialize() -> (job_type, f, args, kwargs)
```

- **job_type should be a string that is a class name defined in this module.**
  (e.g. "PlacementJob")
- **f should be the function object that the job's execute method should**
  execute.
- args and kwargs are the args to be passed to that f.

The *Job.execute()* handler can help interpret args and kwargs before running f. The execute handler has access to the scaffold on the MPI process so one best serializes just the name of some part of the configuration, rather than trying to pickle the complex objects. For example, the *PlacementJob* uses the first args element to store the *PlacementStrategy* name and then retrieve it from the scaffold:

```
@staticmethod
def execute(job_owner, f, args, kwargs):
    placement = job_owner.placement[args[0]]
    indicators = placement.get_indicators()
    return f(placement, *args[1:], indicators, **kwargs)
```

A job has a couple of display variables that can be set: _cname for the class name, _name for the job name and _c for the chunk. These are used to display what the workers are doing during parallel execution. This is an experimental API and subject to sudden change in the future.

class bsb.services.pool.**ConnectivityJob**(*pool*, *strategy*, *pre_roi*, *post_roi*, *deps=None*)

Dispatches the execution of a chunk of a connectivity strategy through a JobPool.

static execute(*job_owner*, *args*, *kwargs*)

Job handler

class bsb.services.pool.**FunctionJob**(*pool*, *f*, *args*, *kwargs*, *deps=None*, ***context*)

static execute(*job_owner*, *args*, *kwargs*)

Job handler

class bsb.services.pool.**Job**(*pool*, *submission_context:* SubmissionContext, *args*, *kwargs*, *deps=None*)

Dispatches the execution of a function through a JobPool

cancel(*reason: str | None = None*)

change_status(*status:* JobStatus)

property context

property description

property error

abstract static execute(*job_owner*, *args*, *kwargs*)

Job handler

property name

on_completion(*cb*)

property result

run(*timeout=None*)

Execute the job on the current process, in a thread, and return whether the job is still running.

serialize()

Convert the job to a (de)serializable representation

set_exception(*e: Exception*)

set_result(*value*)

property status

property submitter

exception bsb.services.pool.**JobErroredError**(*message*, *error*)

class bsb.services.pool.**JobPool**(*scaffold*, *fail_fast=False*, *workflow:* Workflow | *None = None*)

add_listener(*listener*, *max_wait=None*)

add_notification(*notification:* PoolProgress)

change_status(*status:* PoolStatus)

execute(*return_results=False*)

Execute the jobs in the queue

In serial execution this runs all of the jobs in the queue in First In First Out order. In parallel execution this enqueues all jobs into the MPIPool unless they have dependencies that need to complete first.

classmethod get_owner(*id*)

get_submissions_of(*submitter*)

classmethod get_tmp_folder(*id*)

is_main()

property jobs: list[*Job*]

notify()

property owner

property parallel

ping()

queue(*f*, *args=None*, *kwargs=None*, *deps=None*, ***context*)

queue_connectivity(*strategy*, *pre_roi*, *post_roi*, *deps=None*)

queue_placement(*strategy*, *chunk*, *deps=None*)

raise_unhandled()

schedule(*nodes*, *scheduler=None*)

property scheduling

property status

property workflow

class bsb.services.pool.**JobStatus**(*value*)

An enumeration.

ABORTED = 'aborted'

CANCELLED = 'cancelled'

FAILED = 'failed'

PENDING = 'pending'

QUEUED = 'queued'

RUNNING = 'running'

SUCCESS = 'success'

class bsb.services.pool.**PlacementJob**(*pool*, *strategy*, *chunk*, *deps=None*)

Dispatches the execution of a chunk of a placement strategy through a JobPool.

static execute(*job_owner*, *args*, *kwargs*)

Job handler

class bsb.services.pool.**PoolJobAddedProgress**(*pool:* JobPool, *job:* Job)

property job

class bsb.services.pool.**PoolJobUpdateProgress**(*pool:* JobPool, *job:* Job, *old_status:* JobStatus)

property job

property old_status

property status

class bsb.services.pool.**PoolProgress**(*pool:* JobPool, *reason:* PoolProgressReason)

Class used to report pool progression to listeners.

property jobs

property reason

property status

property workflow

class bsb.services.pool.**PoolProgressReason**(*value*)

An enumeration.

JOB_ADDED = 2

JOB_STATUS_CHANGE = 3

MAX_TIMEOUT_PING = 4

POOL_STATUS_CHANGE = 1

class bsb.services.pool.**PoolStatus**(*value*)

An enumeration.

CLOSING = 'closing'

EXECUTING = 'executing'

SCHEDULING = 'scheduling'

class bsb.services.pool.**PoolStatusProgress**(*pool:* JobPool, *old_status:* PoolStatus)

class bsb.services.pool.**SubmissionContext**(*submitter*, *chunks=None*, ***kwargs*)

Context information on who submitted a certain job.

property chunks

property context

property name

property submitter

class bsb.services.pool.**Workflow**(*phases: list[str]*)

property finished

next_phase()

property phase

property phases

exception bsb.services.pool.**WorkflowError**(*_ExceptionGroup__message: str*, *_ExceptionGroup__exceptions: Sequence[_ExceptionT_co]*)

bsb.services.pool.**dispatcher**(*pool_id*, *job_args*)

Developer modules

class bsb.services._util.**ErrorModule**(*message*)

Bases: object

class bsb.services._util.**MockModule**(*module*)

Bases: object

class bsb.services._util.**_ExceptionT_co**

bsb.storage package

Subpackages

Submodules

bsb.storage.interfaces module

class bsb.storage.interfaces.**ConnectivityIterator**(*cs: ConnectivitySet*, *direction*, *lchunks=None*, *gchunks=None*, *scoped=True*)

Bases: object

all()

as_globals()

as_scoped()

chunk_iter()

Iterate over the connection data chunk by chunk.

Returns

The presynaptic chunk, presynaptic locations, postsynaptic chunk, and postsynaptic locations.

Return type

Tuple[Chunk, numpy.ndarray, Chunk, numpy.ndarray]

from_(*chunks*)

incoming()

outgoing()

to(*chunks*)

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

Bases: *Interface*

Stores the connections between 2 types of cell as `local` and `global` locations. A location is a cell id, referring to the n-th cell in the chunk, a branch id, and a point id, to specify the location on the morphology. Local locations refer to cells on this chunk, while global locations can come from any chunk and is associated to a certain chunk id as well.

Locations are either placement-context or chunk dependent: You may form connections between the n-th cells of a placement set (using *connect()*), or of the n-th cells of 2 chunks (using *chunk_connect()*).

A cell has both incoming and outgoing connections; when speaking of incoming connections, the local locations are the postsynaptic cells, and when speaking of outgoing connections they are the presynaptic cells. Vice versa for the global connections.

abstract chunk_connect(*src_chunk*, *dst_chunk*, *src_locs*, *dst_locs*)

Must connect the `src_locs` to the `dest_locs`, interpreting the cell ids (first column of the locs) as the cell rank in the chunk.

abstract clear(*chunks=None*)

Must clear (some chunks of) the placement set

abstract connect(*pre_set*, *post_set*, *src_locs*, *dest_locs*)

Must connect the `src_locs` to the `dest_locs`, interpreting the cell ids (first column of the locs) as the cell rank in the placement set.

abstract classmethod create(*engine*, *tag*)

Must create the placement set.

abstract static exists(*engine*, *tag*)

Must check the existence of the connectivity set

abstract flat_iter_connections(*direction=None*, *local_=None*, *global_=None*)

Must iterate over the connectivity data, yielding the direction, local chunk, global chunk, and data:

