neuron morphology

Release 1.0.0.beta0

Jul 02, 2020

Contents

CHAPTER 1

Feature Extraction

1.1 Introduction

Morphological features are useful for investigating and clustering neuron morphologies. The Feature Extractor package is designed to allow flexible morpholigcal feature extraction from swc neuron reconstruction files and supplementary data. The default_feature set is a combination of L-measure and other features used by the Allen Institute.

1.2 Running Feature Extraction from the Command Line

The feature extractor module is an argschema module, which can be run from the command line:

```
feature_extractor --input_json path_to_inputs.json --output_json write_outputs_here.
↪json
```

Please see the schema file for usage details and options.

1.3 Running in Python/Notebooks

You can take advantage of all of the capabilities of Feature Extractor by running it in python and jupyter notebooks. By running in python and notebooks, you can easily add your own features, create different feature sets, and customize your feature extractor to meet your needs.

Here are two basic examples for running IVSCC and fMOST data:

- IVSCC example notebook
- fMOST example notebook

For a more detailed look at the feature extractor capabilites, checkout feature_extractor_example

CHAPTER 2

IVSCC Spatial Transformation

For feature extraction or visualization, we often need to apply a transformation to the space in which our reconstruction dwells. Some examples:

- **unshrink** : If a neuron is reconstructed from slice, the depth dimension may not scaled equivalently to the width and height dimensions, due to tissue shrinkage. In this case, the neuron must be rescaled along the depth dimension in order for features like compartment volume to be meaningful.
- **upright** : Images of single cortical neurons reconstructed in slice may be rotated arbitrarily. In order to visualize or calculate the symmetry of a neuron's apical dendrites, we must rotate the neuron so that the piaward direction is "up".

The `neuron_morphology` repository contains a set of utilities for calculating and applying such transformations. These utilities are ones that we, the Allen Institute, use for processing our in-vitro single cell characterization data (IVSCC, whitepaper here), but you may also find them handy if your data are similar.

2.1 Components

Here are the spatial transform components that we use for our IVSCC data. For each one, we've also included a link to the detailed input and output specification for that executable.

- pia_wm_streamlines (schema) : Given 2D linestrings describing the pia and white matter surfaces local to a neuron, calculate a cortical depth field, whose values are the depth between pia and white matter.
- upright_angle (schema) : Given an swc-formatted reconstruction and the outputs of pia_wm_streamlines, find the angle of rotation about the soma which will align the "y" axis of the reconstruction with the piaward direction.
- apply_affine_transform (schema) : Given a 3D affine transform and an swc-formatted reconstruction, produce a transformed reconstruction also in swc format.

2.2 Command-line invocation

Once you have installed `neuron_morphology`, you can run these utilities from the command line as you would any argschema module. Here is an example:

```
pia_wm_streamlines --input_json path_to_inputs.json --output_json write_outputs_here.
↪json
```

In this case, the contents of `path_to_inputs.json` might look like:

```
{
    "pia_path_str": "10.0,1.0,10.0,3.0,9.0,5.0",
    "wm_path_str": ".0,1.0,1.0,3.0,1.0,4.0"
}
```

Please see the schema file for more details and options.

2.3 Putting it all together

Most likely, you would like to run several of these components in sequence. Here is a jupyter notebook which demonstrates in depth how to go from a "raw" morphology and cortical boundaries to an upright-transformed morphology.

CHAPTER 3

Install

3.1 requirements

We support Python 3.7 on Linux, OSX, and Windows. Similar Python versions (e.g. 3.6, 3.8) will probably work, but we don't regularly test using those versions.

3.2 managing your Python environment

We recommend installing *neuron_morphology* into a managed Python environment. Having multiple isolated environments lets you install incompatible packages (or different versions of the same package!) simultaneously and prevents unexpected behavior by utilities that rely on the system Python installation.

Two popular tools for managing Python environments are anaconda and venv. The rest of this document assumes that you have created and activated an environment using one of these tools. Using anaconda, this looks like:

```
conda create -y --name environment-name python=3.6
conda activate environment-name
```

and using venv:

```
python -m venv path/to/environment
source path/to/environment/bin/activate
```

3.3 installing from github

If you want to install a specific branch, tag, or commit of *neuron_morphology*, you can do so using pip:

```
pip install git+https://github.com/alleninstitute/neuron_morphology@dev
```

The *dev* branch contains cutting-edge features that might not have been formally released yet. By installing this way, you can access those features.

3.4 installing for development

If you want to work on *neuron_morphology*, you should first clone the repository, then install it in editable mode so that you can easily test your changes:

```
git clone https://github.com/alleninstitute/neuron_morphology
cd neuron_morphology

conda install -c conda-forge fenics mshr # optional, these dependencies' pypi␣
↪packages don't work out of the box on all platforms
pip install -r requirements.txt -U
pip install -r test_requirements.txt -U

pip install -e .
```

3.5 installing from conda-forge [coming soon!]

To install using conda (, run

```
conda install -c conda-forge -y neuron_morphology
```

This method is preferred vs. pip, since some subpackages of *neuron_morphology* depend on 3rd party packages which don't pip install well on all major platforms. Note that this use of conda as a *package* manager does not require or depend on using conda as your *environment* manager

3.6 installing from pypy [coming soon!]

You can install the latest release from pypy by running:

```
pip install neuron_morphology
```

CHAPTER 4

neuron_morphology

4.1 neuron_morphology package

4.1.1 Subpackages

neuron_morphology.feature_extractor package

Submodules

neuron_morphology.feature_extractor.data module

class neuron_morphology.feature_extractor.data.**Data**(*morphology: neuron_morphology.morphology.Morphology*, ***other_things*)

Bases: object

neuron_morphology.feature_extractor.data.**get_morphology**(*data: Union[neuron_morphology.feature_extractor.data.D neuron_morphology.morphology.Morphology]*)

Decay a Data to a Morphology, leaving Morphologies untouched

neuron_morphology.feature_extractor.feature_extraction_run module

class neuron_morphology.feature_extractor.feature_extraction_run.**FeatureExtractionRun**(*data*)

Bases: object

Methods

extract(self)	For each selected feature, carry out calculation on this run's dataset.
select_features(self, features, only_marks, ...)	Choose features to calculated for this run on the basis of selected marks.
select_marks(self, marks, required_marks)	Choose marks for this run by validating a set of candidates against the data.
serialize(self)	Return a dictionary describing this run

extract (*self*)
For each selected feature, carry out calculation on this run's dataset.

Returns

self [This FeatureExtractionRun, with results updated]

select_features (*self, features: Collection[neuron_morphology.feature_extractor.marked_feature.MarkedFeature], only_marks: Union[AbstractSet[Type[neuron_morphology.feature_extractor.mark.Mark]], NoneType] = None*)
Choose features to calculated for this run on the basis of selected marks.

Parameters

features [Candidates features for selection]

only_marks [if provided, reject features not marked with marks in] this set

Returns

self [This FeatureExtractionRun, with selected_features updated]

select_marks (*self, marks: Collection[Type[neuron_morphology.feature_extractor.mark.Mark]], required_marks: AbstractSet[Type[neuron_morphology.feature_extractor.mark.Mark]] = frozenset()*)
Choose marks for this run by validating a set of candidates against the data.

Parameters

marks [candidate marks to be validated]

required_marks [if provided, raise an exception if any of these marks] do not validate successfully

Returns

self [This FeatureExtractionRun, with selected_marks updated]

serialize (*self*)
Return a dictionary describing this run

neuron_morphology.feature_extractor.feature_extractor module

class neuron_morphology.feature_extractor.feature_extractor.**FeatureExtractor** (*features: Sequence[Union[Call Any], neuron_morphology.fea = ()*)

Bases: object

Methods

extract(self, data, only_marks, ...)	Run the feature extractor for a single dataset
register_features(self, features, Any], ...)	Add a new feature to the list of options

extract (*self, data: neuron_morphology.feature_extractor.data.Data, only_marks: Union[AbstractSet[Type[neuron_morphology.feature_extractor.mark.Mark]], NoneType] = None, required_marks: AbstractSet[Type[neuron_morphology.feature_extractor.mark.Mark]] = frozenset()*) → neuron_morphology.feature_extractor.feature_extraction_run.FeatureExtractionRun

Run the feature extractor for a single dataset

Parameters

data [the dataset from which features will be calculated]

only_marks [if provided, reject marks not in this set]

required_marks [if provided, raise an exception if any of these marks] do not validate successfully

Returns

The calculated features, along with a record of the marks and features selected.

register_features (*self, features: Sequence[Union[Callable[[neuron_morphology.feature_extractor.data.Data], Any], neuron_morphology.feature_extractor.marked_feature.MarkedFeature, Mapping[Any, Union[Callable[[neuron_morphology.feature_extractor.data.Data], Any], neuron_morphology.feature_extractor.marked_feature.MarkedFeature]], Iterable[Union[Callable[[neuron_morphology.feature_extractor.data.Data], Any], neuron_morphology.feature_extractor.marked_feature.MarkedFeature]]]*)

Add a new feature to the list of options

Parameters

features [the features to be registered. If it is not already marked,] it will be registered with no marks

neuron_morphology.feature_extractor.feature_specialization module

class neuron_morphology.feature_extractor.feature_specialization.**AllNeuriteCompareSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types_to_compare': None}

marks = {<class 'neuron_morphology.feature_extractor.mark.AllNeuriteTypes'>}

name = 'all_neurites'

class neuron_morphology.feature_extractor.feature_specialization.**AllNeuriteSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.*

