neuron morphology

Release 1.0.1

Dec 19, 2022

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 3.8-3.10 on Linux, OSX, and Windows. Similar Python versions (e.g. 3.7-3.10) 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=3.10
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-dolfinx>=0.4.2" # optional for using streamlines
pip install gmsh # optional for using streamlines; note that certain platforms (e.g.␣
↪Centos) may need to install from source
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

API Reference

This page contains auto-generated API reference documentation[1].

4.1 neuron_morphology

Top-level package for neuron_morphology.

4.1.1 Subpackages

neuron_morphology.feature_extractor

Submodules

neuron_morphology.feature_extractor.__main__

Module Contents

Functions

[1] Created with sphinx-autoapi

extract_multiple(reconstructions: List[Dict[str, Any]], feature_set: str, heavy_output_path: str, required_marks: Optional[List[str]] = None, only_marks: Optional[List[str]] = None, num_processes: Optional[int] = None, global_parameters: Optional[Dict[str, Any]] = None, output_table_path: Optional[str] = None)	For each path in swc_paths, load the file into a morphology and (attempt
main()	

neuron_morphology.feature_extractor.__main__.**extract_multiple**(*reconstructions: List[Dict[str, Any]], feature_set: str, heavy_output_path: str, required_marks: Optional[List[str]] = None, only_marks: Optional[List[str]] = None, num_processes: Optional[int] = None, global_parameters: Optional[Dict[str, Any]] = None, output_table_path: Optional[str] = None*)

For each path in swc_paths, load the file into a morphology and (attempt to) extract each feature in the set specified by feature_set.

Because of how Windows handles multiprocessing, run_feature_extraction must be in another py file.

Parameters

reconstructions [specify the reconstructions on which to compute features]

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

heavy_output_path [write "heavy" outputs, such as arrays, to this h5 file]

only_marks [names marks to which calculation will be restricted]

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

num_processes [use this many cores in the multiprocessing pool.]

global_parameters [a dictionary specifying cross-reconstruction] parameters

output_table_path [if not none, write a flattened table of features here]

Returns

a dictionary whose keys are reconstruction identifers and whose values are the outputs of run_feature_extraction for those reconstructions.

neuron_morphology.feature_extractor.__main__.**main**()

neuron_morphology.feature_extractor._schemas

Module Contents

Classes

ReferenceLayerDepths	mm.Schema class with support for making fields default to
Reconstruction	mm.Schema class with support for making fields default to
GlobalParameters	mm.Schema class with support for making fields default to
InputParameters	The base marshmallow schema used by ArgSchemaParser to identify
OutputParameters	mm.Schema class with support for making fields default to

Functions

validate_point_depths_path(path: str)	Check whether a layered point depths path is usable.

neuron_morphology.feature_extractor._schemas.**validate_point_depths_path**(*path: str*)

Check whether a layered point depths path is usable.

class neuron_morphology.feature_extractor._schemas.**ReferenceLayerDepths**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

key

names

boundaries

classmethod is_valid(*cls*, *value*)

class neuron_morphology.feature_extractor._schemas.**Reconstruction**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

swc_path

identifier

layered_point_depths_path

class neuron_morphology.feature_extractor._schemas.**GlobalParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

reference_layer_depths

class neuron_morphology.feature_extractor._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

The base marshmallow schema used by ArgSchemaParser to identify input_json and output_json files and the log_level

reconstructions

heavy_output_path

feature_set

only_marks

required_marks

output_table_path

num_processes

global_parameters

class neuron_morphology.feature_extractor._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

results

neuron_morphology.feature_extractor.data

Module Contents

Classes

Data	

Functions

get_morphology(data: MorphologyLike)	Decay a Data to a Morphology, leaving Morphologies untouched

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

__hash__(*self*)
Return hash(self).

neuron_morphology.feature_extractor.data.**MorphologyLike**

neuron_morphology.feature_extractor.data.**get_morphology**(*data: MorphologyLike*)
Decay a Data to a Morphology, leaving Morphologies untouched

neuron_morphology.feature_extractor.feature_extraction_run

Module Contents

Classes

FeatureExtractionRun

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

select_marks(*self, marks: Collection[Type[Mark]], required_marks: AbstractSet[Type[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]

select_features(*self, features: Collection[MarkedFeature], only_marks: Optional[AbstractSet[Type[Mark]]] = 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]

extract(*self*)

For each selected feature, carry out calculation on this run's dataset.

Returns

self [This FeatureExtractionRun, with results updated]

serialize(*self*)

Return a dictionary describing this run

neuron_morphology.feature_extractor.feature_extractor

Module Contents

Classes

FeatureExtractor

neuron_morphology.feature_extractor.feature_extractor.**RegistrableFeature**

class neuron_morphology.feature_extractor.feature_extractor.**FeatureExtractor**(*features: Sequence[Feature] = tuple()*)

register_features(*self, features: Sequence[RegistrableFeature]*)

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

extract(*self*, *data: Data*, *only_marks: Optional[AbstractSet[Type[Mark]]] = None*, *required_marks: AbstractSet[Type[Mark]] = frozenset()*)

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.

neuron_morphology.feature_extractor.feature_specialization

Module Contents

Classes

FeatureSpecialization
AxonSpec
ApicalDendriteSpec
BasalDendriteSpec
DendriteSpec
AllNeuriteSpec
AxonCompareSpec
ApicalDendriteCompareSpec
BasalDendriteCompareSpec
DendriteCompareSpec
AllNeuriteCompareSpec

neuron_morphology.feature_extractor.feature_specialization.**Fs**

class neuron_morphology.feature_extractor.feature_specialization.**FeatureSpecialization**

name :str

marks :Set[Type[Mark]]

kwargs :Dict[str, Any]

classmethod factory(*cls: Type[Fs], name: str, marks: Set[Type[Mark]], kwargs: Dict[str, Any], display_name: Optional[str] = None*)

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

neuron_morphology.feature_extractor.feature_specialization.**SpecializationOption**

neuron_morphology.feature_extractor.feature_specialization.**SpecializationSet**

neuron_morphology.feature_extractor.feature_specialization.**SpecializationSets**

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

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

name = axon

marks

kwargs

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

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

name = apical_dendrite

marks

kwargs

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

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

name = basal_dendrite

marks

kwargs

class neuron_morphology.feature_extractor.feature_specialization.**DendriteSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

name = dendrite

marks

kwargs

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

name = all_neurites

marks

kwargs

neuron_morphology.feature_extractor.feature_specialization.**NEURITE_SPECIALIZATIONS**

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

name = axon

marks

kwargs

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

name = apical_dendrite

marks

kwargs

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

name = basal_dendrite

marks

kwargs

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

name = dendrite

marks

kwargs

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

name = all_neurites

marks

kwargs

neuron_morphology.feature_extractor.feature_specialization.**NEURITE_COMPARISON_SPECIALIZATIO**

neuron_morphology.feature_extractor.feature_writer

Utilities (mainly the FeatureWriter class) used by the feature extractor executable to format and write outputs.

Module Contents

Classes

FeatureWriter	
FeatureFormatter	Format feature results for output

Functions

has_subkey(subkey: str, key: str)	Check whether a string occurs as one of the "."-separated members of
add_layer_histogram(writer: FeatureWriter, owner: str, key: str, histogram: LayerHistogram) → str	Add a layer histogram to this writer's heavy data
process_earth_movers_distance(_writer: FeatureWriter, _owner: str, _key: str, value: EarthMoversDistanceResult) → Dict[str, Union[str, float]]	Convert an EarthMoversDistanceResult to a form suitable for json

class neuron_morphology.feature_extractor.feature_writer.**FeatureWriter**(*heavy_path: str*, *table_path: Optional[str] = None*, *formatters: Optional[Iterable['FeatureForm = None*, *filemode: Optional[str] = 'w'*)

add_run(*self, identifier: str, run: Dict[str, Any]*)

Add the results of a feature extraction run to this writer

Parameters

identifier [the unique identifier for this run]

run [will be added]

process_feature(*self*, *owner: str*, *key: str*, *value: Any*)

Processes a feature for writing. This may involve:

1. changing its type to something json serializable
2. adding its value to this writer's heavy output

Parameters

owner [identifies the reconstruction that owns this feature]

key [the name of the feature]

value [the feature's raw value]

Returns

Transformed feature value

write(*self*)

Write this writer's output to disk

Returns

This writer's outputs as a dictionary

validate_table_extension(*self*)

If an output table was requested, check that the path has a supported extension.

build_output_table(*self*)

Convert this writer's output to a reconstruction X feature table

Returns

the generated table

write_table(*self*)

Construct and write a reconstructions X features table from this writer.

register_formatters(*self, formatters: Iterable['FeatureFormatter']*)

Add formatters to this writer. The order matters! If multiple formatters match a feature, only the first will be applied.