```
for dir, lchunk, gchunk, data in self.flat_iter_connections():
    print(f"Flat {dir} block between {lchunk} and {gchunk}")
```

If a keyword argument is given, that axis is not iterated over, and the value is fixed in each iteration.

abstract get_global_chunks(*direction*, *local_*)

Must list all the global chunks that contain data coming from a `local` chunk in the given `direction`

abstract get_local_chunks(*direction*)

Must list all the local chunks that contain data in the given `direction` (`"inc"` or `"out"`).

abstract classmethod get_tags(*engine*)

Must return the tags of all existing connectivity sets.

Parameters
engine – Storage engine to inspect.

abstract load_block_connections(*direction*, *local_*, *global_*)

Must load the connections from `direction` perspective between `local_` and `global_`.

Returns
The local and global connections locations

Return type
Tuple[numpy.ndarray, numpy.ndarray]

`load_connections()`

Loads connections as a `CSIterator`.

Returns
A connectivity set iterator, that will load data

`abstract load_local_connections(direction, local_)`

Must load all the connections from `direction` perspective in `local_`.

Returns
The local connection locations, a vector of the global connection chunks (1 chunk id per connection), and the global connections locations. To identify a cell in the global connections, use the corresponding chunk id from the second return value.

Return type
Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

`abstract nested_iter_connections(direction=None, local_=None, global_=None)`

Must iterate over the connectivity data, leaving room for the end-user to set up nested for loops:

```
for dir, itr in self.nested_iter_connections():
    for lchunk, itr in itr:
        for gchunk, data in itr:
            print(f"Nested {dir} block between {lchunk} and {gchunk}")
```

If a keyword argument is given, that axis is not iterated over, and the amount of nested loops is reduced.

`post_type: CellType`

`post_type_name: str`

`pre_type: CellType`

`pre_type_name: str`

`require(engine, tag)`

Must make sure the connectivity set exists. The default implementation uses the class's `exists` and `create` methods.

`tag: str`

`class bsb.storage.interfaces.Engine(root, comm)`

Bases: *Interface*

Engines perform the transactions that come from the storage object, and read/write data in a specific format. They can perform collective or individual actions.

> **Warning:** Collective actions can only be performed from all nodes, or deadlocks occur. This means in particular that they may not be called from component code.

abstract clear_connectivity()

collective Must clear existing connectivity data.

abstract clear_placement()

collective Must clear existing placement data.

property comm

The communicator in charge of collective operations.

abstract copy(*new_root*)

collective Must copy the storage object to the new root.

abstract create()

collective Must create the storage engine.

abstract exists()

Must check existence of the storage object.

property format

Name of the type of engine. Automatically set through the plugin system.

abstract get_chunk_stats()

readonly Must return a dictionary with all chunk statistics.

abstract move(*new_root*)

collective Must move the storage object to the new root.

classmethod peek_exists(*root*)

Must peek at the existence of the given root, without instantiating anything.

read_only()

A context manager that enters the engine into readonly mode. In readonly mode the engine does not perform any locking, write-operations or network synchronization, and errors out if a write operation is attempted.

readwrite()

abstract recognizes(*root*)

Must return whether the given argument is recognized as a valid storage object.

abstract remove()

collective Must remove the storage object.

property root

The unique identifier for the storage. Usually pathlike, but can be anything.

abstract property root_slug

Must return a pathlike unique identifier for the root of the storage object.

set_comm(*comm*)

collective Set a new communicator in charge of collective operations.

abstract property versions

Must return a dictionary containing the version of the engine package, and bsb package, used to last write to this storage object.

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

Bases: *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.

find_file(*predicate*)

find_files(*predicate*)

find_id(*id*)

find_meta(*key*, *value*)

get(*id*) → *StoredFile*

Return a StoredFile wrapper

abstract get_encoding(*id*)

Must return the encoding of the file with the given id, or None if it is unspecified binary data.

abstract get_meta(*id*) → Mapping[str, Any]

Must return the metadata of the given id.

abstract get_mtime(*id*)

Must return the last modified timestamp of file with the given id.

abstract has(*id*)

Must return whether the file store has a file with the given id.

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*, *encoding=None*, *overwrite=False*)

Store content in the file store. Should also store the current timestamp as *mtime* meta.

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
- **encoding** (*str*) – Optional encoding
- **overwrite** (*bool*) – Overwrite existing file

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

class bsb.storage.interfaces.**GeneratedMorphology**(*name*, *generated*, *meta*)

Bases: *StoredMorphology*

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

Bases: ABC

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

Bases: *Interface*

abstract all()

Fetch all of the stored morphologies.

Returns
List of the stored morphologies.

Return type
List[StoredMorphology]

abstract get_all_meta()

Get the metadata of all stored morphologies. :returns: Metadata dictionary :rtype: dict

abstract get_meta(*name*)

Get the metadata of a stored morphology.

Parameters
name (*str*) – Key of the stored morphology.

Returns
Metadata dictionary

Return type
dict

abstract has(*name*)

Check whether a morphology under the given name exists

Parameters
name (*str*) – Key of the stored morphology.

Returns
Whether the key exists in the repo.

Return type
bool

list()

List all the names of the morphologies in the repository.

abstract load(*name*)

Load a stored morphology as a constructed morphology object.

Parameters
name (*str*) – Key of the stored morphology.

Returns
A morphology

Return type
Morphology

abstract preload(*name*)

Load a stored morphology as a morphology loader.

Parameters
name (*str*) – Key of the stored morphology.

Returns
The stored morphology

Return type
StoredMorphology

abstract save(*name*, *morphology*, *overwrite=False*)

Store a morphology

Parameters

- **name** (*str*) – Key to store the morphology under.
- **morphology** (bsb.morphologies.Morphology) – Morphology to store
- **overwrite** (*bool*) – Overwrite any stored morphology that already exists under that name

Returns
The stored morphology

Return type
StoredMorphology

abstract select(**selectors*)

Select stored morphologies.

Parameters
selectors (*List[bsb.morphologies.selector.MorphologySelector]*) – Any number of morphology selectors.

Returns
All stored morphologies that match at least one selector.

Return type
List[StoredMorphology]

abstract set_all_meta(*all_meta*)

Set the metadata of all stored morphologies. :param all_meta: Metadata dictionary. :type all_meta: dict

abstract update_all_meta(*meta*)

Update the metadata of stored morphologies with the provided key values

Parameters
meta (*str*) – Metadata dictionary.

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

Bases: *Interface*

class bsb.storage.interfaces.**NoopLock**

Bases: object

class bsb.storage.interfaces.**PlacementSet**(*engine*, *cell_type*)

Bases: *Interface*

Interface for the storage of placement data of a cell type.

abstract append_additional(*name*, *chunk*, *data*)

Append arbitrary user data to the placement set. The length of the data must match that of the placement set, and must be storable by the engine.

Parameters

- **name** –
- **chunk** (Chunk) – The chunk to store data in.
- **data** (*numpy.ndarray*) – Arbitrary user data. You decide

abstract append_data(*chunk*, *positions=None*, *morphologies=None*, *rotations=None*, *additional=None*, *count=None*)

Append data to the placement set. If any of `positions`, `morphologies`, or `rotations` is given, the arguments to its left must also be given (e.g. passing morphologies, but no positions, is not allowed, passing just positions is allowed)

Parameters

- **chunk** (Chunk) – The chunk to store data in.
- **positions** (*numpy.ndarray*) – Cell positions
- **rotations** (RotationSet) – Cell rotations
- **morphologies** (MorphologySet) – Cell morphologies
- **additional** (*Dict[str, numpy.ndarray]*) – Additional datasets with 1 value per cell, will be stored under its key in the dictionary
- **count** (*int*) – Amount of entities to place. Excludes the use of any positional, rotational or morphological data.

property cell_type

The associated cell type.

Returns
The cell type

Return type
CellType

abstract chunk_context(*chunks*)

abstract clear(*chunks=None*)

Clear (some chunks of) the placement set.

Parameters
chunks (*List[bsb.storage._chunks.Chunk]*) – If given, the specific chunks to clear.

count_morphologies()

Must return the number of different morphologies used in the set.

abstract classmethod create(*engine*, *cell_type*)

Create a placement set.

Parameters

- **engine** (*bsb.storage.interfaces.Engine*) – The engine that governs this PlacementSet.
- **cell_type** (bsb.cell_types.CellType) – The cell type whose data is stored in the placement set.

Returns
A placement set

Return type
bsb.storage.interfaces.PlacementSet

abstract static exists(*engine*, *cell_type*)

Check existence of a placement set.

Parameters

- **engine** (*bsb.storage.interfaces.Engine*) – The engine that governs the existence check.
- **cell_type** (bsb.cell_types.CellType) – The cell type to look for.

Returns
Whether the placement set exists.

Return type
bool

abstract get_all_chunks()

Get all the chunks that exist in the placement set.

Returns
List of existing chunks.

Return type
List[*bsb.storage._chunks.Chunk*]

abstract get_chunk_stats()

Should return how many cells were placed in each chunk.

abstract get_label_mask(*labels*)

Should return a mask that fits the placement set for the cells with given labels.

Parameters

- **cells** (*numpy.ndarray*) – Array of cells in this set to label.
- **labels** (*list[str]*) – List of labels

abstract get_labelled(*labels*)

Should return the cells labelled with given labels.

Parameters

- **cells** (*numpy.ndarray*) – Array of cells in this set to label.
- **labels** (*list[str]*) – List of labels

abstract label(*labels*, *cells*)

Should label the cells with given labels.

Parameters

- **cells** (*numpy.ndarray*) – Array of cells in this set to label.
- **labels** (*list[str]*) – List of labels

abstract load_additional(*key=None*)

load_box_tree(*morpho_cache=None*)

Load boxes, and form an RTree with them, for fast spatial lookup of rhomboid intersection.

Parameters
morpho_cache – See *load_boxes()*.

Returns
A boxtree

Return type
bsb.trees.BoxTree

load_boxes(*morpho_cache=None*)

Load the cells as axis aligned bounding box rhomboids matching the extension, orientation and position in space. This function loads morphologies, unless a *morpho_cache* is given, then that is used.

Parameters
morpho_cache (MorphologySet) – If you've previously loaded morphologies with soft or hard caching enabled, you can pass the resulting morphology set here to reuse it. If afterwards you need the morphology set, you best call *load_morphologies()* first and reuse it here.

Returns
An iterator with 6 coordinates per cell: 3 min and 3 max coords, the bounding box of that cell's translated and rotated morphology.

Return type
Iterator[Tuple[float, float, float, float, float, float]]

Raises
DatasetNotFoundError if no morphologies are found.

abstract load_ids()

abstract load_morphologies(*allow_empty=False*)