FeatureSpecialization

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types': None}

marks = {<class 'neuron_morphology.feature_extractor.mark.AllNeuriteTypes'>}

name = 'all_neurites'

class neuron_morphology.feature_extractor.feature_specialization.**ApicalDendriteCompareSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types_to_compare': [4]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresApical'>}

name = 'apical_dendrite'

class neuron_morphology.feature_extractor.feature_specialization.**ApicalDendriteSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types': [4]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresApical'>}

name = 'apical_dendrite'

class neuron_morphology.feature_extractor.feature_specialization.**AxonCompareSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types_to_compare': [2]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresAxon'>}

name = 'axon'

class neuron_morphology.feature_extractor.feature_specialization.**AxonSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types': [2]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresAxon'>}

name = 'axon'

class neuron_morphology.feature_extractor.feature_specialization.**BasalDendriteCompareSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types_to_compare': [3]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresBasal'>}

name = 'basal_dendrite'

class neuron_morphology.feature_extractor.feature_specialization.**BasalDendriteSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types': [3]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresBasal'>}

name = 'basal_dendrite'

class neuron_morphology.feature_extractor.feature_specialization.**DendriteCompareSpec**
Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types_to_compare': [4, 3]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresDendrite'>}

name = 'dendrite'

class neuron_morphology.feature_extractor.feature_specialization.**DendriteSpec**
Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'node_types': [4, 3]}

marks = {<class 'neuron_morphology.feature_extractor.mark.RequiresDendrite'>}

name = 'dendrite'

class neuron_morphology.feature_extractor.feature_specialization.**FeatureSpecialization**
Bases: object

Attributes

kwargs

marks

name

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

classmethod factory(*name: str, marks: Set[Type[neuron_morphology.feature_extractor.mark.Mark]], kwargs: Dict[str, Any], display_name: Union[str, NoneType] = None*) → Type[~Fs]
A utility for quickly generating feature specializations

Parameters

name [The name of the generated class. If display_name is not] provided, this will also be used as the name attribute of the generated class

marks [the marks which this specialization implies.]

kwargs [the keyword argument values defining this specialization]

display_name [if provided, the name attribute of the generated] specialization.

Returns

A generated FeatureSpecialization subclass

kwargs

marks

name

neuron_morphology.feature_extractor.feature_writer module

neuron_morphology.feature_extractor.mark module

class neuron_morphology.feature_extractor.mark.**AllNeuriteTypes**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates features that are calculated for all neurite types.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**BifurcationFeatures**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates a feature calculated on bifurcations.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**CompartmentFeatures**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates a feature calculated on compartments.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**Geometric**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates features that change depending on coordinate frame.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**Intrinsic**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates intrinsic features that don't rely on a ccf or scale.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**Mark**
Bases: object

A tag, intended for use in feature selection.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod factory(*name: str*) → Type[~Mr]

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**NeuriteTypeComparison**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates a feature that is a comparison between neurite types.

Function should be decorated with the appropriate RequiresType marks

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**RequiresApical**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates that these features require an apical dendrite.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresAxon**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates that these features require an axon.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresBasal**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates that these features require a basal dendrite.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresDendrite**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

This feature can only be calculated for neurons with at least one dendrite node

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool

Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresLayerAnnotations**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Methods

validate(data)	Checks whether each node in the data's morphology is annotated with a cortical layer.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Checks whether each node in the data's morphology is annotated with a cortical layer. Returns False if any are missing.

class neuron_morphology.feature_extractor.mark.**RequiresLayeredPointDepths**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

This feature can only be calculated if (cortical) points are annotated with a collection of within-layer depths. See features.layer.layered_point_depths for more information.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresRadii**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

This feature can only be calculated if the radii of nodes are annotated.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresReferenceLayerDepths**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

This feature can only be calculated if a referenceset of average depths for cortical layers is provided. See features.layer.reference_layer_depths for more information.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresRegularPointSpacing**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

This features can only be (meaningfully) calculated if the points (e.g. node positions) on which it is based are resampled to have regular spacing.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

class neuron_morphology.feature_extractor.mark.**RequiresRelativeSomaDepth**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

This feature can only be calculated for relative soma depth

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresRoot**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates that this features require a root. Warns if the root is not unique

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**RequiresSoma**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates that these features require a soma.

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

classmethod validate(*data: neuron_morphology.feature_extractor.data.Data*) → bool
Determine if this feature is calculable from the provided data.

Parameters

data [Data from a single morphological reconstruction]

Returns

whether marked features can be calculated from these data

class neuron_morphology.feature_extractor.mark.**TipFeatures**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates a feature calculated on tips (leaf nodes).

Methods

validate(data)	Determine if this feature is calculable from the provided data.

factory

neuron_morphology.feature_extractor.mark.**check_nodes_have_key**(*data: neuron_morphology.feature_extractor.data.Da*
key: str) → bool
Checks whether each node in a morphology is annotated with some key.

neuron_morphology.feature_extractor.marked_feature module

class neuron_morphology.feature_extractor.marked_feature.**MarkedFeature**(*marks: Set[Type[neuron_morpholog feature: Union[Callable[[neuron_mo Any], neuron_morphology.feature_extr name: Optional[str] = None, preserve_marks: bool = True*)

Bases: object

Attributes

feature

marks

name

Methods

__call__(self, *args, **kwargs)	Execute the underlying feature, passing along all arguments
add_mark(self, mark)	Assign an additional mark to this feature
deepcopy(self, **kwargs)	Make a deep copy of this marked feature
ensure(feature)	If a function is not a MarkedFeature, convert it.
partial(self, *args, **kwargs)	Fix one or more parameters on this feature's callable
specialize(self, option)	Apply a specialization option to this feature.

add_mark(*self, mark: Type[neuron_morphology.feature_extractor.mark.Mark]*)
Assign an additional mark to this feature

deepcopy(*self, **kwargs*)
Make a deep copy of this marked feature

classmethod ensure(*feature: 'Feature'*) → ~M
If a function is not a MarkedFeature, convert it.

Parameters

feature [the feature to be converted]

Returns

Either a marked feature generated from the input, or the input marked feature.

feature

marks

name

partial(*self*, **args*, ***kwargs*)
Fix one or more parameters on this feature's callable

specialize(*self, option: Type[neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization]*)
Apply a specialization option to this feature. This binds parameters on the feature's __call__ method, sets 0 or more additional marks, and namespaces the feature's name.

Parameters

option [The specialization option with which to specialize this] feature.

Returns

a deep copy of this feature with updated callable, marks and name

neuron_morphology.feature_extractor.marked_feature.**marked**(*mark: Type[neuron_morphology.feature_extractor.mari*
Decorator for adding a mark to a function.

Parameters

mark [the mark to be applied]

Examples

@marked(RequiresA) @marked(RequiresB) def some_feature_requiring_a_and_b(. . .):

. . .

neuron_morphology.feature_extractor.marked_feature.**nested_specialize**(*feature: Union[Callable[[neuron_morph Any], neuron_morphology.feature_extract specialization_sets: List[Set[Type[neuron_morpholo* → Dict[str, neuron_morphology.feature_extract
Apply specializations hierarchically to a base feature. Generating a new collection of specialized features.

Parameters

feature [will be used as a basis for specialization]

specialization_sets [each element describes a set of specialization] options. The output will have one specialization for each element of the cartesian product of these sets.

Returns

A dictionary mapping namespaced feature names to specialized features.

Notes

Specializations are applied from the start of the specialization_sets to the end. This means that the generated names are structures like:

"last_spec.middle_spec.first_spec.base_feature_name"

neuron_morphology.feature_extractor.marked_feature.**specialize**(*feature: Union[Callable[[neuron_morphology.feat Any], neuron_morphology.feature_extractor.marked specialization_set: Set[Type[neuron_morphology.feature_extr* → Dict[str, neuron_morphology.feature_extractor.marked

Bind some of a feature's keyword arguments, using provided specialization options.

Parameters

feature [will be used as a basis for specialization]

specialization_set [each element defines a particular specialization (i.e] a set of keyword argument values and marks) to be applied to the feature

Returns

A dictionary mapping (namespaced) feature names to specialized features.

Note that names are formatted as "specialization_name.base_feature_name"

neuron_morphology.feature_extractor.run_feature_extraction module

neuron_morphology.feature_extractor.run_feature_extraction.**hydrate_parameters**(*parameters: Dict[str, Any]*) → Dict[str, Any]

Resolve argued feature parameters to a format comprehensible by the features. e.g. loading data from a path.

Parameters

parameters [to be hydrated]

Returns

The hydrated parameters

neuron_morphology.feature_extractor.run_feature_extraction.**resolve_reference_layer_depths**(*k n l a*

Given either the name of a well known depths set or a set of names and corresponding boundaries, produce a ReferenceLayerDepths

Parameters

key [of a well known reference layer]

names [the names of each layer in a custom sequence]

boundaries [the upper and lower depths of each layer in a custom sequence]

Returns

the requested reference layer depths

neuron_morphology.feature_extractor.run_feature_extraction.**run_feature_extraction**(*reconstructio Dict[str, Any], feature_set: str, only_marks: List[str], required_mark List[str], global_para Dict[str, Any]*) → Tuple[str, Dict]

Run feature extraction for a single reconstruction.

Parameters

reconstruction_spec [a dictionary specifying a reconstruction. Must] have an swc_path.

feature_set [names the set of features for which calculation will be] attempted

only_marks [names marks to which calculation will be restricted]

required_marks [raise an exception if these named marks fail validation]

global_parameter_spec [a dictionary specifying cross-reconstruction] parameters

Returns

identifier [a label for this reconstruction]

A dict with keys: results - a dict, mapping features to calculated values selected_marks - the set of marks that passed validation selected features - the set of features for which calculation was

attempted

neuron_morphology.feature_extractor.run_feature_extraction.**setup_data**(*reconstruction: Dict[str, Any], global_parameters: Dict[str, Any]*) → Tuple[str, neuron_morphology.feature_extra

Construct a Data for extracting features from a single reconstruction.

Parameters

reconstruction [The reconstruction to be setup. Must specify an swc_path]

global_parameters [any cross-reconstruction feature parameters]

Returns

identifier [a label for this reconstruction]

data suitable for feature extraction

neuron_morphology.feature_extractor.utilities module

A collection of miscellaneous tools used by the feature extractor

neuron_morphology.feature_extractor.utilities.**unnest**(*inputs: Dict[str, Any]*, *_prefix=''*) → Dict[str, Any]

Convert nested dictionaries (with string keys) to a dot-notation flat dictionary.

inputs: The dictionary to unnest. Must have all string keys _prefix : Used during recursion to build up a dot-notation prefix. Don't

argue this yourself!