Parameters

foratters [an ordered collection of formatter to register]

neuron_morphology.feature_extractor.feature_writer.**FeatureOutputHandler**

neuron_morphology.feature_extractor.feature_writer.**FeatureOutputCheck**

class neuron_morphology.feature_extractor.feature_writer.**FeatureFormatter**

Bases: typing.NamedTuple

Format feature results for output

name :str

check :FeatureOutputCheck

handler :FeatureOutputHandler

neuron_morphology.feature_extractor.feature_writer.**has_subkey**(*subkey: str*, *key: str*)

Check whether a string occurs as one of the "."-separated members of another.

neuron_morphology.feature_extractor.feature_writer.**add_layer_histogram**(*writer: FeatureWriter*, *owner: str*, *key: str*, *histogram: LayerHistogram*) → str

Add a layer histogram to this writer's heavy data

Parameters

owner [identifies the reconstruction that owns this feature]

key [the name of the histogram]

histogram [the histogram's data]

Returns

the path at which this writer's heavy data will be stored

neuron_morphology.feature_extractor.feature_writer.**process_earth_movers_distance**(*_writer: FeatureWriter*, *_owner: str*, *_key: str*, *value: EarthMoversDistanceResult*) → Dict[str, Union[str, float]]

Convert an EarthMoversDistanceResult to a form suitable for json serialization

Parameters

_* [these are unused]

value [The result to be transformed]

Returns

a dictionary like:

{ “result”: <float result>, “interpretation” : <name of interpretation enum value>

}

neuron_morphology.feature_extractor.feature_writer.**numpy_array_formatter**

neuron_morphology.feature_extractor.feature_writer.**normalized_depth_histogram_formatter**

neuron_morphology.feature_extractor.feature_writer.**earth_movers_distance_formatter**

neuron_morphology.feature_extractor.feature_writer.**DEFAULT_FEATURE_FORMATTERS**

neuron_morphology.feature_extractor.mark

Module Contents

Classes

Mark	A tag, intended for use in feature selection.
RequiresLayerAnnotations	A tag, intended for use in feature selection.
Intrinsic	Indicates intrinsic features that don’t rely on a ccf or scale.
Geometric	Indicates features that change depending on coordinate frame.
AllNeuriteTypes	Indicates features that are calculated for all neurite types.
RequiresDendrite	This feature can only be calculated for neurons with at least one
RequiresRelativeSomaDepth	This feature can only be calculated for relative soma depth
RequiresSoma	Indicates that these features require a soma.
RequiresApical	Indicates that these features require an apical dendrite.
RequiresBasal	Indicates that these features require a basal dendrite.
RequiresAxon	Indicates that these features require an axon.
RequiresRoot	Indicates that this features require a root. Warns if the root
BifurcationFeatures	Indicates a feature calculated on bifurcations.
CompartmentFeatures	Indicates a feature calculated on compartments.
TipFeatures	Indicates a feature calculated on tips (leaf nodes).
NeuriteTypeComparison	Indicates a feature that is a comparison between neurite types.
RequiresRadii	This feature can only be calculated if the radii of nodes are annotated.
RequiresReferenceLayerDepths	This feature can only be calculated if a referenceset of average depths
RequiresLayeredPointDepths	This feature can only be calculated if (cortical) points are annotated
RequiresRegularPointSpacing	This features can only be (meaningfully) calculated if the points (e.g.

Functions

check_nodes_have_key(data: Data, key: str) → bool	Checks whether each node in a morphology is annotated with some key.

neuron_morphology.feature_extractor.mark.**Mr**

class neuron_morphology.feature_extractor.mark.**Mark**

A tag, intended for use in feature selection.

classmethod validate(*cls*, *data: Data*)

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

classmethod factory(*cls: Type[Mr]*, *name: str*)

class neuron_morphology.feature_extractor.mark.**RequiresLayerAnnotations**

Bases: *neuron_morphology.feature_extractor.mark.Mark*

A tag, intended for use in feature selection.

classmethod validate(*cls*, *data: Data*)

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.**Intrinsic**

Bases: *neuron_morphology.feature_extractor.mark.Mark*

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

class neuron_morphology.feature_extractor.mark.**Geometric**

Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates features that change depending on coordinate frame.

class neuron_morphology.feature_extractor.mark.**AllNeuriteTypes**

Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates features that are calculated for all neurite types.

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

classmethod validate(*cls*, *data: Data*)

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

This feature can only be calculated for relative soma depth

classmethod validate(*cls*, *data: Data*)
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.

classmethod validate(*cls*, *data: Data*)
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.**RequiresApical**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates that these features require an apical dendrite.

classmethod validate(*cls*, *data: Data*)
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.

classmethod validate(*cls*, *data: Data*)
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.

classmethod validate(*cls*, *data: Data*)
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

classmethod validate(*cls*, *data: Data*)
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.**BifurcationFeatures**
Bases: *neuron_morphology.feature_extractor.mark.Mark*

Indicates a feature calculated on bifurcations.

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

Indicates a feature calculated on compartments.

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

Indicates a feature calculated on tips (leaf nodes).

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

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.

classmethod validate(*cls*, *data: Data*)
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.

classmethod validate(*cls*, *data: Data*)
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.**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.

classmethod validate(*cls*, *data: Data*)
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.

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

neuron_morphology.feature_extractor.marked_feature

Module Contents

Classes

MarkedFeature

Functions

specialize(feature: Feature, specialization_set: SpecializationSet) → Dict[str, MarkedFeature]	Bind some of a feature's keyword arguments, using provided
nested_specialize(feature: Feature, specialization_sets: SpecializationSets) → Dict[str, MarkedFeature]	Apply specializations hierarchically to a base feature. Generating a
marked(mark: Type[Mark])	Decorator for adding a mark to a function.

neuron_morphology.feature_extractor.marked_feature.**FeatureFn**

neuron_morphology.feature_extractor.marked_feature.**M**

class neuron_morphology.feature_extractor.marked_feature.**MarkedFeature**(*marks: Set[Type[Mark]], feature: Feature, name: Optional[str] = None, preserve_marks: bool = True*)

__slots__ = ['marks', 'feature', 'name']

__name__

__repr__(*self*)
Return repr(self).

__hash__(*self*)
Return hash(self).

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

__call__(*self*, **args*, ***kwargs*)
Execute the underlying feature, passing along all arguments

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

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

specialize(*self*, *option: SpecializationOption*)
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

classmethod ensure(*cls: Type[M], feature: Feature*)
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.

neuron_morphology.feature_extractor.marked_feature.**Feature**

neuron_morphology.feature_extractor.marked_feature.**specialize**(*feature: Feature*, *specialization_set: SpecializationSet*) → Dict[str, MarkedFeature]

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.marked_feature.**nested_specialize**(*feature: Feature*, *specialization_sets: SpecializationSets*) → Dict[str, MarkedFeature]

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.**marked**(*mark: Type[Mark]*)

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.run_feature_extraction

Module Contents

Functions

resolve_reference_layer_depths(key=None, names=None, boundaries=None)	Given either the name of a well known depths set or a set of names and
hydrate_parameters(parameters: Dict[str, Any]) → Dict[str, Any]	Resolve argued feature parameters to a format comprehensible by
setup_data(reconstruction: Dict[str, Any], global_parameters: Dict[str, Any]) → Tuple[str, Data]	Construct a Data for extracting features from a single reconstruction.
run_feature_extraction(reconstruction_spec: Dict[str, Any], feature_set: str, only_marks: List[str], required_marks: List[str], global_parameter_spec: Dict[str, Any]) → Tuple[str, Dict]	Run feature extraction for a single reconstruction.