Return a *MorphologySet* associated to the cells. Raises an error if there is no morphology data, unless *allow_empty=True*.

Parameters
allow_empty (*bool*) – Silence missing morphology data error, and return an empty morphology set.

Returns
Set of morphologies

Return type
MorphologySet

abstract load_positions()

Return a dataset of cell positions.

Returns
An (Nx3) dataset of positions.

Return type
numpy.ndarray

abstract load_rotations()

Load the rotation data of the placement set :returns: A rotation set :rtype: ~bsb.morphologies.RotationSet

classmethod require(*engine*, *type*)

Return and create a placement set, if it didn't exist before.

The default implementation uses the *exists()* and *create()* methods.

Parameters

- **engine** (*bsb.storage.interfaces.Engine*) – The engine that governs this PlacementSet.
- **cell_type** (*bsb.cell_types.CellType*) – The cell type whose data is stored in the placement set.

Returns
A placement set

Return type
bsb.storage.interfaces.PlacementSet

abstract set_chunk_filter(*chunks*)

Should limit the scope of the placement set to the given chunks.

Parameters
chunks (*list[bsb.storage._chunks.Chunk]*) – List of chunks

abstract set_label_filter(*labels*)

Should limit the scope of the placement set to the given labels.

Parameters
labels (*list[str]*) – List of labels

abstract set_morphology_label_filter(*morphology_labels*)

Should limit the scope of the placement set to the given sub-cellular labels. The morphologies returned by *load_morphologies()* should return a filtered form of themselves if *as_filtered()* is called on them.

Parameters
morphology_labels (*list[str]*) – List of labels

property tag

The unique identifier of the placement set.

Returns
Unique identifier

Return type
str

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

Bases: object

class bsb.storage.interfaces.**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: Any

class bsb.storage.interfaces.**StoredFile**(*store*, *id*)

Bases: object

load()

property meta

property mtime

class bsb.storage.interfaces.**StoredMorphology**(*name*, *loader*, *meta*)

Bases: object

cached_load(*labels=None*)

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.

bsb.storage._chunks.**chunklist**(*chunks*) → List[*Chunk*]

Convert an iterable of chunk like objects to a sorted unique chunklist

class bsb.storage._chunks.**Chunk**(*chunk*, *chunk_size*)

Chunk identifier, consisting of chunk coordinates and size.

class bsb.storage.**NotSupported**(*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*, *main=0*, *missing_ok=True*)

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*)

copy(*new_root*)

Move the storage to a new root.

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_chunk_stats()

get_connectivity_set(*tag*)

Get a connection set.

Parameters
tag (*str*) – Connection tag

Returns
~bsb.storage.interfaces.ConnectivitySet

get_connectivity_sets()

Return a ConnectivitySet for the given type.

Parameters
type (CellType) – Specific cell type.

Returns
~bsb.storage.interfaces.ConnectivitySet

get_placement_set(*type*, *chunks=None*, *labels=None*, *morphology_labels=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.
- **labels** (*list[str]*) – Labels to filter the placement set by.

Returns
~bsb.storage.interfaces.PlacementSet

init(*scaffold*)
Initialize the storage to be ready for use by the specified scaffold.

init_placement(*scaffold*)

is_main_process()

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

read_only()

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
~bsb.storage.interfaces.ConnectivitySet

require_placement_set(*cell_type*)
Get a placement set.

Parameters
cell_type (CellType) – Connection cell_type

Returns
~bsb.storage.interfaces.PlacementSet

property root

property root_slug

store_active_config(*config*)
Store a configuration object in the storage.

supports(*feature*)

bsb.storage.**create_engine**(*name*, *root*, *comm*)

bsb.storage.**discover_engines**()
Get a dictionary of all available storage engines.

bsb.storage.**get_engine_node**(*engine_name*)

bsb.storage.**get_engines**()

bsb.storage.**open_storage**(*root*)

bsb.storage.**view_support**(*engine=None*)
Return which storage engines support which features.

class bsb.storage._files.**MorphologyOperationCallable**(**args*, ***kwargs*)
Hello

class bsb.storage._files.**OperationCallable**(**args*, ***kwargs*)

class bsb.storage._files.**CodeDependencyNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

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

file: *FileDependency*

get_node_name()

load_object()

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

class bsb.storage._files.**FileDependency**(*source: str | PathLike*, *file_store:* FileStore = *None*, *ext: str* = *None*, *cache=True*)

get_content(*check_store=True*)

get_meta(*check_store=True*)

get_stored_file()

provide_locally()

provide_stream()

should_update()

store_content(*content*, *encoding=None*, *meta=None*)

update(*force=False*)

property uri

class bsb.storage._files.**FileDependencyNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

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

get_node_name()

get_stored_file()

load_object()

provide_locally()

provide_stream()

scaffold: *Scaffold*

class bsb.storage._files.**FileScheme**

find(*file:* FileDependency)

get_content(*file:* FileDependency)

get_local_path(*file:* FileDependency)

get_meta(*file:* FileDependency)

provide_stream(*file:* FileDependency)

should_update(*file:* FileDependency, *stored_file*)

class bsb.storage._files.**MorphologyDependencyNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Configuration dependency node to load morphology files. The content of these files will be stored in bsb.morphologies.Morphology instances.

get_morphology_name()

Returns morphology name provided by the user or extract it from its file name.

Returns
Morphology name

Return type
str

get_node_name()

load_object(*parser=None*, *save=True*) → *Morphology*

name: str

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

parser: *MorphologyParser*

Name associated to the morphology. If not provided, the program will use the name of the file in which the morphology is stored.

pipeline: *cfglist*[*MorphologyOperation*]

queue(*pool*)

Add the loading of the current morphology to a job queue.

Parameters
pool (bsb.services.pool.JobPool) – Queue of jobs.

store_content(*content*, **args*, *encoding=None*, *meta=None*)

store_object(*morpho*, *hash_*)

Save a morphology into the circuit file under the name of this instance morphology.

Parameters

- **hash** (*str*) – Hash key to store as metadata with the morphology
- **morpho** (*bsb.morphologies.Morphology*) – Morphology to store

class bsb.storage._files.**MorphologyOperation**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

func: *MorphologyOperationCallable*

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

get_node_name()

class bsb.storage._files.**NeuroMorphoScheme**

create_session()

get_base_url()

get_meta(*file:* FileDependency)

get_nm_meta(*file:* FileDependency)

resolve_uri(*file:* FileDependency)

class bsb.storage._files.**NrrdDependencyNode**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Configuration dependency node to load NRRD files.

get_data()

get_header()

get_node_name()

load_object()

class bsb.storage._files.**Operation**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

func: *OperationCallable*

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

get_node_name()

parameters: dict[Any]

class bsb.storage._files.**UriScheme**

abstract find(*file:* FileDependency)

abstract get_content(*file:* FileDependency)

abstract get_local_path(*file:* FileDependency)

abstract get_meta(*file:* FileDependency)

abstract provide_stream(*file*)

abstract should_update(*file:* FileDependency, *stored_file*)

class bsb.storage._files.**UrlScheme**

create_session()

find(*file:* FileDependency)

get_base_url()

get_content(*file:* FileDependency)

get_local_path(*file:* FileDependency)

get_meta(*file:* FileDependency)

provide_stream(*file*)

resolve_uri(*file:* FileDependency)

should_update(*file:* FileDependency, *stored_file*)

27.1.2 Submodules

27.1.3 bsb.core module

class bsb.core.**ReportListener**(*scaffold*, *file*)

Bases: object

class bsb.core.**Scaffold**(*config=None*, *storage=None*, *clear=False*, *comm=None*)

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*, *fail_fast=True*)

Run reconstruction steps in the scaffold sequence to obtain a full network.

property configuration: *Configuration*

property connectivity

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

create_job_pool(*fail_fast=None*, *quiet=False*)

property files: *FileStore*

get_cell_types() → List[*CellType*]

Return a list of all cell types in the network.

get_config_diagram()

get_connectivity(*anywhere=None*, *presynaptic=None*, *postsynaptic=None*, *skip=None*, *only=None*) → List[*ConnectivitySet*]

get_connectivity_set(*tag=None*, *pre=None*, *post=None*) → *ConnectivitySet*

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() → List[*ConnectivitySet*]

Return all connectivity sets from the output formatter.

Parameters

tag (*str*) – Unique identifier of the connectivity set in the output formatter

Returns

All connectivity sets

get_dependency_pipelines()

get_placement(*cell_types=None*, *skip=None*, *only=None*) → List[*PlacementStrategy*]

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*, *labels=None*, *morphology_labels=None*) → *PlacementSet*

Return a cell type's placement set from the output formatter.

Parameters

- **tag** (*str*) – Unique identifier of the placement set in the storage
- **labels** (*list[str]*) – Labels to filter the placement set by.
- **morphology_labels** (*list[str]*) – Subcellular labels to apply to the morphologies.

Returns
A placement set

Return type
PlacementSet

get_placement_sets() → List[*PlacementSet*]

Return all of the placement sets present in the network.

Return type
List[PlacementSet]

get_simulation(*sim_name: str*) → *Simulation*

Retrieve the default single-instance adapter for a simulation.

get_storage_diagram()

is_main_process() → bool

is_worker_process() → bool

merge(*other*, *label=None*)

property morphologies: *MorphologyRepository*

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