Returns

a flattened dictionary

Module contents

neuron_morphology.features package

Subpackages

neuron_morphology.features.branching package

Submodules

neuron_morphology.features.branching.bifurcations module

neuron_morphology.features.branching.bifurcations.**calculate_outer_bifs**(*morphology: neuron_morphology.morphology soma: Dict, node_types: Union[List[int], NoneType]*) → int

Counts the number of bifurcation points beyond the a sphere with 1/2 the radius from the soma to the most distant point in the morphology, with that sphere centered at the soma.

Parameters

morphology: Describes the structure of a neuron

soma: Must have keys “x”, “y”, and “z”, describing the position of this morphology’s soma in

node_types: Restrict included nodes to these types. See neuron_morphology.constants for avaiable node types.

Returns

the number of bifurcations

Module contents

neuron_morphology.features.layer package

Submodules

neuron_morphology.features.layer.layer_histogram module

class neuron_morphology.features.layer.layer_histogram.**EarthMoversDistanceInterpretation**

Bases: enum.Enum

Describes how to understand an earth mover’s distance result. This is useful in the case that one or both histograms are all 0.

BothEmpty = 2

BothPresent = 0

OneEmpty = 1

class neuron_morphology.features.layer.layer_histogram.**EarthMoversDistanceResult**

Bases: tuple

The result of comparing two histograms using earth mover’s distance

Attributes

interpretation Alias for field number 1

result Alias for field number 0

Methods

count(self, value, /)	Return number of occurrences of value.
index(self, value[, start, stop])	Return first index of value.

to_dict_human_readable	

interpretation
Alias for field number 1

result
Alias for field number 0

to_dict_human_readable(*self*)

class neuron_morphology.features.layer.layer_histogram.**LayerHistogram**
Bases: tuple

The results of calculating a within-layer depth histogram of points within some cortical layer.

Attributes

bin_edges Alias for field number 1

counts Alias for field number 0

Methods

count(self, value, /)	Return number of occurrences of value.
index(self, value[, start, stop])	Return first index of value.

bin_edges
Alias for field number 1

counts
Alias for field number 0

neuron_morphology.features.layer.layer_histogram.**ensure_layers**(*layers*)
Make sure the argued layer array is a tuple

neuron_morphology.features.layer.layer_histogram.**ensure_node_types**(*node_types*)
Make sure the argued node types are a tuple

neuron_morphology.features.layer.layer_histogram.**ensure_tuple**(*inputs: Any*, *item_type: Type*, *if_none: Union[str, Tuple] = 'raise'*) → Tuple

Try to smartly coerce inputs to a tuple.

Parameters

inputs [the data to be coerced]

item_type [which type do/should the elements of the tuple have?]

if_none [if the inputs are none, return this value. If the value is] "raise", instead raise an exception

Returns

the coerced inputs

neuron_morphology.features.layer.layer_histogram.**histogram_earth_movers_distance**(*from_hist: numpy.ndarra*
to_hist: numpy.ndarra
→ neu-
ron_morpholo

Calculate the earth mover's distance between to histograms, normalizing each. If one histogram is empty, return the sum of the other and a flag. If both are empty, return 0 a and a flag.

Parameters

from_hist [distance is calculated between (the normalized form of) this] histogram and to_hist. The result is symmetric.

to_hist [distance is calculated between (the normalized form of) this] histogram and from_hist

Returns

The distance between input histograms, along with an enum indicating

whether one or both of the histograms was all 0.

neuron_morphology.features.layer.layer_histogram.**normalized_depth_histogram_within_layer**(*po*
nu
lo
ca
nu
lo
ca
nu
re
er
en
ne
ro
bi
flo
→
nu

Calculates a histogram of node depths within a single (cortical) layer. Uses reference information about layer boundaries to normalize these depths for cross-reconstruction comparison.

Parameters

depths [Each item corresponds to a point of interest (such as a node] in a morphological reconstruction). Values are the depths of these points of interest from the pia surface.

local_layer_pia_side_depths [Each item corresponds to a point of interest.] Values are the depth of the intersection point between a path of steepest descent from the pia surface to the point of interest and the upper surface of the layer.

local_layer_wm_side_depths [Each item corresponds to a point of interest.] Values are the depth of the intersection point between the layer's lower boundary and the path described above.

reference_layer_depths [Used to provide normalized depths suitable] for comparison across reconstructions. Should provide a generic equivalent of local layer depths for a population or reference space.

bin_size [The width of each bin, in terms of depths from pia in the] reference space. Provide only one of bin_edges or bin_size.

Returns

A numpy array listing for each depth bin the number of nodes falling within that bin.

Notes

This function relies on the notion of a steepest descent path through cortex, but is agnostic to the method used to obtain such a path and to features of the path (e.g. whether it is allowed to curve). Rather the caller must ensure that all depths have been calculated according to a consistent scheme.

neuron_morphology.features.layer.layer_histogram.**normalized_depth_histograms_across_layers**

A helper function for running cortical depth histograms across multiple layers.

Parameters

data [must have reference_layer_depths and layered_point_depths]

point_types [calculate histograms for points labeled with these types]

only_layers [exclude other layers from this calculation]

bin_size [the size of each depth bin. Default is appropriate if the units] are microns.

neuron_morphology.features.layer.layered_point_depths module

class neuron_morphology.features.layer.layered_point_depths.**LayeredPointDepths**(*ids: Sequence[T_co], layer_name: Sequence[str], depth: Sequence[T_co], local_layer_pia_si Sequence[T_co], local_layer_wm_si Sequence[T_co], point_type: Sequence[T_co]*)

Bases: object

Methods

from_csv	
from_dataframe	
from_hdf5	
read	
to_csv	

DF_COLS = {'depth', 'ids', 'layer_name', 'local_layer_pia_side_depth', 'local_layer_wm

classmethod from_csv(*path: str*)

classmethod from_dataframe(*df: pandas.core.frame.DataFrame*)

classmethod from_hdf5(*path: str*)

classmethod read(*path: str*)

to_csv(*self*, *path: str*)

neuron_morphology.features.layer.reference_layer_depths module

class neuron_morphology.features.layer.reference_layer_depths.**ReferenceLayerDepths**

Bases: tuple

Reference (e.g. average across specimens and regions) depths of cortical layer boundaries. Depths are given from pia. Units are not specified, but the user should ensure they are consistent with other positional and size units (e.g. node positions and radii, point depths). Several features in this package specify defaults in microns; if you provide reference layer depths in other units, you should review features which use these depths and ensure that any default values agree with your units.

Attributes

pia_side [the (average) depth of the upper surface of the layer]

wm_side [the (average) depth of the lower (closer to white matter) surface] of the layer

scale [if True, these depths are taken as describing the upper and lower] surfaces of a real feature of the data. If False, one or both of them is taken to represent a user-selected boundary. In the latter case, features such as the layer histograms will not attempt to rescale point depths based on observed local layer thicknesses.

Methods

count(self, value, /)	Return number of occurrences of value.
index(self, value[, start, stop])	Return first index of value.
sequential(names, boundaries[, last_is_scale])	A utility for constructing multiple ordered reference layer depths without intervening space.

pia_side

Alias for field number 0

scale

Alias for field number 2

classmethod sequential(*names: Sequence[str], boundaries: Sequence[float], last_is_scale=False*)
A utility for constructing multiple ordered reference layer depths without intervening space.

Parameters

names [The name of each layer]

boundaries [The pia and wm side depth of each layer. Should be a flat] sequence that has 1 more element than names.

last_is_scale [If True, the last boundary will be interpreted as a] true anatomical boundary. If false, as an arbitrary cutoff.

thickness

wm_side
Alias for field number 1

Module contents

neuron_morphology.features.statistics package

Submodules

neuron_morphology.features.statistics.coordinates module

class neuron_morphology.features.statistics.coordinates.**BifurcationSpec**
Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'coord_type': <COORD_TYPE.BIFURCATION: 2>}

marks = {<class 'neuron_morphology.feature_extractor.mark.BifurcationFeatures'>}

name = 'bifurcation'

class neuron_morphology.features.statistics.coordinates.**COORD_TYPE**
Bases: enum.Enum

An enumeration.

BIFURCATION = 2

COMPARTMENT = 1

NODE = 0

TIP = 3

get_coordinates(*self, morphology, node_types: Union[List[int], NoneType] = None*)

class neuron_morphology.features.statistics.coordinates.**CompartmentSpec**
Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'coord_type': <COORD_TYPE.COMPARTMENT: 1>}

marks = {<class 'neuron_morphology.feature_extractor.mark.CompartmentFeatures'>}

name = 'compartment'

class neuron_morphology.features.statistics.coordinates.**NodeSpec**
Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'coord_type': <COORD_TYPE.NODE: 0>}

marks = {}

name = 'node'

class neuron_morphology.features.statistics.coordinates.**TipSpec**
Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

Methods

factory(name, marks, kwargs, Any], ...)	A utility for quickly generating feature specializations

kwargs = {'coord_type': <COORD_TYPE.TIP: 3>}

marks = {<class 'neuron_morphology.feature_extractor.mark.TipFeatures'>}

name = 'tip'

neuron_morphology.features.statistics.moments module

neuron_morphology.features.statistics.overlap module

neuron_morphology.features.statistics.overlap.**calculate_coordinate_overlap**(*coordinates_a*, *coordinates_b*, *dimension: int = 1*)

Return the % of coordinates_a that are above, overlaping, and below coordinates_b, and the same for b over a

Parameters

coordinates_a: 2d array-like with x, y, z cols

coordinates_b: 2d array-like with x, y, z cols

dimension: dimension to compare (0, 1, 2 for x, y, z), default 1 (y)

Returns

dict: a_above_b, a_overlap_b, a_below_b, or -1's if coordinates_b is empty

neuron_morphology.features.statistics.overlap.**calculate_coordinate_overlap_from_min_max**(*coo nun min floa max floa di- mer sio int = 1*)