neuron_morphology.feature_extractor.run_feature_extraction.**well_known_marks :Dict[str, Type**

neuron_morphology.feature_extractor.run_feature_extraction.**item**

neuron_morphology.feature_extractor.run_feature_extraction.**known_feature_sets**

neuron_morphology.feature_extractor.run_feature_extraction.**resolve_reference_layer_depths**(

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.**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.**setup_data**(*reconstruction: Dict[str, Any], global_parameters: Dict[str, Any]*) → Tuple[str, Data]

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.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.utilities

A collection of miscellaneous tools used by the feature extractor

Module Contents

Functions

unnest(inputs: Dict[str, Any], _prefix='') → Dict[str, Any]	Convert nested dictionaries (with string keys) to a dot-notation flat

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

neuron_morphology.features

Subpackages

neuron_morphology.features.branching

Submodules

neuron_morphology.features.branching.bifurcations

Module Contents

Functions

calculate_outer_bifs(morphology: Morphology, soma: Dict, node_types: Optional[List[int]]) → int	Counts the number of bifurcation points beyond the a sphere
num_outer_bifurcations(data: MorphologyLike, node_types: Optional[List[int]] = None) → int	Feature Extractor interface to calculate_outer_bifurcations. Returns
mean_bifurcation_angle_local(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Compute the average angle between child segments at
mean_bifurcation_angle_remote(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Compute the average angle between the next branch point or terminal

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

neuron_morphology.features.branching.bifurcations.**num_outer_bifurcations**(*data: MorphologyLike, node_types: Optional[List[int]] = None*) → int

Feature Extractor interface to calculate_outer_bifurcations. Returns the number of bifurcations (branch points), excluding those too close to the root (threshold is 1/2 the max distance from the root to any node).

Parameters

data [Holds a morphology object. No additional data is required]

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

neuron_morphology.features.branching.bifurcations.**mean_bifurcation_angle_local**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Compute the average angle between child segments at bifurcations throughout the morphology. Trifurcations are ignored. Note: this introduces possible segmentation artifacts if trifurcations are due to large segment sizes.

Parameters

data: The reconstruction whose max euclidean distance will be calculated

node_types: restrict consideration to these types

Returns

Scalar value

neuron_morphology.features.branching.bifurcations.**mean_bifurcation_angle_remote**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Compute the average angle between the next branch point or terminal tip of child segments at each bifurcation. Trifurcations are ignored. Note: this introduces possible segmentation artifacts if trifurcations are due to large segment sizes.

Parameters

data: The reconstruction whose max euclidean distance will be calculated

node_types: restrict consideration to these types

Returns

Scalar value, nan if no nodes

neuron_morphology.features.layer

Submodules

neuron_morphology.features.layer.layer_histogram

Module Contents

Classes

LayerHistogram	The results of calculating a within-layer depth histogram of points
EarthMoversDistanceInterpretation	Describes how to understand an earth mover's distance result. This is
EarthMoversDistanceResult	The result of comparing two histograms using earth mover's distance

Functions

ensure_tuple(inputs: Any, item_type: Type, if_none: Union[str, Tuple] = 'raise') → Tuple	Try to smartly coerce inputs to a tuple.
ensure_node_types(node_types)	Make sure the argued node types are a tuple
ensure_layers(layers)	Make sure the argued layer array is a tuple
earth_movers_distance(data: Data, node_types: Sequence[int], node_types_to_compare: Sequence[int], bin_size: float = 5) → Dict[str, EarthMoversDistanceResult]	Calculate the earth mover's distance between normalized histograms of
histogram_earth_movers_distance(from_hist: np.ndarray, to_hist: np.ndarray) → EarthMoversDistanceResult	Calculate the earth mover's distance between to histograms, normalizing
normalized_depth_histogram(data: Data, node_types: Optional[Sequence[int]] = None, bin_size=5.0) → Dict[str, LayerHistogram]	Calculates for each cortical layer a histogram of node depths within
normalized_depth_histograms_across_layers(data: Data, point_types: Optional[Tuple[int]] = None, only_layers: Optional[Tuple[str]] = None, bin_size=5.0) → Dict[str, LayerHistogram]	A helper function for running cortical depth histograms across multiple
normalized_depth_histogram_within_layer(point_depths: np.ndarray, local_layer_pia_side_depths: np.ndarray, local_layer_wm_side_depths: np.ndarray, reference_layer_depths: ReferenceLayerDepths, bin_size: float) → np.ndarray	Calculate a histogram of node depths within a single (cortical) layer.

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.**ensure_node_types**(*node_types*)
Make sure the argued node types are a tuple

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

class neuron_morphology.features.layer.layer_histogram.**LayerHistogram**
Bases: typing.NamedTuple

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

counts :np.ndarray

bin_edges :np.ndarray

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.

BothPresent = 0

OneEmpty = 1

BothEmpty = 2

class neuron_morphology.features.layer.layer_histogram.**EarthMoversDistanceResult**
Bases: typing.NamedTuple

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

result :float

interpretation :EarthMoversDistanceInterpretation

to_dict_human_readable(*self*)

neuron_morphology.features.layer.layer_histogram.**earth_movers_distance**(*data: Data, node_types: Sequence[int], node_types_to_compare: Sequence[int], bin_size: float = 5*) → Dict[str, EarthMoversDistanceResult]

Calculate the earth mover's distance between normalized histograms of node depths within cortical layers. Calculates one distance for each layer.

Parameters

data [Must be endowed with layered_point_depths and reference_layer_depths.] The morphology is not actually used directly.

node_types [Defines one set of points whose histograms to compare.]

node_types_to_compare [Defines the other set of points]

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

Returns

A mapping from layers to distances between histograms within those layers.

neuron_morphology.features.layer.layer_histogram.**histogram_earth_movers_distance**(*from_hist: np.ndarray, to_hist: np.ndarray*) → EarthMoversDistanceResult

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**(*data: Data*, *node_types: Optional[Sequence[int]] = None*, *bin_size=5.0*) → Dict[str, LayerHistogram]

Calculates for each cortical layer a histogram of node depths within that layer.

Parameters

data [Must have the following attributes:]

reference_layer_depths [A dictionary mapping layer names (str) to] ReferenceLayerDepths objects describing the average pia and white- matter side depths of this each layer.

layered_point_depths [A LayeredPointDepths defining for each point a] depth from pia. See LayeredPointDepths for more information.

node_types [for which to calculate the histograms]

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

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.layer_histogram.**normalized_depth_histogram_within_layer**(*po*
np
lo
ca
np
lo
ca
np
re
er
en
Re
er
en
La
er
bi
flo
→
np

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.layered_point_depths

Module Contents

Classes

LayeredPointDepths	

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

DF_COLS

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

classmethod from_dataframe(*cls*, *df: pd.DataFrame*)

classmethod from_csv(*cls*, *path: str*)

classmethod from_hdf5(*cls*, *path: str*)

classmethod read(*cls*, *path: str*)

neuron_morphology.features.layer.reference_layer_depths

Module Contents

Classes

ReferenceLayerDepths	Reference (e.g. average across specimens and regions) depths of

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

Bases: typing.NamedTuple

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.

pia_side :float

wm_side :float

scale :bool = True

thickness

classmethod sequential(*cls, 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.

neuron_morphology.features.layer.reference_layer_depths.**DEFAULT_HUMAN_ME_MET_REFERENCE_LAYE**

neuron_morphology.features.layer.reference_layer_depths.**DEFAULT_HUMAN_MTG_REFERENCE_LAYER_D**

neuron_morphology.features.layer.reference_layer_depths.**DEFAULT_MOUSE_ME_MET_REFERENCE_LAYE**

neuron_morphology.features.layer.reference_layer_depths.**DEFAULT_MOUSE_REFERENCE_LAYER_DEPTH**

neuron_morphology.features.layer.reference_layer_depths.**WELL_KNOWN_REFERENCE_LAYER_DEPTHS**

neuron_morphology.features.statistics

Submodules

neuron_morphology.features.statistics.coordinates

Module Contents

Classes

COORD_TYPE	Generic enumeration.
NodeSpec	
BifurcationSpec	
CompartmentSpec	
TipSpec	