property regions

register_listener(*listener*, *max_wait=None*)

remove_listener(*listener*)

require_connectivity_set(*pre*, *post*, *tag=None*) → *ConnectivitySet*

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(*hooks=None*, *fail_fast=None*, *pipelines=True*)

Run after placement hooks.

run_after_placement(*hooks=None*, *fail_fast=None*, *pipelines=True*)

Run after placement hooks.

run_connectivity(*strategies=None*, *fail_fast=True*, *pipelines=True*)

Run connection strategies.

run_pipelines(*fail_fast=True*, *pipelines=None*)

run_placement(*strategies=None*, *fail_fast=True*, *pipelines=True*)

Run placement strategies.

run_placement_strategy(*strategy*)

Run a single placement strategy.

run_simulation(*simulation_name: str*)

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: *Storage*

property storage_cfg

bsb.core.**from_storage**(*root*)

Load *core.Scaffold* from a storage object.

Parameters
root – Root (usually path) pointing to the storage object.

Returns
A network scaffold

Return type
Scaffold

27.1.4 bsb.cell_types module

Module for the CellType configuration node and its dependencies.

class bsb.cell_types.**CellType**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

Information on a population of cells.

clear()

Clear all the placement and connectivity data associated with this cell type.

clear_connections()

Clear all the connectivity data associated with this cell type. Any connectivity set that this cell type is a part of will be entirely removed.

clear_placement()

Clear all the placement data associated with this cell type. Connectivity data will remain, but be invalid.

entity

Whether this cell type is an entity type. Entity types don't have representations in space, but can still be connected and simulated.

get_morphologies()

Return the list of morphologies of this cell type.

Return type
List[StoredMorphology]

get_node_name()

get_placement()

Get the placement components this cell type is a part of.

get_placement_set(**args*, ***kwargs*)

Retrieve this cell type's placement data

Parameters
chunks (*List[bsb.storage._chunks.Chunk]*) – When given, restricts the placement data to these chunks.

property morphologies

name

Name of the cell type, equivalent to the key it occurs under in the configuration.

plotting

Plotting information about the cell type, such as color and labels.

scaffold: *Scaffold*

spatial

Spatial information about the cell type such as radius and density, and geometric or morphological information.

class bsb.cell_types.**PlacementIndications**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

count: int

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

count_ratio: float

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

density: float

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

density_key: *str*

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

density_ratio: *float*

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

geometry: *dict*

get_node_name()

morphologies: *cfglist*[*MorphologySelector*]

planar_density: *float*

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

radius: *float*

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

relative_to: *CellType*

scaffold: *Scaffold*

class bsb.cell_types.**Plotting**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: object

color

Color used to display this cell type in plots.

display_name

Label used to display this cell type in plots.

get_node_name()

opacity

Opacity used to display this cell type in plots.

scaffold: *Scaffold*

27.1.5 bsb.exceptions module

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

Bases: *ScaffoldError*

AdapterError exception

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

Bases: *GatewayError*

AllenApiError exception

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

Bases: *StorageError*

AttributeMissingError exception

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

Bases: *ConfigurationError*

BootError exception

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

Bases: *ScaffoldError*

CLIError exception

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

Bases: *ConfigurationError*

CastConfigurationError exception

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

Bases: *ConfigurationError*

CastError exception

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

Bases: *ConfigurationError*

CfgReferenceError exception

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

Bases: *PlacementError*

ChunkError exception

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

Bases: *MorphologyError*

CircularMorphologyError exception

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

Bases: *ScaffoldError*

ClassError exception

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

Bases: *DynamicClassError*

ClassMapMissingError exception

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

Bases: *ScaffoldError*

CodeImportError exception

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

Bases: *CLIError*

CommandError exception

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

Bases: *MorphologyError*

CompartmentError exception

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

Bases: *ScaffoldError*

CompilationError exception

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

Bases: *CLIError*

ConfigTemplateNotFoundError exception

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

Bases: *ScaffoldError*

ConfigurationError exception

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

Bases: *ConfigurationError*

ConfigurationFormatError exception

exception bsb.exceptions.**ConfigurationWarning**

Bases: *ScaffoldWarning*

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

Bases: *ScaffoldError*

ConnectivityError exception

exception bsb.exceptions.**ConnectivityWarning**

Bases: *ScaffoldWarning*

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

Bases: *PlacementError*

ContinuityError exception

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

Bases: *StorageError*

DataNotFoundError exception

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

Bases: *ScaffoldError*

DataNotProvidedError exception

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

Bases: *StorageError*

DatasetExistsError exception

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

Bases: *StorageError*

DatasetNotFoundError exception

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

Bases: *ScaffoldError*

DependencyError exception

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

Bases: *CastError*

DistributionCastError exception

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

Bases: *CompilationError*

DistributorError exception

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

Bases: *CLIError*

DryrunError exception

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

Bases: *ConfigurationError*

DynamicClassError exception

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

Bases: *DynamicClassError*

DynamicClassInheritanceError exception

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

Bases: *DynamicClassError*

DynamicObjectNotFoundError exception

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

Bases: *MorphologyError*

EmptyBranchError exception

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

Bases: *MorphologyError*

EmptySelectionError exception

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

Bases: *VoxelSetError*

EmptyVoxelSetError exception

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

Bases: *ConnectivityError*

ExternalSourceError exception

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

Bases: *ScaffoldError*

GatewayError exception

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

Bases: *ExternalSourceError*

IncompleteExternalMapError exception

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

Bases: *MorphologyError*

IncompleteMorphologyError exception

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

Bases: *ConfigurationError*

IndicatorError exception

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

Bases: *CLIError*

InputError exception

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

Bases: *DatasetNotFoundError*

IntersectionDataNotFoundError exception

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

Bases: *TypeHandlingError*

InvalidReferenceError exception

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

Bases: *JobPoolError*

JobCancelledError exception

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

Bases: *JobPoolError*

JobPoolContextError exception

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

Bases: *ScaffoldError*

JobPoolError exception

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

Bases: *JobPoolError*

JobSchedulingError exception

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

Bases: *TopologyError*

LayoutError exception

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

Bases: *StorageError*

MissingActiveConfigError exception

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

Bases: *MorphologyError*

MissingMorphologyError exception

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

Bases: *ExternalSourceError*

MissingSourceError exception

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

Bases: *MorphologyError*

MorphologyDataError exception

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

Bases: *ScaffoldError*

MorphologyError exception

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

Bases: *MorphologyError*

MorphologyRepositoryError exception

exception bsb.exceptions.**MorphologyWarning**

Bases: *ScaffoldWarning*

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

Bases: *CfgReferenceError*

NoReferenceAttributeSignal exception

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

Bases: *ScaffoldError*

NodeNotFoundError exception

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

Bases: *TypeHandlingError*

NoneReferenceError exception

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

Bases: *ScaffoldError*

OptionError exception

exception bsb.exceptions.**PackageRequirementWarning**

Bases: *ScaffoldWarning*

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

Bases: *PlacementError*

PackingError exception

exception bsb.exceptions.**PackingWarning**