Return the % of coordinates that are above the max, between, or below the min

Parameters

coordinates: np.ndarray with x, y, z columns

minv: min to check against

maxv: max to check against

dimension: dimension to compare (0, 1, 2 for x, y, z), default 1 (y)

Module contents

Submodules

neuron_morphology.features.default_features module

neuron_morphology.features.dimension module

neuron_morphology.features.intrinsic module

neuron_morphology.features.intrinsic.**calculate_branches_from_root**(*morphology*, *root*, *node_types=None*)

Calculate the number of branches of a specific neuron type in a morphology. A branch is defined as being between two bifurcations or between a bifurcation and a tip if a node has three or more children, it is treated as succesive bifurcations, e.g a trifurcation: _/_/__ creates 4 branches since the branch between the two bifurcations counts

Parameters

morphology: a morphology object

root: the root node to traverse from

node_types: a list of node types (see neuron_morphology constants)

neuron_morphology.features.intrinsic.**calculate_max_branch_order_from_root**(*morphology*, *root*, *node_types=None*)

Calculate the greatest number of branches encountered among all directed paths from the morphology's root to its leaves. A branch is defined as a root->leaf ordered path for which:

1. **the first node on the path is either**
   a. a bifurcation (has > 1 children)
   b. the root node
2. **the last node on the path is either**
   a. a bifurcation
   b. a leaf node (has 0 children)

Parameters

morphology: the reconstruction whose max branch order will be calculated

root: treat this node as root

node_types: If not None, consider only root->leaf paths whose leaf nodes are among these types (see neuron_morphology constants)

Returns

The greatest branch count encountered among all considered root->leaf

paths

neuron_morphology.features.intrinsic.**calculate_mean_fragmentation_from_root**(*morphology*, *root*, *node_types=None*)

Calculate the mean fragmentation from a root in a morphology. Mean fragmentation is the number of compartments over the number of branches. A branch is defined as being between two bifurcations or between a bifurcation and a tip if a node has three or more children, it is treated as succesive bifurcations, e.g a trifurcation: _/_/__ creates 4 branches since the branch between the two bifurcations counts

Parameters

morphology: a morphology object

root: the root node to traverse from

node_types: a list of node types (see neuron_morphology constants)

neuron_morphology.features.intrinsic.**child_ids_by_type**(*node_id*, *morphology*, *node_types=None*)

Helper function for the traversal functions

neuron_morphology.features.path module

neuron_morphology.features.path.**calculate_max_path_distance**(*morphology*, *root*, *node_types=None*)

Helper for max_path_distance. See below for more information.

neuron_morphology.features.path.**calculate_mean_contraction**(*morphology*, *root=None*, *node_types=None*)

See mean_contraction

neuron_morphology.features.size module

neuron_morphology.features.size.**parent_daughter_ratio_visitor**(*node: Dict[str, Any]*, *morphology: neuron_morphology.morphology.Morphology*, *counters: Dict[str, Union[int, float]]*, *node_types: Union[List[int], NoneType] = None*)

Calculates for a single node the ratio of the node's parent's radius to the node's radius. Stores these values in a provided dictionary.

Parameters

node [The node under consideration]

morphology [The reconstruction to which this node belongs]

counters [a dictionary used for storing running ratio totals and counts.]

node_types [skip nodes not of one of these types]

Notes

see mean_parent_daughter_ratio for usage

neuron_morphology.features.soma module

Module contents

neuron_morphology.layered_point_depths package

Module contents

neuron_morphology.pipeline package

Submodules

neuron_morphology.pipeline.post_data_to_s3 module

Module contents

neuron_morphology.snap_polygons package

Submodules

neuron_morphology.snap_polygons.bounding_box module

Contains a simple utility for representing a "growable" 2D rectangle

class neuron_morphology.snap_polygons.bounding_box.**BoundingBox**(*horigin: float*, *vorigin: float*, *hextent: float*, *vextent: float*)

Bases: object

Represents the bounds of a set of 2D objects

Parameters

vorigin, horigin [the near corner of the box]

vextent, hextent [the far corner of the box]

Attributes

aspect_ratio Width / height ratio of the box.

coordinates Origin and extent coordinates.

extent Coordinates of the box's extent (opposite corner from the origin).

height Vertical side length of the box.

hextent

horigin

origin Coordinates of the box's origin.

vextent

vorigin

width Horizontal side length of the box.

Methods

copy(self)	Duplicates this bounding box
round(self, inplace, origin_via, ...)	Round the coordinates of this box
transform(self, transform, float], ...)	Apply a transform to this box
update(self, horigin, vorigin, hextent, vextent)	Potentially enlarges this box.

aspect_ratio
Width / height ratio of the box.

coordinates
Origin and extent coordinates.

copy(*self*) → 'BoundingBox'
Duplicates this bounding box

Returns

A copy of this object.

extent
Coordinates of the box's extent (opposite corner from the origin).

height
Vertical side length of the box.

hextent

horigin

origin
Coordinates of the box's origin.

round(*self, inplace: bool = False, origin_via: Callable[[float], float] = <function around at 0x7f0938050d30>, extent_via: Callable[[float], float] = <function around at 0x7f0938050d30>*)
Round the coordinates of this box

Parameters

inplace [If True, round the coordinates of this object]

origin_via [method to use when rounding the origin]

extent_via [method to use when rounding the extent]

Returns

The rounded box (potentially self.

transform(*self, transform: Callable[[float, float], Tuple[float, float]], inplace: bool = False*) → 'BoundingBox'
Apply a transform to this box

Parameters

transform [A callable which maps (vertical, horizontal) coordinates to] new (vertical, horizontal) coordinates.

inplace [if True, apply the transform to this object]

Returns

The transformed box (potentially self)

update(*self*, *horigin: float*, *vorigin: float*, *hextent: float*, *vextent: float*)
Potentially enlarges this box.

Parameters

As to the constructor of BoundingBox. The new shape of this box is the smallest box enclosing both this and the inputs.

vextent

vorigin

width
Horizontal side length of the box.

neuron_morphology.snap_polygons.cortex_surfaces module

This module contains utilities for processing cortical surface drawings. In general we take these as given (they even take precedence of e.g. the upper and lower surfaces of layers 1 and 6b for instance), but some drawings pose resolvable problems.

The main such problem occurs when cortical layer drawings extend far from the layer drawings. Extrapolating layer drawings into this space is dangerous and not very useful (only the drawings near the cell are useful downstream). The solution implemented here is to cut out a segment of each surface whose endpoints are sufficiently close to the layer drawings and discard the rest.

neuron_morphology.snap_polygons.cortex_surfaces.**find_transition**(*unmet: shapely.geometry.point.Point, met: shapely.geometry.point.Point, condition: Callable[[shapely.geometry.point.Point, bool], iterations: int)* → shapely.geometry.point.Point

Given two points in space, one of which meets a condition, locate the position along a line segment between these points where the condition becomes true.

Parameters

unmet [a point at which the condition is not met]

met [a point at which the condition is met]

condition [used to evaluate intermediate points]

iterations [refine this many times]

Returns

A point along the input segment at which the condition is met.

Notes

No such transition point is required to exist. In that case, this function will find an arbitrary condition-meeting point along the segment. For our use case, this misbehavior is tolerable because an exact transition point is not required.

neuron_morphology.snap_polygons.cortex_surfaces.**first_met**(*coords: Sequence[Union[shapely.geometry.point.Point, Tuple]], condition: Callable[[shapely.geometry.point.Point], bool], iterations: int*) → Tuple[int, shapely.geometry.point.Point]

Locate the first point along a coordinate sequence at which a condition is met.

Parameters

coords [sequence to evaluate]

condition [used to evaluate points]

iterations [how many times to refine the transition point.]

Returns

The index and value of the transition point.

neuron_morphology.snap_polygons.cortex_surfaces.**remove_duplicates**(*coords: Sequence[shapely.geometry.point.Poin* → Sequence[shapely.geometry.point.Poin

Remove duplicate points from a coordinate sequence.

Parameters

coords [sequence with potential duplicates]

Returns

list of coordinates with duplicates removed

neuron_morphology.snap_polygons.cortex_surfaces.**trim_coords**(*coords: Sequence[Union[shapely.geometry.point.Point, Tuple]], condition: Callable[[shapely.geometry.point.Point], bool], iterations: int*)

Find the longest subinterval of a coordinate sequence whose endpoints meet some condition.

Parameters

coords [sequence to trim]

condition [used to evaluate points]

iterations [how many times to refine the endpoints.]

Returns

Trimmed sequence

neuron_morphology.snap_polygons.cortex_surfaces.**trim_to_close**(*geometry: shapely.geometry.base.BaseGeometry, threshold: float, linestring: Union[str, Sequence[Sequence[float]], shapely.geometry.linestring.LineString], iterations: int = 10*) → shapely.geometry.linestring.LineString

Find the longest segment of a linestring whose endpoints are within a specified distance of a geometry.

Parameters

geometry [Acceptable distances are defined as extending from this object.]

threshold [Acceptable distances are less than or equal to this value]

linestring [to be trimmed (not in place)]

iterations [Use this many iterations to refine the endpoints of the] linestring

Returns

a trimmed copy of the input linestring

neuron_morphology.snap_polygons.geometries module

A collection of utilities used by snap polygons to manipulate shapely objects.

class neuron_morphology.snap_polygons.geometries.**Geometries**

Bases: object

A collection of polygons and lines

Attributes

close_bounds The smallest bounding box enclosing these geometries.

default_multipolygon_resolver By default, multiple polygons resulting from operations on these

default_multisurface_resolver By default, multiple surfaces arising from operations on these geometries are merged back together (failing if this is not possible).

Methods

convex_hull(self, surfaces, polygons)	Find the convex hull of these geometries.
cut(self, template, multipolygon_resolver, ...)	Crop this Geometries' polygons and surfaces onto a provided template.
fill_gaps(self, working_scale, ...)	Expand this geometries' polygons to fill its bounding box, using distance to assign empty space.
rasterize(self, box, NoneType] = None, ...)	Rasterize one or more owned geometries.
register_polygon(self, name, path, ...)	Adds a named polygon path to this object.
register_polygons(self, polygons, Union[str, ...)	utility for registering multiple polygons.
register_surface(self, name, path, ...)	Adds a line (e.g.