Functions

get_compartment_coordinates(morphology, node_types: Optional[List[int]] = None)	Return the coordinates of the midpoint of each compartment
get_bifurcation_coordinates(morphology, node_types: Optional[List[int]] = None)	Return the coordinates of each bifurcation in the morphology
get_tip_coordinates(morphology, node_types: Optional[List[int]] = None)	Return the coordinates of each tip in the morphology
get_node_coordinates(morphology, node_types: Optional[List[int]] = None)	Return the coordinates of each node in the morphology
get_coordinates(morphology: Morphology, coordinate_type: COORD_TYPE = COORD_TYPE.NODE, node_types: Optional[List[int]] = None)	

class neuron_morphology.features.statistics.coordinates.**COORD_TYPE**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

NODE = 0

COMPARTMENT = 1

BIFURCATION = 2

TIP = 3

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

class neuron_morphology.features.statistics.coordinates.**NodeSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

name = node

marks

kwargs

class neuron_morphology.features.statistics.coordinates.**BifurcationSpec**

Bases: *neuron_morphology.feature_extractor.feature_specialization.FeatureSpecialization*

name = bifurcation

marks

kwargs

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

name = compartment

marks

kwargs

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

name = tip

marks

kwargs

neuron_morphology.features.statistics.coordinates.**COORD_TYPE_SPECIALIZATIONS**

neuron_morphology.features.statistics.coordinates.**get_compartment_coordinates**(*morphology*, *node_types: Optional[List[int]] = None*)

Return the coordinates of the midpoint of each compartment in the morphology

Parameters

morphology: Morphology object

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Returns

list: list of coordinates [x, y, z]

neuron_morphology.features.statistics.coordinates.**get_bifurcation_coordinates**(*morphology*, *node_types: Optional[List[int]] = None*)

Return the coordinates of each bifurcation in the morphology

Parameters

morphology: Morphology object

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Returns

list: list of coordinates [x, y, z]

neuron_morphology.features.statistics.coordinates.**get_tip_coordinates**(*morphology*, *node_types: Optional[List[int]] = None*)

Return the coordinates of each tip in the morphology

Parameters

morphology: Morphology object

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Returns

list: list of coordinates [x, y, z]

neuron_morphology.features.statistics.coordinates.**get_node_coordinates**(*morphology*, *node_types: Optional[List[int]] = None*)

Return the coordinates of each node in the morphology

Parameters

morphology: Morphology object

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Returns

list: list of coordinates [x, y, z]

neuron_morphology.features.statistics.coordinates.**get_coordinates**(*morphology: Morphology*, *coordinate_type: COORD_TYPE = COORD_TYPE.NODE*, *node_types: Optional[List[int]] = None*)

neuron_morphology.features.statistics.moments

Module Contents

Functions

moments(data: Data, node_types: Optional[List] = None, coord_type: COORD_TYPE = COORD_TYPE.NODE)	Calculate the moments of specific coordinate type and node type

neuron_morphology.features.statistics.moments.**moments**(*data: Data*, *node_types: Optional[List] = None*, *coord_type: COORD_TYPE = COORD_TYPE.NODE*)

Calculate the moments of specific coordinate type and node type

Parameters

data: Data Object containing a morphology

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

coord_type: Restrict analysis to specific coordinate type (see neuron_morphology.features.statistics.coordinates for options)

neuron_morphology.features.statistics.moments_along_max_distance_projection

Module Contents

Functions

moments_along_max_distance_projection(data: Data, node_types: Optional[List] = None, coord_type: COORD_TYPE = COORD_TYPE.BIFURCATION)	Calculate the distance projections of a specific compartment and coordinate type along

neuron_morphology.features.statistics.moments_along_max_distance_projection.**moments_along_m**

Calculate the distance projections of a specific compartment and coordinate type along the line segment connecting soma to the most distant (from soma) node of that compartment.

Parameters

data: Data Object containing a morphology

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

coord_type: Restrict which coordinate types are measured (i.e. projected along line segment) (see neuron_morphology.features.statistics.coordinates for options)

Returns

summary_dict: summary stats of distances projected along specified line segment

neuron_morphology.features.statistics.overlap

Module Contents

Functions

calculate_coordinate_overlap_from_min_max(coordinates: np.ndarray, minv: float, maxv: float, dimension: int = 1)	Return the % of coordinates that are above the max,
calculate_coordinate_overlap(coordinates_a, coordinates_b, dimension: int = 1)	Return the % of coordinates_a that are above, overlaping, and below
overlap(data: Data, node_types: Optional[List[int]] = None, node_types_to_compare: Optional[List[int]] = None, coord_type: COORD_TYPE = COORD_TYPE.NODE, dimension: int = 1)	Compares the locations of node_types to node_types_to_compare

neuron_morphology.features.statistics.overlap.**calculate_coordinate_overlap_from_min_max**(*coo np.r min floa max floa di- mer sion 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)

neuron_morphology.features.statistics.overlap.**calculate_coordinate_overlap**(*coordinates_a*, *co- or- di- nates_b*, *di- men- sion: 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.**overlap**(*data: Data, node_types: Optional[List[int]] = None, node_types_to_compare: Optional[List[int]] = None, coord_type: COORD_TYPE = COORD_TYPE.NODE, dimension: int = 1*)

Compares the locations of node_types to node_types_to_compare Calculate % of coordinates of node_types that are above, overlapping, and below the coordinates of node_types_to_compare

Example: calculate_overlap(

morphology, node_types=[AXON], node_types_to_compare=[APICAL_DENDRITE, BASAL_DENDRITE])

will return the percentage of AXON nodes that are above, overlapping, and below DENDRITE nodes

Parameters

data: Data Object containing a morphology

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

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

coord_type: Restrict analysis to specific coordinate type (see neuron_morphology.features.statistics.coordinates for options)

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

Submodules

neuron_morphology.features.default_features

Module Contents

neuron_morphology.features.default_features.**default_features**

neuron_morphology.features.dimension

Module Contents

Functions

dimension(data: Data, node_types: Optional[List] = None, coord_type: COORD_TYPE = COORD_TYPE.NODE, signed_bias=(False, True, False))	Get the height, width, depth, minimum, and maximum values of

neuron_morphology.features.dimension.**dimension**(*data: Data, node_types: Optional[List] = None, coord_type: COORD_TYPE = COORD_TYPE.NODE, signed_bias=(False, True, False)*)

Get the height, width, depth, minimum, and maximum values of specific coordinate type and node type centered

about the root

Parameters

data: Data Object containing a morphology

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

coord_type: Restrict analysis to specific coordinate type (see neuron_morphology.features.statistics.coordinates for options)

signed_bias: boolean tuple for whether the bias measure should be signed for (x, y, z)

neuron_morphology.features.intrinsic

Module Contents

Functions

num_tips(data: Data, node_types: Optional[List] = None)	Calculate number of tips
num_nodes(data: Data, node_types: Optional[List] = None)	Calculate number of nodes of a given type
child_ids_by_type(node_id, morphology, node_types=None)	Helper function for the traversal functions
calculate_branches_from_root(morphology, root, node_types=None)	Calculate the number of branches of a specific neuron type
num_branches(data: Data, node_types: Optional[List] = None)	Calculate number of branches
calculate_mean_fragmentation_from_root(morphology, root, node_types=None)	Calculate the mean fragmentation from a root
mean_fragmentation(data: Data, node_types: Optional[List] = None)	Calculate the mean number of compartments per branch
calculate_max_branch_order_from_root(morphology, root, node_types=None)	Calculate the greatest number of branches encountered among all
max_branch_order(data: Data, node_types: Optional[List] = None)	Calculate mean fragmentation

neuron_morphology.features.intrinsic.**num_tips**(*data: Data*, *node_types: Optional[List] = None*)

Calculate number of tips

Parameters

data: Data Object containing a morphology

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

neuron_morphology.features.intrinsic.**num_nodes**(*data: Data*, *node_types: Optional[List] = None*)

Calculate number of nodes of a given type

Parameters

data: Data Object containing a morphology

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.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.**num_branches**(*data: Data*, *node_types: Optional[List] = None*)

Calculate number of branches

Parameters

data: Data Object containing a morphology

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

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.**mean_fragmentation**(*data: Data*, *node_types: Optional[List] = None*)