Bases: *PlacementWarning*

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

Bases: *SimulationError*

ParameterError exception

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

Bases: *ScaffoldError*

ParserError exception

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

Bases: *ScaffoldError*

PlacementError exception

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

Bases: *PlacementError*

PlacementRelationError exception

exception bsb.exceptions.**PlacementWarning**

Bases: *ScaffoldWarning*

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

Bases: *ScaffoldError*

PluginError exception

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

Bases: *OptionError*

ReadOnlyOptionError exception

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

Bases: *CompilationError*

RedoError exception

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

Bases: *ParameterError*

ReificationError exception

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

Bases: *ConfigurationError*

RequirementError exception

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

Bases: DetailedException

ScaffoldError exception

exception bsb.exceptions.**ScaffoldWarning**

Bases: UserWarning

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

Bases: *ScaffoldError*

SelectorError exception

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

Bases: *ScaffoldError*

SimulationError exception

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

Bases: *ExternalSourceError*

SourceQualityError exception

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

Bases: *ScaffoldError*

StorageError exception

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

Bases: *ScaffoldError*

TopologyError exception

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

Bases: *ScaffoldError*

TreeError exception

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

Bases: *ScaffoldError*

TypeHandlingError exception

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

Bases: *CastError*

UnfitClassCastError exception

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

Bases: *ConfigurationError*

UnknownConfigAttrError exception

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

Bases: *ConnectivityError*

UnknownGIDError exception

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

Bases: *StorageError*

UnknownStorageEngineError exception

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

Bases: *TopologyError*

UnmanagedPartitionError exception

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

Bases: *CastError*

UnresolvedClassCastError exception

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

Bases: *ScaffoldError*

VoxelSetError exception

27.1.6 bsb.exceptions module

class bsb.mixins.**HasDependencies**

Mixin class to mark that this node may depend on other nodes.

abstract get_deps()

classmethod sort_deps(*objects*)

Orders a given dictionary of objects by the class's default mechanism and then apply the *after* attribute for further restrictions.

class bsb.mixins.**InvertedRoI**

This mixin inverts the perspective of the `get_region_of_interest` interface and lets you find presynaptic regions of interest for a postsynaptic chunk.

Usage:

..code-block:: python

class MyConnStrat(InvertedRoI, ConnectionStrategy):

def get_region_of_interest(post_chunk):
return [pre_chunk1, pre_chunk2]

queue(*pool*)

class bsb.mixins.**NotParallel**

27.1.7 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.

is_set(*slug*)

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: *OptionDescriptor*

Descriptor that retrieves its value from the given CLI command arguments.

slug = 'cli'

class bsb.option.**EnvOptionDescriptor**(**args*, *flag=False*)

Bases: *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: *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: *OptionDescriptor*

Descriptor that retrieves and sets its value from/to the *bsb.options* module.

is_set(*instance*)

slug = 'script'

27.1.8 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 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.

class bsb.options.**ProfilingOption**(*positional=False*)

Bases: *BsbOption*

Enables profiling.

cli

Descriptor that retrieves its value from the given CLI command arguments.

description = None

env

Descriptor that retrieves its value from the environment variables.

get_default()

Override to specify the default value of the option.

getter(*value*)

inverted_flag = False

is_flag = True

name = 'profiling'

positional = False

project

Descriptor that retrieves and stores values in the *pyproject.toml* file. Traverses up the filesystem tree until one is found.

readonly = False

script

Descriptor that retrieves and sets its value from/to the *bsb.options* module.

setter(*value*)

use_action = False

use_extend = False

class bsb.options.**VerbosityOption**(*positional=False*)

Bases: *BsbOption*

Set the verbosity of the package. Verbosity 0 is completely silent, 1 is default, 2 is verbose, 3 is progress and 4 is debug.

cli

Descriptor that retrieves its value from the given CLI command arguments.

description = None

env

Descriptor that retrieves its value from the environment variables.

get_default()

Override to specify the default value of the option.

getter(*value*)

inverted_flag = False

is_flag = False

name = 'verbosity'

positional = False

project

Descriptor that retrieves and stores values in the *pyproject.toml* file. Traverses up the filesystem tree until one is found.

readonly = False

script

Descriptor that retrieves and sets its value from/to the *bsb.options* module.

setter(*value*)

use_action = False

use_extend = False

bsb.options.**discover_options**()

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**(*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_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_option_descriptor**(*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_descriptors**()

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_option**(*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_option**(*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.

27.1.9 bsb.plugins module

Plugins module. Uses `pkg_resources` to detect installed plugins and loads them as categories.

bsb.plugins.**discover**(*category*)

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

27.1.10 bsb.postprocessing module

class bsb.postprocessing.**AfterConnectivityHook**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: ABC

get_node_name()

name: str

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

abstract postprocess()

queue(*pool*)

strategy

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

class bsb.postprocessing.**AfterPlacementHook**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: ABC

get_node_name()

name: *str*

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

abstract postprocess()

queue(*pool*)

strategy

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

class bsb.postprocessing.**BidirectionalContact**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *AfterConnectivityHook*

postprocess()

class bsb.postprocessing.**Relay**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *AfterConnectivityHook*

Replaces connections on a cell with the relayed connections to the connection targets of that cell. Not implemented yet.

cell_types

get_node_name()

postprocess()

class bsb.postprocessing.**SpoofDetails**(**args*, *_parent=None*, *_key=None*, ***kwargs*)

Bases: *AfterConnectivityHook*

Create fake morphological intersections between already connected non-detailed connection types.

casts = {'postsynaptic': <class 'str'>, 'presynaptic': <class 'str'>}

postprocess()

spoof_connections(*connection_type*, *connectivity_matrix*)

27.1.11 bsb.reporting module

bsb.reporting.**report**(**message*, *level=2*, *ongoing=False*, *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.

bsb.reporting.**warn**(*message*, *category=None*, *stacklevel=2*, *log_exc=None*)

Send a warning.

Parameters

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

27.1.12 bsb.trees module

Module for binary space partitioning, to facilitate optimal runtime complexity for n-point problems.

class bsb.trees.**BoxTree**(*boxes*)

Tree for fast lookup of repeat queries of axis aligned rhomboids.

class bsb.trees.**BoxTreeInterface**

Tree for fast lookup of queries of axis aligned rhomboids.

abstract query(*boxes*, *unique=False*)

Should return a generator that yields lists of intersecting IDs per query box if `unique=False`. If `unique=True`, yield a flat list of unique intersecting box IDs for all queried boxes.

class bsb.trees.**_BoxRTree**(*boxes*)

Tree for fast lookup of queries of axis aligned rhomboids using the Rtree package.

27.1.13 bsb.voxels module

class bsb.voxels.**BoxTree**(*boxes*)

Bases: *_BoxRTree*

Tree for fast lookup of repeat queries of axis aligned rhomboids.

class bsb.voxels.**VoxelData**(*data*, *keys=None*)

Bases: 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.**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*)

coordinates_of(*positions*)

copy()

crop(*ldc*, *mdc*)

crop_chunk(*chunk*)

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]

property data_keys

classmethod empty(*size=None*)

property equilateral

Whether all sides of all voxels have the same lengths.

Return type
bool

classmethod fill(*positions*, *voxel_size*, *unique=True*)

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

index_of(*positions*)

inside(*positions*)

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.

Return type
bool

resize(*size*)

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(*voxel_size*, *unique=False*)

unique()

property volume

27.1.14 Module contents

bsb-core is the backbone package contain the essential code of the BSB: A component framework for multiscale bottom-up neural modelling.

bsb-core needs to be installed alongside a bundle of desired bsb plugins, some of which are essential for *bsb-core* to function. First time users are recommended to install the *bsb* package instead.

CHAPTER

TWENTYEIGHT

INDEX

CHAPTER

TWENTYNINE

MODULE INDEX

CHAPTER

THIRTY

DEVELOPER INSTALLATION

To install:

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

Test your install with:

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

30.1 Releases

To release a new version:

```
bump-my-version bump pre_n
python -m build
twine upload dist/* --skip-existing
```

CHAPTER

THIRTYONE

DOCUMENTATION

Install the documentation dependencies of the BSB:

```
pip install -e .[docs]
```

Then navigate to the `docs` directory and run:

```
cd docs
make html
```

The output will be in the `/docs/_build` folder.

31.1 Conventions

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

CHAPTER

THIRTYTWO

SERVICES

The BSB provides some "services", which can be provided by a fallback system of providers. Usually they import a package, and if it isn't found, provide a sensible mock, or an object that errors on first use, so that the framework and any downstream packages can always import and use the service (if a mock is provided).

32.1 MPI

The MPI service provided by *bsb.services.MPI* is the `COMM_WORLD` `mpi4py.MPI.Comm` if `mpi4py` is available, otherwise it is an emulator that emulates a single node parallel context.

Error: If any environment variables are present that contain `MPI` in their name an error is raised, as execution in an actual MPI environment won't work without `mpi4py`.

32.2 MPILock

The MPILock service provides `mpilock`'s `WindowController` if it is available, or a mock that immediately and unconditionally acquires its lock and continues.

Note: Depends on the MPI service.

32.3 JobPool

The `JobPool` service allows you to `submit` `Jobs` and then `execute` them.

Note: Depends on the MPI service.

Most component types have a `queue` method that takes a job pool as an argument and lets them schedule their jobs.

The recommended way to open a job pool is to use the *create_job_pool()* context manager:

```
network = from_storage("example.hdf5")
with network.create_job_pool() as pool:
  if pool.is_main():
```

(continues on next page)

(continued from previous page)

```
    # Only the main node needs to schedule the jobs
    for component in network.placement.values():
      component.queue(pool)
  # But everyone needs to partake in the execute call
  pool.execute()
```

32.3.1 Scheduling

Pools can concurrently schedule the jobs on the main node, while executing them on worker nodes with the *schedule()* method:

```
network = from_storage("example.hdf5")
with network.create_job_pool() as pool:
  if pool.is_main():
    pool.schedule([*network.placement.values]())
  pool.execute()
```

Warning: Pass in topologically sorted arrays of nodes! Dependencies are only checked between the nodes, not the jobs, by checking for a `depends_on` attribute.

32.3.2 Listeners

On top of opening the job pool this also registers the appropriate listeners. Listeners listen to updates emitted by the job pool and can respond to changes, for example by printing them out to display the progress of the job pool:

```
_t = None
def report_time_elapsed(progress):
  global _t
  if progress.reason == PoolProgressReason.POOL_STATUS_CHANGE:
    if progress.status == PoolStatus.SCHEDULING:
      _t = time.time()
    elif progress.status == PoolStatus.CLOSING:
      print(f"Pool execution finished. {time.time()} seconds elapsed.")

with network.create_job_pool() as pool:
  pool.add_listener(report_time_elapsed)
  pool.submit(lambda scaffold: time.sleep(2))
  pool.execute()
  # Will print `Pool execution finished. 2 seconds elapsed.`
```

Listeners can also be context managers, and will enter and exit the same context as the JobPool.

CHAPTER

THIRTYTHREE

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.

33.1 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.

33.1.1 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"]
}
```

33.2 Categories

33.2.1 Configuration parsers

Category: `bsb.config.parsers`

Inherit from *ConfigurationParser*. 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.

33.2.2 Storage engines

Category: `bsb.storage.engines`

33.2.3 Simulator backends

Category: `bsb.simulation_backends`

33.2.4 Components

Category: `bsb.components`

Using component plugins, plugin authors can distribute reusable components. You can either eagerly load your components by loading the module, or lazy load them by registering a classmap extension:

```
[project.entry-points."bsb.components"]
my_components = "my_package.my_module:classmap"
```

And in `my_package/my_module.py` you can give a `classmap` dictionary that is keyed by the fully qualified class name of the components's classmaps you would like to extend. E.g., to add a placement strategy:

```
classmap = {
  "bsb.placement.strategy.PlacementStrategy": {
    "super_placement": "my_package.placement_module.SuperPlacementStrategy"
  }
}
```

A user can then use this placement strategy as follows:

```
strat = PlacementStrategy(strategy="super_placement", ...)
```

CHAPTER

THIRTYFOUR

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*.

34.1 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*.

34.2 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, Scaffold, 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.
```

34.3 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

34.4 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.

34.5 List of hooks

`__boot__`?

CHAPTER

THIRTYFIVE

DEVELOPER MODULES

35.1 bsb.services

Provides several services for optional dependencies.

bsb.services.**MPI = <bsb.services.mpi.MPIService object>**

MPI service

bsb.services.**MPILock = <bsb.services.mpilock.MPILockModule object>**

MPILock service

Service module. Register or access interfaces that may be provided, mocked or missing, but should always behave neatly on import.

exception bsb.services.**WorkflowError**(*_ExceptionGroup__message: str*, *_ExceptionGroup__exceptions: Sequence[_ExceptionT_co]*)

Bases: ExceptionGroup

bsb.services.**register_service**(*attr*, *provider*)

35.2 bsb.topology._layout module

Internal layout module. Makes sure regions and partitions don't mutate during layout.

class bsb.topology._layout.**Layout**(*data*, *owner=None*, *children=None*, *frozen=False*)

Bases: object

Container class for all types of partition data. The layout swaps the data of the partition with temporary layout associated data, and tries out experimental changes to the partition data, if the layout process fails, the original partition data is reinstated.

accept()

property children

copy()

property data

swap()

class bsb.topology._layout.**PartitionData**

Bases: ABC

The partition data is a class that stores the description of a partition for a partition. This allows the Partition interface to define mutating operations such as translate, rotate, scale; for a dry-run we only have to swap out the actual data with temporary data, and the mutation is prevented.

abstract copy()

class bsb.topology._layout.**RhomboidData**(*ldc*, *mdc*)

Bases: *PartitionData*

copy()

Copy this boundary to a new instance.

property depth

property dimensions

property height

property width

property x

property y

property z

bsb.topology._layout.**box_layout**(*ldc*, *mdc*)

35.3 bsb._util

Global internal utility module.

bsb._util.**assert_samelen**(**args*)

Assert that all input arguments have the same length.

bsb._util.**get_qualified_class_name**(*x*)

Return an object's module and class name

bsb._util.**ichain**(*iterable*, */*)

Alternative chain() constructor taking a single iterable argument that evaluates lazily.

bsb._util.**immutable**()

Decorator to mark a method as immutable, so that any calls to it return, and are performed on, a copy of the instance.

bsb._util.**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._util.**merge_dicts**(*a*, *b*)

Merge 2 dictionaries and their subdictionaries

bsb._util.**obj_str_insert**(*__str__*)

Decorator to insert the return value of __str__ into '<classname {returnvalue} at 0x...>'

bsb._util.**rotation_matrix_from_vectors**(*vec1*, *vec2*)

Find the rotation matrix that aligns vec1 to vec2

Parameters

- **vec1** – A 3d "source" vector
- **vec2** – A 3d "destination" vector

Return mat

A transform matrix (3x3) which when applied to vec1, aligns it with vec2.

bsb._util.**sanitize_ndarray**(*arr_input*, *shape*, *dtype=None*)

Convert an object to an ndarray and shape, avoiding to copy it wherever possible.

bsb._util.**suppress_stdout**()

Context manager that attempts to silence regular stdout and stderr. Some binary components may yet circumvene this if they access the underlying OS's stdout directly, like streaming to */dev/stdout*.

bsb._util.**unique**(*iter_: Iterable[Any]*)

Return a new list containing all the unique elements of an iterator

PYTHON MODULE INDEX

b

INDEX

Symbols

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