Continued on next page

Table 43 – continued from previous page

register_surfaces(self, surfaces, Union[str, ...)	utility for registering multiple surfaces.
to_json(self)	Write contained polygons to a json-serializable format
transform(self, transform, float], ...)	Apply a transform to each owned geometry.

close_bounds
The smallest bounding box enclosing these geometries.

convex_hull(*self, surfaces: bool = True, polygons: bool = True*) → shapely.geometry.polygon.Polygon
Find the convex hull of these geometries.

Parameters

surfaces [if True, include surfaces in the hull]

polygons [if True, include polygons in the hull]

Returns

The convex hull of the included geometries

cut(*self, template: shapely.geometry.polygon.Polygon, multipolygon_resolver: Union[Callable[[Iterable[shapely.geometry.polygon.Polygon]], shapely.geometry.polygon.Polygon], NoneType] = None, multisurface_resolver: Union[Callable[[Iterable[shapely.geometry.linestring.LineString]], shapely.geometry.linestring.LineString], NoneType] = None*) → 'Geometries'
Crop this Geometries' polygons and surfaces onto a provided template.

Parameters

template [portions of surfaces and polygons outside this shape will be] removed

multipolygon_resolver [This callable is applied to the outputs of] the intersection operation in order to resolve cases where a polygon has been cut into multiple components. The default method selects the largest by area.

multisurface_resolver [As multipolygon resolver, for surfaces. The] default method attempts to merge the surfaces.

Returns

A copy of this Geometries object, with polygons and surfaces cropped

default_multipolygon_resolver
By default, multiple polygons resulting from operations on these geometries are resolved by discarding all but the largest

default_multisurface_resolver
By default, multiple surfaces arising from operations on these geometries are merged back together (failing if this is not possible).

fill_gaps(*self, working_scale: float = 1.0, multipolygon_resolver: Union[Callable[[Iterable[shapely.geometry.polygon.Polygon]], shapely.geometry.polygon.Polygon], NoneType] = None*) → 'Geometries'
Expand this geometries' polygons to fill its bounding box, using distance to assign empty space.

Parameters

working_scale [The filling is carried out in a raster space, with 1] pixel corresponding to 1 unit in the coordinate system of your polygons. You can optionally rescale the polygons before rasterizing.

multipolygon_resolver [This method might obtain multiple output] polygons for a given input polygon. This callable collapses them into a single geometry. The default selects the largest.

Returns

A copy of this geometries object with the entire bounding box having

been filled.

rasterize(*self, box: Union[neuron_morphology.snap_polygons.bounding_box.BoundingBox, NoneType] = None, polygons: Union[Sequence[str], bool] = True, surfaces: Union[Sequence[str], bool] = False*) → Dict[str, numpy.ndarray]

Rasterize one or more owned geometries. Produce a mapping from object names to masks.

Parameters

shape [if provided, the output image shape. Otherwise, use the] rounded close bounding box shape

polygons [a list of names. Alternatively all (True) or none (False)]

lines [a list of names. Alternatively all (True) or none (False)]

Notes

uses rasterio.features.rasterize

register_polygon(*self, name: str, path: Union[str, Sequence[Sequence[float]], shapely.geometry.polygon.Polygon, shapely.geometry.polygon.LinearRing]*)

Adds a named polygon path to this object. Updates the close bounding box.

Parameters

name [identifier for this polygon]

path [defines the exterior of this (simple) polygon]

register_polygons(*self, polygons: Union[Dict[str, Union[str, Sequence[Sequence[float]], shapely.geometry.polygon.Polygon, shapely.geometry.polygon.LinearRing]], Sequence[Dict[str, Union[str, Sequence[Sequence[float]], shapely.geometry.polygon.Polygon, shapely.geometry.polygon.LinearRing]]]]*)

utility for registering multiple polygons. See register_polygon

register_surface(*self, name: str, path: Union[str, Sequence[Sequence[float]], shapely.geometry.linestring.LineString]*)

Adds a line (e.g. the pia/wm surfaces) to this object. Updates the bounding box.

Parameters

name [identifier for this surface]

path [defines the surface]

register_surfaces(*self, surfaces: Dict[str, Union[str, Sequence[Sequence[float]], shapely.geometry.linestring.LineString]]*)

utility for registering multiple surfaces. See register_surface

to_json(*self*) → Dict
Write contained polygons to a json-serializable format

transform(*self, transform: Callable[[float, float], Tuple[float, float]]*) → 'Geometries'
Apply a transform to each owned geometry. Return a new collection.

Parameters

transform [A callable which maps (vertical, horizontal) coordinates to] new (vertical, horizontal) coordinates.

neuron_morphology.snap_polygons.geometries.**clear_overlaps**(*stack: Dict[str, numpy.ndarray]*)
Given a stack of masks, remove all inter-mask overlaps inplace

Parameters

stack [Keys are names, values are masks (of the same shape). 0 indicates] absence

neuron_morphology.snap_polygons.geometries.**closest_from_stack**(*stack: Dict[str, numpy.ndarray]*)
Given a stack of images describing distance from several objects, find the closest object to each pixel.

Parameters

stack [Keys are names, values are ndarrays (of the same shape). Each pixel] in the values describes the distance from that pixel to the named object

Returns

closest [An integer array whose values are the closest object to each] pixel

names [A mapping from the integer codes in the "closest" array to names]

neuron_morphology.snap_polygons.geometries.**find_vertical_surfaces**(*polygons: Dict[str, shapely.geometry.polygon.Polygon], order: Sequence[str], pia: Union[shapely.geometry.linestring.L NoneType] = None, white_matter: Union[shapely.geometry.linestring.L NoneType] = None*)
Given a set of polygons describing cortical layer boundaries, find the boundaries between each layer.

Parameters

polygons [named layer polygons]

order [A sequence of names defining the order of the layer polygons from] pia to white matter

pia [The upper (from the perspective of cortex) pia surface.]

white_matter [The lower (from the perspective of cortex) white matter] surface.

Returns

dictionary whose keys are as "{name}_{side}" and whose values are linestrings describing these boundaries.

neuron_morphology.snap_polygons.geometries.**get_snapped_polys**(*closest: numpy.ndarray, name_lut: Dict[int, str], multipolygon_resolver: Callable[[Iterable[shapely.geometry.polygo shapely.geometry.polygon.Polygon]*) → Dict[str, shapely.geometry.polygon.Polygon]

Obtains named shapes from a label image.

Parameters

closest [label integer with integer codes]

name_lut [look up table from integer codes to string names]

Returns

mapping from names to polygons describing each labelled region

neuron_morphology.snap_polygons.geometries.**make_scale**(*scale: float = 1.0*) → Callable[[float, float], Tuple[float, float]]

A utility for making a 2D scale transform, suitable for transforming bounding boxes and Geometries

Parameters

scale [isometric scale factor]

Returns

A transform function

neuron_morphology.snap_polygons.geometries.**make_translation**(*horizontal: float, vertical: float*) → Callable[[float, float], Tuple[float, float]]

Utility for building a 2D translation transform

Parameters

horizontal [translate by this much along the first axis]

vertical [translate by this much along the second axis]

Returns

Function which applies the argued translation

neuron_morphology.snap_polygons.geometries.**rasterize**(*geometry: shapely.geometry.base.BaseGeometry, box: neuron_morphology.snap_polygons.bounding_box.Boundin* → <built-in function array>

Rasterize a shapely object to a grid defined by a provided bounding box.

Parameters

geometry [to be rasterized]

box [defines the window (in the same coordinate space as the geometry)] into which the geometry will be rasterized

Returns

A mask, where 1 indicates presence and 0 absence

neuron_morphology.snap_polygons.geometries.**safe_linemerge**(*linestrings: Union[shapely.geometry.linestring.LineString, Sequence[shapely.geometry.linestring.LineString]]*) → shapely.geometry.linestring.LineString

Wrapper around shapely.ops.linemerge that no-ops in case a single LineString or length-1 collection is argued.

neuron_morphology.snap_polygons.geometries.**select_largest_subpolygon**(*polygons: Union[shapely.geometry.polygo Iterable[shapely.geometry.polygon. error_threshold: float*) → shapely.geometry.polygon.Polyg

Given a collection of polygons, find the largest by area.

Parameters

polygons [To be filtered]

error_threshold [If the ratio of the largest polygon to the second] largest does not meet or exceed this value, reject the largest polygon.

Returns

the largest polygon

neuron_morphology.snap_polygons.geometries.**shared_faces**(*poly: shapely.geometry.polygon.Polygon, others: Iterable[shapely.geometry.polygon.Polygon]*) → shapely.geometry.linestring.LineString

Given a polygon and a set of other polygons that could be adjacent on the same side, find and connect that shared face.

Parameters

poly [Polygon] Polygon whose boundary with others we want to identify

others [list] List of other Polygons

Returns

LineString representing the shared face

neuron_morphology.snap_polygons.image_outputter module

Utilites for writing diagnostic overlay images

class neuron_morphology.snap_polygons.image_outputter.**ImageOutputter**(*native_geo: neuron_morphology.snap_polygons.result_geo: neuron_morphology.snap_polygons.image_specs: Optional[Sequence[Dict[KT, VT]]], alpha: float = 0.4, color_cycle: Optional[Sequence[T_co]] = None, savefig_kwargs: Optional[Dict[KT, VT]] = None*)

Bases: object

Overlays polygons and surfaces on provided images. Writes the results to files.

Parameters

native_geo [Layer geometries before gaps are filled]

result_geo [Layer geometries after gaps are filled]

image_specs [Each is a dictionary defining a single image. Must]

provide string keys:

- input_path : read from here
- output_path : write to (siblings of) this path
- **downsample** [the image will be scaled by this factor in each] dimension
- overlay_types : produce these kinds of overlay for this image

alpha [of the transparent overlays]

color_cycle [as polygon fills are drawn, cycle through these colors]

savefig_kwargs [Passed directly to pyplot's savefig, use to specify] e.g dpi.