Calculate the mean number of compartments per branch

Parameters

data: Data Object containing a morphology

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.**max_branch_order**(*data: Data*, *node_types: Optional[List] = None*)

Calculate mean fragmentation

Parameters

data: Data Object containing a morphology

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

neuron_morphology.features.path

Module Contents

Functions

_calculate_max_path_distance(morphology, root, node_types)	
calculate_max_path_distance(morphology, root, node_types=None)	Helper for max_path_distance. See below for more information.
max_path_distance(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculate the distance, following the path of adjacent neurites, from
early_branch_path(data: MorphologyLike, node_types: Optional[List[int]] = None, soma: Optional[Dict] = None) → float	Returns the ratio of the longest 'short' branch from a bifurcation to
_calculate_mean_contraction(morphology, reference, root, node_types)	Calculate the average contraction of all sections. In other words,
calculate_mean_contraction(morphology, root=None, node_types=None)	See mean_contraction
mean_contraction(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculate the average contraction of all sections. In other words,

neuron_morphology.features.path.**_calculate_max_path_distance**(*morphology*, *root*, *node_types*)

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.**max_path_distance**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Calculate the distance, following the path of adjacent neurites, from the soma to the furthest compartment. This is equivalent to the distance to the furthest SWC node.

Parameters

data [the input reconstruction]

node_types [if provided, restrict the calculation to nodes of these] types

Returns

The along-path distance from the soma to the farthest (in the along-path

sense) node.

neuron_morphology.features.path.**early_branch_path**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*, *soma: Optional[Dict] = None*) → float

Returns the ratio of the longest 'short' branch from a bifurcation to the maximum path length of the tree. In other words, for each bifurcation, the maximum path length below that branch is calculated, and the shorter of these values is used. The maximum of these short values is divided by the maximum path length.

Parameters

data [the input reconstruction]

node_types [if provided, restrict the calculation to nodes of these] types

soma [if provided, use this node as the root, otherwise infer the root] from the argued morphology

Returns

ratio of max short branch to max path length

neuron_morphology.features.path.**_calculate_mean_contraction**(*morphology*, *reference*, *root*, *node_types*)

Calculate the average contraction of all sections. In other words, calculate the average ratio of euclidean distance to path distance between all bifurcations in the morphology. Trifurcations are treated as bifurcations.

Parameters

morphology: Morphology object

reference: dict

This is the node of the previous bifurcation

root: dict

This is the node from which to measure branch contraction under

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Type to restrict search to

Returns

Two scalars: euclidean distance, path distance

These are the total bif-bif and bif-tip distances under this root

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

See mean_contraction

neuron_morphology.features.path.**mean_contraction**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Calculate the average contraction of all sections. In other words, calculate the average ratio of euclidean distance to path distance between all bifurcations in the morphology. Trifurcations are treated as bifurcations.

Parameters

data [the input reconstruction]

node_types [if provided, restrict the calculation to nodes of these] types

Returns

The average contraction across all sections in this reconstruction

neuron_morphology.features.size

Module Contents

Functions

total_length(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculate the total length across all compartments in a reconstruction
total_surface_area(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculates the sum of lateral surface areas across all comparments
total_volume(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculates the sum of volumes across all comparments (linked pairs of
mean_diameter(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculates the mean diameter of all nodes
parent_daughter_ratio_visitor(node: Dict[str, Any], morphology: Morphology, counters: Dict[str, Union[int, float]], node_types: Optional[List[int]] = None)	Calculates for a single node the ratio of the node's parent's radius to
mean_parent_daughter_ratio(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculate the average ratio of parent radii to child radii across a
max_euclidean_distance(data: MorphologyLike, node_types: Optional[List[int]] = None) → float	Calculate the furthest distance, in 3-space, of a compartment's end from

neuron_morphology.features.size.**total_length**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Calculate the total length across all compartments in a reconstruction

Parameters

data [the input reconstruction]

node_types [if provided, restrict the calculation to compartments] involving these types

Returns

The sum of segment lengths across all segments in the reconstruction

Notes

Excludes compartments where the parent is:

1. the soma
2. a root of the reconstruction

The logic here is that the soma root is likely to substantially overlap any of its compartments, while non-root soma nodes will be closer to the soma surface.

neuron_morphology.features.size.**total_surface_area**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Calculates the sum of lateral surface areas across all comparments (linked pairs of nodes) in a reconstruction. This approximates the total surface area of the reconstruction. See Morphology.get_compartment_surface_area for details.

Parameters

data [The reconstruction whose surface area will be computed]

node_types [restrict the calculation to compartments involving these node] types

Returns

The sum of compartment lateral surface areas across this reconstruction

neuron_morphology.features.size.**total_volume**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Calculates the sum of volumes across all comparments (linked pairs of nodes) in a reconstruction. This approximates the total volume of the reconstruction. See Morphology.get_compartment_volume for details.

Parameters

data [The reconstruction whose volume will be computed]

node_types [restrict the calculation to compartments involving these node] types

Returns

The sum of compartment volumes across this reconstruction

neuron_morphology.features.size.**mean_diameter**(*data: MorphologyLike*, *node_types: Optional[List[int]] = None*) → float

Calculates the mean diameter of all nodes

Parameters

morphology [The reconstruction whose mean diameter]

node_types [restrict the calculation to compartments involving these node] types

Returns

The average diameter across selected nodes

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

Calculate the average ratio of parent radii to child radii across a reconstruction.

Parameters

data [The reconstruction whose mean parent daugther ratio will be computed]

node_types [restrict the calculation to compartments involving these node] types

Notes

Note that this function differs from the L-measure parent daughter ratio, which calculates the ratio of the child node size to the parent. Note also that both the parent and child must be in node_types in order for a compartment to be included in the calculation

neuron_morphology.features.size.**max_euclidean_distance**(*data: MorphologyLike, node_types: Optional[List[int]] = None*) → float

Calculate the furthest distance, in 3-space, of a compartment's end from the soma. This is equivalent to the distance to the furthest SWC node.

Parameters

data: The reconstruction whose max euclidean distance will be calculated

node_types: restrict consideration to these types

Returns

The distance between the soma and the farthest-from-soma node in this

morphology.

neuron_morphology.features.soma

Module Contents

Functions

calculate_soma_surface(data: Data) → float	Approximates the surface area of the soma. Morphologies with only
calculate_relative_soma_depth(data: Data) → float	Calculate the soma depth relative to pia/wm
calculate_soma_features(data: Data)	Calculate the soma features
calculate_stem_exit_and_distance(data: Data, node_types: Optional[List[int]], z_scale=3.0)	Returns the relative radial position (stem_exit) on the soma where the

neuron_morphology.features.soma.**calculate_soma_surface**(*data: Data*) → float

Approximates the surface area of the soma. Morphologies with only a single soma node are supported.

Parameters

data: Data Object containing a morphology

Returns

Scalar value

neuron_morphology.features.soma.**calculate_relative_soma_depth**(*data: Data*) → float

Calculate the soma depth relative to pia/wm

Parameters

data: Data Object containing a morphology

Returns

Scalar value

neuron_morphology.features.soma.**calculate_soma_features**(*data: Data*)

Calculate the soma features

Parameters

data: Data Object containing a morphology

Returns

soma_features

neuron_morphology.features.soma.**calculate_stem_exit_and_distance**(*data: Data, node_types: Optional[List[int]], z_scale=3.0*)

Returns the relative radial position (stem_exit) on the soma where the tree holding the tree connects to the soma. 0 is on the bottom, 1 on the top, and 0.5 out a side. Also returns the distance (stem_distance) between the tree root and the soma surface.