Methods

draw_after(self, image, scale)	Display the post-fill polygons and surfaces overlaid on an image.

Continued on next page

Table 44 – continued from previous page

draw_before(self, image, scale)	Display the pre-fill polygons and surfaces overlaid on an image.
write_images(self)	For each image specified in this outputter and each overlay type requested for that image, produce and save an overlay.

DEFAULT_COLOR_CYCLE = ('c', 'm', 'y', 'k', 'r', 'g', 'b')

OVERLAY_TYPES = {'after': 'draw_after', 'before': 'draw_before'}

draw_after(*self*, *image: numpy.ndarray*, *scale: float = 1.0*)

Display the post-fill polygons and surfaces overlaid on an image.

Parameters

image [onto which objects will be drawn]

scale [required to transform from object space to image space]

Returns

A matplotlib figure containing the overlay

draw_before(*self*, *image: numpy.ndarray*, *scale: float = 1.0*)

Display the pre-fill polygons and surfaces overlaid on an image.

Parameters

image [onto which objects will be drawn]

scale [required to transform from object space to image space]

Returns

A matplotlib figure containing the overlay

write_images(*self*)

For each image specified in this outputter and each overlay type requested for that image, produce and save an overlay.

neuron_morphology.snap_polygons.image_outputter.**fname_suffix**(*path: str*, *suffix: str*)

Utility for adding a suffix to a path string. The suffix will be inserted before the extension.

neuron_morphology.snap_polygons.image_outputter.**make_pathpatch**(*vertices: Sequence[Tuple[float, float]]*, ***patch_kwargs*) → matplotlib.patches.PathPatch

Utility for building a matplotlib pathpatch from an array of vertices

Parameters

vertices [Defines the path. May be closed or open]

****patch_kwargs** [passed directly to pathpatch constructor]

neuron_morphology.snap_polygons.image_outputter.**read_image**(*path: str*, *decimate: int = 1*)

Read an image. Dispatch to an appropriate library based on that image's extension.

Parameters

path [to the image]

decimate [apply a decimation of this factor along each axis of the image]

neuron_morphology.snap_polygons.image_outputter.**read_jp2**(*path: str*, *decimate: int*)
Read (and symmetrically decimate) a jp2 file into a numpy array

neuron_morphology.snap_polygons.image_outputter.**read_with_ndimage**(*path: str*, *decimate: int*)
Read (and symmetrically decimate) an image file into a numpy array

neuron_morphology.snap_polygons.image_outputter.**write_figure**(*fig: matplotlib.figure.Figure*, **args*, ***kwargs*)
Write a matplotlib figure without respect to the current figure.

Parameters

fig [the figure to be writter]

***args, **kwargs** [passed to plt.savefig]

neuron_morphology.snap_polygons.types module

neuron_morphology.snap_polygons.types.**ensure_linestring**(*candidate: Union[str, Sequence[Sequence[float]], shapely.geometry.linestring.LineString]*) → shapely.geometry.linestring.LineString
Convert from one of many line representations to LineString

neuron_morphology.snap_polygons.types.**ensure_path**(*candidate: Union[str, Sequence[Sequence[float]]]*, *num_dims: int = 2*) → Sequence[Sequence[float]]
Ensure that an input path, which might be a "x,y,x,y" string, is represented as a list of lists instead.

Parameters

candidate [input coordinate sequence]

num_dims [how manu elements define a coordinate]

Returns

Contents of inputs, with each coordinate a list of float

neuron_morphology.snap_polygons.types.**ensure_polygon**(*candidate: Union[str, Sequence[Sequence[float]], shapely.geometry.polygon.Polygon, shapely.geometry.polygon.LinearRing]*) → shapely.geometry.polygon.Polygon
Convert from one of many polygon representations to Polygon

neuron_morphology.snap_polygons.types.**split_pathstring**(*pathstring: str*, *num_dims: int = 2*, *sep: str = ','*) → Sequence[Sequence[float]]
Converts a pathstring ("x,y,x,y...") to a num_points X num_dims list of lists of float

Parameters

pathstring [input coordinate sequence]

num_dims [how manu elements define a coordinate]

sep [character separating elements]

Returns

Contents of pathstring, with each coordinate a list of float

Module contents

neuron_morphology.transforms package

Subpackages

neuron_morphology.transforms.affine_transformer package

Submodules

neuron_morphology.transforms.affine_transformer.apply_affine_transform module

neuron_morphology.transforms.affine_transformer.apply_affine_transform.**main**()

Module contents

neuron_morphology.transforms.pia_wm_streamlines package

Submodules

neuron_morphology.transforms.pia_wm_streamlines.calculate_pia_wm_streamlines module

Module contents

neuron_morphology.transforms.scale_correction package

Submodules

neuron_morphology.transforms.scale_correction.compute_scale_correction module

neuron_morphology.transforms.scale_correction.compute_scale_correction.**collect_inputs**(*args: Dict[st Any]*) → Dict[st Any]

Parameters

args: dict of InputParameters

Returns

dict with string keys: morphology: Morphology object soma_marker_z: z value from the marker file soma_depth: soma depth cut_thickness: slice thickness

neuron_morphology.transforms.scale_correction.compute_scale_correction.**estimate_scale_corre**

Estimate a scale factor to correct the reconstructed morphology for slice shrinkage

Prior to reconstruction, the slice shrinks due to evaporation. This is most notable in the z axis, which is the slice thickness.

To correct for shrinkage we compare soma depth within the slice obtained soon after cutting the slice to the fixed_soma_depth obtained during the reconstruction. Then the scale correction is estimated as: scale = soma_depth / fixed_soma_depth. This is sensible as long as the z span of the corrected reconstruction is contained within the slice thickness. Thus we also estimate the maximum scale correction as: scale_max = cut_thickness / z_span, and take the smaller of scale and scale_max

Parameters

morphology: Morphology object

soma_depth: recorded depth of the soma when it was sliced

soma_marker_z: soma marker z value from revised marker file (z is on the slice surface for the marker file)

cut_thickness: thickness of the cut slice

Returns

scale factor correction

neuron_morphology.transforms.scale_correction.compute_scale_correction.**get_soma_marker_fro**

neuron_morphology.transforms.scale_correction.compute_scale_correction.**main**()

neuron_morphology.transforms.scale_correction.compute_scale_correction.**run_scale_correctio**

Module contents

neuron_morphology.transforms.tilt_correction package

Submodules

neuron_morphology.transforms.tilt_correction.compute_tilt_correction module

Module contents

neuron_morphology.transforms.upright_angle package

Submodules

neuron_morphology.transforms.upright_angle.compute_angle module

Module contents

Submodules

neuron_morphology.transforms.affine_transform module

class neuron_morphology.transforms.affine_transform.**AffineTransform**(*affine: Optional[Any] = None*)

Bases: *neuron_morphology.transforms.transform_base.TransformBase*

Handles transformations to a pia/wm aligned coordinate frame.

Methods

from_dict(affine_dict, float])	Create an AffineTransform from a dict with keys and values.
from_list(affine_list)	Create an Affine Transform from a list
to_dict(self)	Create dictionary defining the transformation.
to_list(self)	Create a list defining the transformation.
transform(self, vector)	Apply this transform to (3,) point or (n,3) array-like of points.
transform_morphology(self, morphology, ...)	Apply this transform to all nodes in a morphology.

classmethod from_dict (*affine_dict: Dict[str, float]*)

Create an AffineTransform from a dict with keys and values.

Parameters

affine_dict: keys and values corresponding to the following

[[tvr_00 tvr_01 tvr_02 tvr_09] [tvr_03 tvr_04 tvr_05 tvr_10] [tvr_06 tvr_07 tvr_08 tvr_11] [0 0 0 1]]

Returns

AffineTransform object

classmethod from_list(*affine_list: List[float]*)
Create an Affine Transform from a list

Parameters

affine_list: list of tvr values corresponding to:

[[tvr_00 tvr_01 tvr_02 tvr_09] [tvr_03 tvr_04 tvr_05 tvr_10] [tvr_06 tvr_07 tvr_08 tvr_11] [0 0 0 1]]

Returns

AffineTransform object

to_dict(*self*) → Dict
Create dictionary defining the transformation.

Returns

Dict with keys and values corresponding to the following:

[[tvr_00 tvr_01 tvr_02 tvr_09] [tvr_03 tvr_04 tvr_05 tvr_10] [tvr_06 tvr_07 tvr_08 tvr_11] [0 0 0 1]]

to_list(*self*) → List
Create a list defining the transformation.

Returns

List with values corresponding to the following:

[[tvr_00 tvr_01 tvr_02 tvr_09] [tvr_03 tvr_04 tvr_05 tvr_10] [tvr_06 tvr_07 tvr_08 tvr_11] [0 0 0 1]]

transform(*self*, *vector: Any*) → numpy.ndarray
Apply this transform to (3,) point or (n,3) array-like of points.

Parameters

vector: a (3,) array-like point or a (n,3) array-like array of points to be transformed

Returns

numpy.ndarray with same shape as input

transform_morphology(*self*, *morphology: neuron_morphology.morphology.Morphology*, *clone: bool = False*, *scale_radius: bool = True*) → neuron_morphology.morphology.Morphology
Apply this transform to all nodes in a morphology.

Parameters

morphology: a Morphology loaded from an swc file

clone: make a new object if True

scale_radius: apply radius scaling if True

Returns

A Morphology

neuron_morphology.transforms.affine_transform.**affine_from_transform**(*transform: Any*)
Create affine transformation.

Parameters

transformation: (3, 3) row major array-like transformation

Returns

(4, 4) numpy.ndarray affine matrix

neuron_morphology.transforms.affine_transform.**affine_from_transform_translation**(*transform: Union[Any, NoneType] = None*, *translation: Union[Any, NoneType] = None*, *translate_first: bool = False*)

Create affine from linear transformation and translation.

Affine transformation of vector x -> Ax + b in 3D: [A, b

0, 0, 0, 1]

A is a 3x3 linear tranformation b is a 3x1 translation

Parameters

transform: linear transformation (3, 3) array-like

translation: linear translation (3,) array-like

translate_first: apply the translation before the transform

Returns

(4, 4) numpy.ndarray affine matrix

neuron_morphology.transforms.affine_transform.**affine_from_translation**(*translation: Any*)

Create an affine translation.

Parameters

translation: array-like vector of x, y, and z translations

Returns

(4, 4) numpy.ndarray affine matrix

neuron_morphology.transforms.affine_transform.**rotation_from_angle**(*angle: float*, *axis: int = 2*)

Create an affine matrix from a rotation about a specific axis.

Parameters

angle: rotation angle in radians

axis: axis to rotate about, 0=x, 1=y, 2=z (default z axis)

Returns

(3, 3) numpy.ndarray rotation matrix

neuron_morphology.transforms.geometry module

Some handy utilities for working with vector geometries

neuron_morphology.transforms.geometry.**get_ccw_vertices**(*vertices: List[Tuple]*)

Generates counter clockwise vertices from vertices describing a simple polygon

Method: Simplification of the shoelace formula, which calculates area of a simple polygon by integrating the area under each line segment of the polygon. If the total area is positive, the vertices were traversed in clockwise order, and if it is negative, they were traversed in counterclockwise order.

Parameters

vertices: vertices describing a convex polygon (vertices[0] = vertices[-1])

Returns

vertices in counter clockwise order

neuron_morphology.transforms.geometry.**get_ccw_vertices_from_two_lines**(*line1: List[Tuple], line2: List[Tuple]*)

Convenience method two do both get_vertices_from_two_lines() and get_ccw_vertices()

neuron_morphology.transforms.geometry.**get_vertices_from_two_lines**(*line1: List[Tuple], line2: List[Tuple]*)

Generates circular vertices from two lines

Parameters

line1, line2: List of coordinates describing two lines

Returns

vertices of the simple polygon created from line 1 and 2

(first vertex = last vertex)

1-2-3-4

5-6-7-8 -> [1-2-3-4-8-7-6-5-1]

neuron_morphology.transforms.geometry.**prune_two_lines**(*line1: List[Tuple], line2: List[Tuple]*)

check the boundary to avoid intersections with side lines

Parameters

line1, line2: List of coordinates describing two lines

Returns

line1, line2: boundary pruned if needed

neuron_morphology.transforms.streamline module

neuron_morphology.transforms.transform_base module

class neuron_morphology.transforms.transform_base.**TransformBase**
Bases: abc.ABC

Abstract base class for implementing swc transforms. Each child class should implement these methods.

Methods

transform(self)	Apply this transform to (3,) point or (3,n) array-like of points.
transform_morphology(self)	Apply this transform to all nodes in a morphology.

transform(*self*)
Apply this transform to (3,) point or (3,n) array-like of points.

Returns

numpy.ndarray with same shape as input

transform_morphology(*self*) → neuron_morphology.morphology.Morphology
Apply this transform to all nodes in a morphology.

Returns

A Morphology

Module contents

neuron_morphology.validation package

Submodules

neuron_morphology.validation.bits_validation module

neuron_morphology.validation.bits_validation.**validate**(*morphology*)

neuron_morphology.validation.bits_validation.**validate_independent_axon_has_more_than_four_**
This function checks if an independent (parent is -1) axon has more than three nodes

neuron_morphology.validation.bits_validation.**validate_types_three_four_traceable_back_to_so**
This function checks if types 3,4 are traceable back to soma

neuron_morphology.validation.marker_validation module

neuron_morphology.validation.marker_validation.**validate**(*marker_file*, *morphology*)

neuron_morphology.validation.marker_validation.**validate_coordinates_corresponding_to_axon_t**

This function checks whether the coordinates for each axon marker corresponds to a tip of a axon type in the related morphology

neuron_morphology.validation.marker_validation.**validate_coordinates_corresponding_to_dendri**

This function checks whether the coordinates for each dendrite marker corresponds to a tip of a dendrite type in the related morphology

neuron_morphology.validation.marker_validation.**validate_expected_name**(*marker_file*)
This function checks whether the markers have the expected types

neuron_morphology.validation.marker_validation.**validate_no_reconstruction_count**(*marker_file*)
This function checks whether there is exactly one type 20 in the file

neuron_morphology.validation.marker_validation.**validate_type_thirty_count**(*marker_file*)
This function checks whether there is exactly one type 30 in the file

neuron_morphology.validation.morphology_statistics module

neuron_morphology.validation.morphology_statistics.**count_number_of_independent_axons**(*morpho*
This functions counts the number of independent axons (parent is -1)

neuron_morphology.validation.morphology_statistics.**morphology_statistics**(*morphology*)

neuron_morphology.validation.radius_validation module

neuron_morphology.validation.radius_validation.**slope_linear_regression_branch_order_avg_rad**

Use linear regression to find the slope of the best fit line

neuron_morphology.validation.radius_validation.**validate**(*morphology*)

neuron_morphology.validation.radius_validation.**validate_constrictions**(*morphology*)
This function checks if the radius of basal dendrite and apical dendrite nodes is smaller 2.0px

neuron_morphology.validation.radius_validation.**validate_extreme_taper**(*morphology*)
This function checks whether there is an extreme taper. Extreme taper occurs when for each segment, the average radius of the first two nodes is more than two times the average radius of the last two nodes.

Note: This tests is limited to segments of at lease 8 nodes.

neuron_morphology.validation.radius_validation.**validate_radius_has_negative_slope_dendrite**

This function checks whether the radius for dendrite nodes decreases when you are going away from the soma.

neuron_morphology.validation.radius_validation.**validate_radius_threshold**(*morphology*)
This function validates the radius for types 1, 3, and 4

neuron_morphology.validation.report module

class neuron_morphology.validation.report.**Report**
Bases: object

Methods

add_swc_stats(self, swc_file, stats)	This function creates a report for swc statistics

add_marker_results	
add_swc_results	
has_results	
to_json	

add_marker_results (*self*, *marker_file*, *results*)

add_swc_results (*self*, *swc_file*, *results*)

add_swc_stats (*self*, *swc_file*, *stats*)
This function creates a report for swc statistics

has_results (*self*)

to_json (*self*)

neuron_morphology.validation.resample_validation module

neuron_morphology.validation.resample_validation.**validate** (*morphology*)

neuron_morphology.validation.resample_validation.**validate_distance_between_connected_nodes**

neuron_morphology.validation.result module

exception neuron_morphology.validation.result.**InvalidMarkerFile** (*validation_errors*)
Bases: ValueError

validation_errors

exception neuron_morphology.validation.result.**InvalidMorphology** (*validation_errors*)
Bases: ValueError

validation_errors

class neuron_morphology.validation.result.**MarkerValidationError** (*message*, *marker*, *level*)
Bases: object

Attributes

level

marker

message

level

marker

message

class neuron_morphology.validation.result.**NodeValidationError** (*message*, *node_ids*, *level*)
Bases: object

Attributes

level

message

node_ids

level

message

node_ids

neuron_morphology.validation.structure_validation module

neuron_morphology.validation.structure_validation.**validate**(*morphology*)

neuron_morphology.validation.structure_validation.**validate_children_nodes_appear_before_pa**

neuron_morphology.validation.type_validation module

neuron_morphology.validation.type_validation.**valid_dendrite_parent**(*morphology*, *node*, *valid_parent_type*)

neuron_morphology.validation.type_validation.**validate**(*morphology*)

neuron_morphology.validation.type_validation.**validate_count_node_parent**(*morphology*, *node_type*, *parent_type*, *expected_count*)

This function validates the number of nodes that have a specific type of parent

neuron_morphology.validation.type_validation.**validate_expected_types**(*morphology*)

This function validates the expected types of the nodes

neuron_morphology.validation.type_validation.**validate_immediate_children_of_soma_cannot_bra**

This function validates that immediate children of soma cannot branch

neuron_morphology.validation.type_validation.**validate_multiple_axon_initiation_points**(*morph*

This function validates that the parent of axon (either type 1 or 3) only happens once

neuron_morphology.validation.type_validation.**validate_node_parent**(*morphology*)

This function validates the type of parent node for a specific type of child node

neuron_morphology.validation.type_validation.**validate_number_of_soma_nodes**(*morphology*)

This function validates the number of type 1 nodes

neuron_morphology.validation.validate_reconstruction module

neuron_morphology.validation.validate_reconstruction.**main**()

neuron_morphology.validation.validate_reconstruction.**parse_arguments**(*args*)

This function parses command line arguments

Module contents

neuron_morphology.validation.**validate_marker**(*marker*, *morphology*)

neuron_morphology.validation.**validate_morphology**(*morphology*)

neuron_morphology.vis package

Submodules

neuron_morphology.vis.morphovis module

neuron_morphology.vis.morphovis.**plot_cortical_boundary**(*pia_coords*, *wm_coords*, *ax*)

neuron_morphology.vis.morphovis.**plot_depth_field**(*depth_field*, *ax*)

neuron_morphology.vis.morphovis.**plot_gradient_field**(*gradient_field*, *ax*)

neuron_morphology.vis.morphovis.**plot_morphology_xy**(*morphology*, *ax*)

neuron_morphology.vis.morphovis.**plot_morphology_zy**(*morphology*, *ax*)

neuron_morphology.vis.morphovis.**plot_soma**(*soma_center*, *ax*)

Module contents

4.1.2 Submodules

neuron_morphology.constants module

neuron_morphology.lims_apical_queries module

neuron_morphology.marker module

class neuron_morphology.marker.**Marker**(**args*, ***kwargs*)

Bases: dict

Simple dictionary class for handling reconstruction marker objects.