Parameters

data: Data Object containing a morphology

soma: dict

soma node

node_types: list (AXON, BASAL_DENDRITE, APICAL_DENDRITE)

Type to restrict search to

Returns

(float, float):

First value is relative position (height, on [0,1]) of a

tree on soma. Second value is distance of the root from soma

neuron_morphology.layered_point_depths

Submodules

neuron_morphology.layered_point_depths.__main__

Module Contents

Functions

translate_field(field: xr.DataArray, by_x: float, by_y: float, inplace: bool = False)	Translate a spatial xarray dataset
setup_interpolator(field: xr.DataArray, dim: Optional[str], **kwargs) → RegularGridInterpolator	Build a regular grid interpolator from a dataarray
containing_layer(pos: Tuple[float, float], layers: List[Dict]) → Optional[str]	Find the layer in which a point is contained
tuplize(arr: np.array) → Tuple	Convert an array to a tuple
step_from_node(pos: Tuple[float, float], depth_interp: RegularGridInterpolator, dx_interp: RegularGridInterpolator, dy_interp: RegularGridInterpolator, surface: LineString, step_size: float, max_iter: int, adaptive_scale: int = 32) → Optional[float]	Walk through a gradient field, until a defined surface is passed.
get_node_intersections(node: Dict, depth_interp: RegularGridInterpolator, dx_interp: RegularGridInterpolator, dy_interp: RegularGridInterpolator, layers: List[Dict], step_size: float, max_iter: int) → Dict	Given a node, find its layer and intersection depths. Then return a row
setup_layers(layers: List[Dict])	Convert layer bounds, pia, and white matter surfaces to shapely objects
run_layered_point_depths(swc_path: str, depth: Dict, layers: List[Dict], step_size: float, max_iter: int, output_path: str)	
main()	

neuron_morphology.layered_point_depths.__main__.**translate_field**(*field: xr.DataArray*, *by_x: float*, *by_y: float*, *inplace: bool = False*)

Translate a spatial xarray dataset

Parameters

field [to be translated]

by_x [=the translation along x]

by_y [the translation along y]

inplace [If True, modify this dataarray, otherwise modify a copy]

Returns

translated dataarray, potentiall the same as the input

neuron_morphology.layered_point_depths.__main__.**setup_interpolator**(*field: xr.DataArray, dim: Optional[str], **kwargs*) → RegularGridInterpolator

Build a regular grid interpolator from a dataarray

Parameters

field [Must have dimensions "x" and "y". May have dimension "dim"]

dim [base the interpolator on values from this dim slice. If None, ignore] dim

****kwargs** [passed to interpolator constructor]

Returns

a callable interpolator

neuron_morphology.layered_point_depths.__main__.**containing_layer**(*pos: Tuple[float, float], layers: List[Dict]*) → Optional[str]

Find the layer in which a point is contained

Parameters

pos [the coordinate of the point]

layers [Each has "name" - a string and "bounds" - a Polygon]

Returns

The name of the containing layer or None if no containing layer was found

neuron_morphology.layered_point_depths.__main__.**tuplize**(*arr: np.array*) → Tuple

Convert an array to a tuple

neuron_morphology.layered_point_depths.__main__.**step_from_node**(*pos: Tuple[float, float], depth_interp: RegularGridInterpolator, dx_interp: RegularGridInterpolator, dy_interp: RegularGridInterpolator, surface: LineString, step_size: float, max_iter: int, adaptive_scale: int = 32*) → Optional[float]

Walk through a gradient field, until a defined surface is passed.

Parameters

pos [the start position]

depth_interp [callable mapping positions to scalar depth values]

dx_interp [callable mapping positions to the x component of the gradient]

dy_interp [callable mapping positions to the y component of the gradient]

surface [Check for the intersection of the path with this surface]

step_size [Each step proceeds in the direction of the local gradient,] scaled to this step size

max_iter [give up (return None) if the surface is not intersected in this] many steps

Returns

The depth of the intersection between the path walked and the given surface

neuron_morphology.layered_point_depths.__main__.**get_node_intersections**(*node: Dict, depth_interp: RegularGridInterpolator, dx_interp: RegularGridInterpolator, dy_interp: RegularGridInterpolator, layers: List[Dict], step_size: float, max_iter: int*) → Dict

Given a node, find its layer and intersection depths. Then return a row of LayeredPointDepths for this node.

Parameters

node [Of a Morphology. must have:] "id" - unique identifier "type" - which kind of node is this? "x", "y" - positions in x and y of this node

depth_interp [callable mapping positions to scalar depth values]

dx_interp [callable mapping positions to the x component of the gradient]

dy_interp [callable mapping positions to the y component of the gradient]

layers [Each has] "name" - an identifier "bounds" - a Polygon describing the entire boundary "pia_surface" - a LineString describing the piaward surface of this

layer

"wm_surface" - a LineString describing the white matter-wise surface of this layer

step_size [Each step proceeds in the direction of the local gradient,] scaled to this step size

max_iter [give up (return None) if the surface is not intersected in this] many steps

Returns

A dictionary representing a single row of LayeredPointDepths. Has Keys: "ids" - the identifier of this node "layer_name" - the layer containing this node "depth" - the depth of this node "local_layer_pia_side_depth" - the depth of the intersection between

this node's steepest ascent path and the piaward surface of its containing layer

"local_layer_wm_side_depth" - the depth of the intersection between this node's steepest ascent path and the white matterward surface of its containing layer

"point_type": The type of this node

neuron_morphology.layered_point_depths.__main__.**setup_layers**(*layers: List[Dict]*)

Convert layer bounds, pia, and white matter surfaces to shapely objects

Parameters

layers [Mutated inplace. Has keys:] "bounds" - a Polygon describing the entire boundary "pia_surface" - a LineString describing the piaward surface of this

layer

"wm_surface" - a LineString describing the white matter-wise surface of this layer

neuron_morphology.layered_point_depths.__main__.**run_layered_point_depths**(*swc_path: str, depth: Dict, layers: List[Dict], step_size: float, max_iter: int, output_path: str*)

neuron_morphology.layered_point_depths.__main__.**main**()

neuron_morphology.layered_point_depths._schemas

Module Contents

Classes

DepthField	mm.Schema class with support for making fields default to

Continued on next page

Table 33 – continued from previous page

Layer	mm.Schema class with support for making fields default to
InputParameters	The base marshmallow schema used by ArgSchemaParser to identify
OutputParameters	mm.Schema class with support for making fields default to

class neuron_morphology.layered_point_depths._schemas.**DepthField**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

gradient_field_path

depth_field_path

soma_origin

pia_sign

class neuron_morphology.layered_point_depths._schemas.**Layer**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

name

bounds

pia_surface

wm_surface

class neuron_morphology.layered_point_depths._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

The base marshmallow schema used by ArgSchemaParser to identify input_json and output_json files and the log_level

swc_path

depth

layers

step_size

output_path

max_iter

class neuron_morphology.layered_point_depths._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

output_path

neuron_morphology.pipeline

Submodules

neuron_morphology.pipeline._schemas

Module Contents

Classes

PathResolution	mm.Schema class with support for making fields default to
PrimaryBoundaries	mm.Schema class with support for making fields default to
S3LandingBucket	mm.Schema class with support for making fields default to
InputParameters	The base marshmallow schema used by ArgSchemaParser to identify

class neuron_morphology.pipeline._schemas.**PathResolution**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

path

resolution

class neuron_morphology.pipeline._schemas.**PrimaryBoundaries**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

Soma

White_Matter

Pia

class neuron_morphology.pipeline._schemas.**S3LandingBucket**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

name

region

credentials_file

class neuron_morphology.pipeline._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

The base marshmallow schema used by ArgSchemaParser to identify input_json and output_json files and the log_level

destination_bucket

neuron_reconstruction_id

specimen_id

primary_boundaries

swc_file

cell_depth

cut_thickness

marker_file

ccf_soma_xyz

slice_transform

slice_image_flip

neuron_morphology.pipeline.post_data_to_s3

Module Contents

Functions

get_credentials(credentials_file)	get credentisals from credentials_file
zip_files(file_dict)	zip files into an archive in memory
post_object_to_s3(data, name, bucket, region, access_key_id=None, secret_access_key=None)	post object (bytes) in memory to S3 bucket
main()	Usage:

neuron_morphology.pipeline.post_data_to_s3.**get_credentials**(*credentials_file*)

get credentisals from credentials_file

Parameters

credentials_file: file path to credentials_file

neuron_morphology.pipeline.post_data_to_s3.**zip_files**(*file_dict*)

zip files into an archive in memory

Parameters

file_dict: file name: file paths or file in bytes to be archived

neuron_morphology.pipeline.post_data_to_s3.**post_object_to_s3**(*data*, *name*, *bucket*, *region*, *access_key_id=None*, *secret_access_key=None*)

post object (bytes) in memory to S3 bucket

Parameters

data: the object data

name: the object data's name in s3

region: where the s3 bucket located

bucket: s3 bucket name or arn

access_key_id, secret_access_key: aws user's credentials implicitly the credentials file located in ~/.aws/credentials or set AWS_SHARED_CREDENTIALS_FILE to the credentials file in your environment

neuron_morphology.pipeline.post_data_to_s3.**main**()

Usage: python post_data_to_s3.py –input_json INPUT_JSON

neuron_morphology.snap_polygons

Submodules

neuron_morphology.snap_polygons.__main__

An executable for finding close-fit boundaries between cortical layer polygons.

Module Contents

Functions

run_snap_polygons(layer_polygons, pia_surface, wm_surface, layer_order, working_scale: float, surface_distance_threshold: float, multipolygon_error_threshold: float, images=None)	Finds and returns close fit boundaries. May write diagnostic images as
main()	CLI entrypoint for snapping polygons

neuron_morphology.snap_polygons.__main__.**run_snap_polygons**(*layer_polygons*, *pia_surface*, *wm_surface*, *layer_order*, *working_scale: float*, *surface_distance_threshold: float*, *multipolygon_error_threshold: float*, *images=None*)

Finds and returns close fit boundaries. May write diagnostic images as a side effect.

neuron_morphology.snap_polygons.__main__.**main**()

CLI entrypoint for snapping polygons

neuron_morphology.snap_polygons._from_lims

This module contains utilities for running snap_polygons directly from the Allen Institute's internal Laboratory Information Management System.

Example Usage

python -m neuron_morphology.snap_polygons –host <lims host> –port <lims port> –user <username> –password <password> –database <lims db> –focal_plane_image_series_id 522408212 # for instance –image_output_root /some_directory

Module Contents

Classes

PostgresInputConfigSchema	The parameters required to query a postgres database.
FromLimsSchema	The parameters required to query LIMS for a set of cortical layer
FromLimsSource	An alternate argschema source which gets its inputs from lims directly

Functions

query_for_layer_polygons(query_engine: QueryEngineType, focal_plane_image_series_id: int, validate_polys: bool = True, treatment: str = 'Biocytin') → List[Dict[str, Union[NicePathType, str]]]	Get all layer polygons for this image series
query_for_cortical_surfaces(query_engine: QueryEngineType, focal_plane_image_series_id: int) → Tuple[Dict[str, Union[NicePathType, str]], Dict[str, Union[NicePathType, str]]]	Return the pia and white matter surface drawings for this image series
query_for_images(query_engine: QueryEngineType, focal_plane_image_series_id: int, output_dir: str) → List[Dict[str, str]]	Return Biocytin and DAPI images associated with a focal plane image
query_for_image_dims(query_engine: QueryEngineType, focal_plane_image_series_id: int) → Tuple[float, float]	Find the dimensions of the Biocytin image associated with a focal plane
get_inputs_from_lims(host: str, port: int, database: str, user: str, password: str, imser_id: int, image_output_root: Optional[str])	Utility for building module inputs from a direct LIMS query

neuron_morphology.snap_polygons._from_lims.**QueryEngineType**

neuron_morphology._from_lims.**query_for_layer_polygons**(*query_engine: QueryEngineType*, *focal_plane_image_series_id: int*, *validate_polys: bool = True*, *treatment: str = 'Biocytin'*) → List[Dict[str, Union[NicePathType, str]]]

Get all layer polygons for this image series

Parameters

query_engine [executes a query, passed in as a string. Must not require] any additional database information.

focal_plane_image_series_id [used to determine which polygons to fetch]

validate_polys [if True, fail when]

- a label is associated with multiple distinct valid geometries
- a label is associated with one or more geometries, but none are valid

treatment: The layer polygons are associated with Biocytin and DAPI treatments. We only need one.

Returns

A collection of labelled polygons.

neuron_morphology._from_lims.**query_for_cortical_surfaces**(*query_engine: QueryEngineType*, *focal_plane_image_series_id: int*) → Tuple[Dict[str, Union[NicePathType, str]], Dict[str, Union[NicePathType, str]]]

Return the pia and white matter surface drawings for this image series

neuron_morphology._from_lims.**query_for_images**(*query_engine: QueryEngineType*, *focal_plane_image_series_id: int*, *output_dir: str*) → List[Dict[str, str]]

Return Biocytin and DAPI images associated with a focal plane image series

neuron_morphology.snap_polygons._from_lims.**query_for_image_dims**(*query_engine: QueryEngineType*, *focal_plane_image_series_id: int*) → Tuple[float, float]

Find the dimensions of the Biocytin image associated with a focal plane image series

neuron_morphology.snap_polygons._from_lims.**get_inputs_from_lims**(*host: str, port: int, database: str, user: str, password: str, imser_id: int, image_output_root: Optional[str]*)

Utility for building module inputs from a direct LIMS query

class neuron_morphology.snap_polygons._from_lims.**PostgresInputConfigSchema**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: marshmallow.Schema

The parameters required to query a postgres database.

host

database

user

password

port

class neuron_morphology.snap_polygons._from_lims.**FromLimsSchema**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: *neuron_morphology.snap_polygons._from_lims.PostgresInputConfigSchema*

The parameters required to query LIMS for a set of cortical layer polygons and cortical surface boundaries.

focal_plane_image_series_id

image_output_root

class neuron_morphology.snap_polygons._from_lims.**FromLimsSource**
Bases: argschema.sources.ArgSource

An alternate argschema source which gets its inputs from lims directly

ConfigSchema

get_dict(*self*)

neuron_morphology.snap_polygons._schemas

CLI schemas for the inputs to and outputs from snap_polygons.

Module Contents

Classes

SimpleGeometry	A named planar geometry
Image	A specification for a diagnostic overlay image
InputParameters	Top-level schema for inputs to snap_polygons
OutputImage	Metadata describing an output diagnoctic overlay image.
OutputParameters	Top-level schema for snap_polygons outputs.

class neuron_morphology.snap_polygons._schemas.**SimpleGeometry**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)
Bases: argschema.schemas.DefaultSchema

A named planar geometry

name

path

class neuron_morphology.snap_polygons._schemas.**Image**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)
Bases: argschema.schemas.DefaultSchema

A specification for a diagnostic overlay image

input_path

output_path

downsample

overlay_types

class neuron_morphology.snap_polygons._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

Top-level schema for inputs to snap_polygons

layer_polygons

pia_surface

wm_surface

working_scale

images

layer_order

surface_distance_threshold

multipolygon_error_threshold

class neuron_morphology.snap_polygons._schemas.**OutputImage**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

Metadata describing an output diagnoctic overlay image.

input_path

output_path

downsample

overlay_type

class neuron_morphology.snap_polygons._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

Top-level schema for snap_polygons outputs.

inputs

polygons

surfaces

images

neuron_morphology.snap_polygons.bounding_box

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

Module Contents

Classes

BoundingBox	Represents the bounds of a set of 2D objects

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

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]

__slots__ = ['vorigin', 'horigin', 'vextent', 'hextent']

width :float
Horizontal side length of the box.

height :float
Vertical side length of the box.

aspect_ratio :float
Width / height ratio of the box.

origin
Coordinates of the box's origin.

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

coordinates
Origin and extent coordinates.

__repr__(*self*)
Return repr(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.

transform(*self*, *transform: TransformType*, *inplace: bool = False*)
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)

copy(*self*)
Duplicates this bounding box

Returns

A copy of this object.

round(*self, inplace: bool = False, origin_via: Callable[[float], float] = np.around, extent_via: Callable[[float], float] = np.around*)
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.

neuron_morphology.snap_polygons.cortex_surfaces

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.