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(self, key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop()	If key is not found, d is returned if given, otherwise KeyError is raised

Continued on next page

Table 48 – continued from previous page

popitem(self, /)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(self, key[, default])	Insert key with a value of default if key is not in the dictionary.
update()	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

neuron_morphology.marker.**read_marker_file**(*file_name*)
read in a marker file and return a list of dictionaries

neuron_morphology.morphology module

class neuron_morphology.morphology.**Morphology**(*nodes*, *node_id_cb*, *parent_id_cb*)
Bases: sphinx.ext.autodoc.importer._MockObject

Methods

__call__(self, *args, **kw)	Call self as a function.
breadth_first_traversal(self, visit[, ...])	Apply a function to each node of a connected graph in breadth-first order
depth_first_traversal(self, visit[, ...])	Apply a function to each node of a connected graph in depth-first order
get_compartment_surface_area(self, compartment)	Calculate the surface area of a single compartment.
get_compartment_volume(self, compartment)	Calculate the volume of a single compartment.
get_root(self)	Return the first found root node If the input SWC file does not have any root node, it will return None
get_roots_for_analysis(self[, root, node_types])	Returns a list of all trees to be analyzed, based on the supplied root.
get_soma(self)	Return one soma node labeled with SOMA If the input SWC file does not have any node labeled with SOMA, it will return None
swap_nodes_edges(self[, merge_cb, ...])	Build a new tree whose nodes are the edges of this tree and vice-versa
validate(self[, strict])	Validate the neuron morphology in

build_intermediate_nodes	
children_of	
clone	
euclidean_distance	
get_branch_order_for_node	
get_branch_order_for_segment	
get_branching_nodes	
get_children	

Continued on next page

Table 50 – continued from previous page

get_children_of_node_by_types	
get_compartment_for_node	
get_compartment_length	
get_compartment_midpoint	
get_compartments	
get_dimensions	
get_leaf_nodes	
get_max_id	
get_node_by_types	
get_non_soma_nodes	
get_number_of_trees	
get_root_for_tree	
get_root_id	
get_roots	
get_roots_for_nodes	
get_segment_length	
get_segment_list	
get_tree_list	
has_type	
is_node_at_beginning_of_segment	
is_node_at_end_of_segment	
is_soma_child	
midpoint	
node_by_id	
parent_of	

breadth_first_traversal (*self*, *visit*, *neighbor_cb=None*, *start_id=None*)
Apply a function to each node of a connected graph in breadth-first order

Parameters

visit [callable] Will be applied to each node. Signature must be visit(node). Return is ignored.

neighbor_cb [callable, optional] Will be used during traversal to find the next nodes to be visited. Signature must be neighbor_cb(node id) -> list of node_ids. Defaults to self.child_ids.

start_id [hashable, optional] Begin the traversal from this node. Defaults to self.get_root_id().

Notes

assumes rooted, acyclic

build_intermediate_nodes (*self*, *make_intermediates_cb*, *set_parent_id_cb*)

children_of (*self*, *node*)

clone (*self*)

depth_first_traversal (*self*, *visit*, *neighbor_cb=None*, *start_id=None*)
Apply a function to each node of a connected graph in depth-first order

Parameters

visit [callable] Will be applied to each node. Signature must be visit(node). Return is ignored.

neighbor_cb [callable, optional] Will be used during traversal to find the next nodes to be visited. Signature must be neighbor_cb(node_id) -> list of node_ids. Defaults to self.child_ids.

start_id [hashable, optional] Begin the traversal from this node. Defaults to self.get_root_id().

Notes

assumes rooted, acyclic

static euclidean_distance(*node1*, *node2*)

get_branch_order_for_node(*self*, *node*)

get_branch_order_for_segment(*self*, *segment*)

get_branching_nodes(*self*, *node_types=None*)

get_children(*self*, *node*, *node_types=None*)

get_children_of_node_by_types(*self*, *node*, *node_types*)

get_compartment_for_node(*self*, *node*, *node_types=None*)

get_compartment_length(*self*, *compartment*)

get_compartment_midpoint(*self*, *compartment*)

get_compartment_surface_area(*self, compartment: Sequence[Dict]*) → float
Calculate the surface area of a single compartment. Treats the compartment as a circular conic frustum and calculates its lateral surface area. This is:

pi * (r_1 + r_2) * sqrt((r_2 - r_1) ** 2 + L ** 2)

Parameters

compartment [two-long sequence. Each element is a node and must have] 3d position data ("x", "y", "z") and a "radius"

Returns

The surface area of the sides of the compartment

get_compartment_volume(*self, compartment: Sequence[Dict]*) → float
Calculate the volume of a single compartment. Treats the compartment as a circular conic frustum and calculates its volume as:

pi * L * (r_1 ** 2 + r_1 * r_2 + r_2 ** 2) / 3

Parameters

compartment [two-long sequence. Each element is a node and must have] 3d position data ("x", "y", "z") and a "radius"

Returns

The volume of the compartment

get_compartments(*self*, *nodes=None*, *node_types=None*)

get_dimensions(*self*, *node_types=None*)

get_leaf_nodes(*self*, *node_types=None*)

get_max_id(*self*)

get_node_by_types(*self*, *node_types=None*)

get_non_soma_nodes(*self*)

get_number_of_trees(*self*, *nodes=None*)

get_root(*self*)
Return the first found root node If the input SWC file does not have any root node, it will return None

Parameters

morphology: Morphology object

Returns

Root node object

get_root_for_tree(*self*, *tree_number*)

get_root_id(*self*)

get_roots(*self*)

get_roots_for_analysis(*self*, *root=None*, *node_types=None*)
Returns a list of all trees to be analyzed, based on the supplied root. These trees are the list of all children of the root, if root is not None, and the root node of all trees in the morphology if root is None.

Parameters

morphology: Morphology object

root: dict

This is the node from which to count branches under. When root=None,

all separate trees in the morphology are returned.

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Type to restrict search to

Returns

Array of Node objects

get_roots_for_nodes(*self*, *nodes*)

get_segment_length(*self*, *segment*)

get_segment_list(*self*, *node_types=None*)

get_soma(*self*)
Return one soma node labeled with SOMA If the input SWC file does not have any node labeled with SOMA, it will return None

Parameters

morphology: Morphology object

Returns

Soma node object

get_tree_list(*self*)

has_type(*self*, *node_type*)

is_node_at_beginning_of_segment(*self*, *node*)

is_node_at_end_of_segment(*self*, *node*)

is_soma_child(*self*, *node*)

static midpoint(*node1*, *node2*)

node_by_id(*self*, *node_id*)

parent_of(*self*, *node*)

swap_nodes_edges(*self*, *merge_cb=None*, *parent_id_cb=None*, *make_root_cb=None*, *start_id=None*)

Build a new tree whose nodes are the edges of this tree and vice-versa

Parameters

merge_cb [callable, optional]

parent_id_cb [callable, optional]

make_root_cb [callable, optional]

start_id [hashable, optional]

Notes

assumes rooted, acyclic

validate(*self*, *strict=False*)

Validate the neuron morphology in [bits, radius, resample, type, structure]

neuron_morphology.morphology_builder module

class neuron_morphology.morphology_builder.**MorphologyBuilder**

Bases: object

Attributes

active_node_id

next_id

parent_id

Methods

apical_dendrite(self[, x, y, z, radius])	Convenience for creating an apical dendrite node.
axon(self[, x, y, z, radius])	Convenience for creating an axon node.
basal_dendrite(self[, x, y, z, radius])	Convenience for creating a basal dendrite node.
build(self)	Construct a Morphology object using this builder.
child(self, x, y, z, node_type[, radius])	Add a child node to the current active node.
root(self[, x, y, z, node_type, radius])	Add a new root node (parent -1) to this reconstruction.
up(self[, by])	Terminate a branch.

active_node_id

apical_dendrite(*self*, *x=None*, *y=None*, *z=None*, *radius=1*)
Convenvience for creating an apical dendrite node. Will not create a root.

axon(*self*, *x=None*, *y=None*, *z=None*, *radius=1*)
Convenvience for creating an axon node. Will not create a root.

basal_dendrite(*self*, *x=None*, *y=None*, *z=None*, *radius=1*)
Convenvience for creating a basal dendrite node. Will not create a root.

build(*self*)
Construct a Morphology object using this builder. This is a non- destructive operation. The Morphology will be validated at this stage.

child(*self*, *x*, *y*, *z*, *node_type*, *radius=1*)
Add a child node to the current active node. This will become the new active node.

next_id

parent_id

root(*self*, *x=0*, *y=0*, *z=0*, *node_type=1*, *radius=1*)
Add a new root node (parent -1) to this reconstruction. This will be the new active node.

up(*self*, *by=1*)
Terminate a branch. Set the active node to the previous active node's ancestor.

Parameters

by [how far (up the tree) to set the new active node. Default is the] parent of the current node (1). 2 would correspond to the

neuron_morphology.swc_io module

neuron_morphology.swc_io.**apply_casts**(*df*, *casts*)

neuron_morphology.swc_io.**morphology_from_swc**(*swc_path*)

neuron_morphology.swc_io.**morphology_to_swc**(*morphology*, *swc_path*, *comments=None*)
Write an swc file from a morphology object

neuron_morphology.swc_io.**read_swc**(*path*, *columns=('id', 'type', 'x', 'y', 'z', 'radius', 'parent')*, *sep=' '*, *casts={'id': <class 'int'>, 'parent': <class 'int'>, 'type': <class 'int'>}*)
Read an swc file into a pandas dataframe

neuron_morphology.swc_io.**write_swc**(*data*, *path*, *comments=None*, *sep=' '*, *columns=('id', 'type', 'x', 'y', 'z', 'radius', 'parent')*, *casts={'id': <class 'int'>, 'parent': <class 'int'>, 'type': <class 'int'>}*)
Write an swc file from a pandas dataframe

4.1.3 Module contents

Top-level package for neuron_morphology.

neuron morphology is an open-source Python package for working with single-neuron morphological reconstruction data, such as those in the Allen Cell Types Database. It provides tools for transforming, analyzing, and visualizing these data. To get started, take a look at the installation instructions and the usage guides.

To report a bug or request a feature, see the issues page.

Python Module Index

n

Index