Module Contents

Functions

trim_to_close(geometry: BaseGeometry, threshold: float, linestring: LineType, iterations: int = 10) → LineString	Find the longest segment of a linestring whose endpoints are within a

Continued on next page

Table 41 – continued from previous page

find_transition(unmet: Point, met: Point, condition: ConditionFn, iterations: int) → Point	Given two points in space, one of which meets a condition, locate the
first_met(coords: Sequence[Union[Point, Tuple]], condition: ConditionFn, iterations: int) → Tuple[int, Point]	Locate the first point along a coordinate sequence at which a condition
remove_duplicates(coords: Sequence[Point]) → Sequence[Point]	Remove duplicate points from a coordinate sequence.
trim_coords(coords: Sequence[Union[Point, Tuple]], condition: ConditionFn, iterations: int)	Find the longest subinterval of a coordinate sequence whose endpoints

neuron_morphology.snap_polygons.cortex_surfaces.**ConditionFn**

neuron_morphology.snap_polygons.cortex_surfaces.**trim_to_close**(*geometry: BaseGeometry*, *threshold: float*, *linestring: LineType*, *iterations: int = 10*) → 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.cortex_surfaces.**find_transition**(*unmet: Point*, *met: Point*, *condition: ConditionFn*, *iterations: int*) → 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[Point, Tuple]]*, *condition: ConditionFn*, *iterations: int*) → Tuple[int, 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[Point]*) → Sequence[Point]

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[Point, Tuple]]*, *condition: ConditionFn*, *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.geometries

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

Module Contents

Classes

Geometries	A collection of polygons and lines

Functions

select_largest_subpolygon(polygons: Union[Polygon, Iterable[Polygon]], error_threshold: float) → Polygon	Given a collection of polygons, find the largest by area.
safe_linemerge(linestrings: Union[LineString, Sequence[LineString]]) → LineString	Wrapper around shapely.ops.linemerge that no-ops in case a single
rasterize(geometry: shapely.geometry.base.BaseGeometry, box: BoundingBox) → np.array	Rasterize a shapely object to a grid defined by a provided bounding box.
make_translation(horizontal: float, vertical: float) → TransformType	Utility for building a 2D translation transform
make_scale(scale: float = 1.0) → TransformType	A utility for making a 2D scale transform, suitable for transforming
clear_overlaps(stack: Dict[str, np.ndarray])	Given a stack of masks, remove all inter-mask overlaps inplace
closest_from_stack(stack: Dict[str, np.ndarray])	Given a stack of images describing distance from several objects, find
get_snapped_polys(closest: np.ndarray, name_lut: Dict[int, str], multipolygon_resolver: MultiPolygonResolverType) → Dict[str, Polygon]	Obtains named shapes from a label image.
find_vertical_surfaces(polygons: Dict[str, Polygon], order: Sequence[str], pia: Optional[LineString] = None, white_matter: Optional[LineString] = None)	Given a set of polygons describing cortical layer boundaries, find the
shared_faces(poly: Polygon, others: Iterable[Polygon], snap_tolerance=10) → LineString	Given a polygon and a set of other polygons that could be adjacent on

neuron_morphology.snap_polygons.geometries.**select_largest_subpolygon**(*polygons: Union[Polygon, Iterable[Polygon]], error_threshold: float*) → Polygon

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.**safe_linemerge**(*linestrings: Union[LineString, Sequence[LineString]]*) → LineString

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

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

A collection of polygons and lines

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).

close_bounds

The smallest bounding box enclosing these geometries.

register_polygon(*self*, *name: str*, *path: PolyType*)

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_many(*self, objects: Union[Dict[str, Union[LineType, PolyType]], Sequence[Dict[str, Union[LineType, PolyType]]]], method: Callable[[str, Union[LineType, PolyType]], None]*)

Utility for registering many polygons or surfaces. See register_polygons and register_surfaces for use.

register_polygons(*self, polygons: Union[Dict[str, PolyType], Sequence[Dict[str, PolyType]]]*)

utility for registering multiple polygons. See register_polygon

register_surface(*self*, *name: str*, *path: LineType*)

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, LineType]*)

utility for registering multiple surfaces. See register_surface

rasterize(*self, box: Optional[BoundingBox] = None, polygons: Union[Sequence[str], bool] = True, surfaces: Union[Sequence[str], bool] = False*)

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

transform(*self*, *transform: TransformType*)
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.

fill_gaps(*self*, *working_scale: float = 1.0*, *multipolygon_resolver: Optional[MultiPolygonResolverType] = None*)
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.

cut(*self*, *template: shapely.geometry.Polygon*, *multipolygon_resolver: Optional[MultiPolygonResolverType] = None*, *multisurface_resolver: Optional[MultiSurfaceResolvertype] = None*)
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

convex_hull(*self*, *surfaces: bool = True*, *polygons: bool = True*)
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

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

neuron_morphology.snap_polygons.geometries.**rasterize**(*geometry: shapely.geometry.base.BaseGeometry*, *box: BoundingBox*) → np.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.**make_translation**(*horizontal: float*, *vertical: float*) → TransformType
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.**make_scale**(*scale: float = 1.0*) → TransformType
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.**clear_overlaps**(*stack: Dict[str, np.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, np.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.**get_snapped_polys**(*closest: np.ndarray, name_lut: Dict[int, str], multipolygon_resolver: MultiPolygonResolverType*) → Dict[str, 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.**find_vertical_surfaces**(*polygons: Dict[str, Polygon], order: Sequence[str], pia: Optional[LineString] = None, white_matter: Optional[LineString] = 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.**shared_faces**(*poly: Polygon, others: Iterable[Polygon], snap_tolerance=10*) → 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

Utilites for writing diagnostic overlay images

Module Contents

Classes

ImageOutputter	Overlays polygons and surfaces on provided images. Writes the

Functions

write_figure(fig: plt.Figure, *args, **kwargs)	Write a matplotlib figure without respect to the current figure.
read_image(path: str, decimate: int = 1)	Read an image. Dispatch to an appropriate library based on that
read_with_ndimage(path: str, decimate: int)	Read (and symmetrically decimate) an image file into a numpy array
read_jp2(path: str, decimate: int)	Read (and symmetrically decimate) a jp2 file into a numpy array
fname_suffix(path: str, suffix: str)	Utility for adding a suffix to a path string. The suffix will be
make_pathpatch(vertices: Sequence[Tuple[float, float]], **patch_kwargs) → mpl.patches.PathPatch	Utility for building a matplotlib pathpatch from an array of vertices

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

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.

DEFAULT_COLOR_CYCLE = ['c', 'm', 'y', 'k', 'r', 'g', 'b']

OVERLAY_TYPES

_draw_geometries(*self*, *geometries: Geometries*, *image: np.ndarray*)

Utility for overlaying polygons and surfaces on an image. See draw_before and draw_after for more details.

draw_before(*self*, *image: np.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

draw_after(*self*, *image: np.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

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

Parameters

fig [the figure to be written]

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

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_with_ndimage**(*path: str*, *decimate: int*)
Read (and symmetrically decimate) an image file into a numpy array

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.**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*) → mpl.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.types

Module Contents

Functions

ensure_polygon(candidate: PolyType) → Polygon	Convert from one of many polygon representations to Polygon
ensure_linestring(candidate: LineType) → LineString	Convert from one of many line representations to LineString
ensure_path(candidate: PathType, num_dims: int = 2) → NicePathType	Ensure that an input path, which might be a "x,y,x,y" string, is
split_pathstring(pathstring: str, num_dims: int = 2, sep: str = ',') → NicePathType	Converts a pathstring ("x,y,x,y…") to a num_points X num_dims

neuron_morphology.snap_polygons.types.**NicePathType**

neuron_morphology.snap_polygons.types.**PathType**

neuron_morphology.snap_polygons.types.**PathsType**

neuron_morphology.snap_polygons.types.**PolyType**

neuron_morphology.snap_polygons.types.**LineType**

neuron_morphology.snap_polygons.types.**TransformType**

neuron_morphology.snap_polygons.types.**MultiPolygonResolverType**

neuron_morphology.snap_polygons.types.**MultiSurfaceResolvertype**

neuron_morphology.snap_polygons.types.**ensure_polygon**(*candidate: PolyType*) → Polygon

Convert from one of many polygon representations to Polygon

neuron_morphology.snap_polygons.types.**ensure_linestring**(*candidate: LineType*) → LineString

Convert from one of many line representations to LineString

neuron_morphology.snap_polygons.types.**ensure_path**(*candidate: PathType*, *num_dims: int = 2*) → NicePathType

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.**split_pathstring**(*pathstring: str*, *num_dims: int = 2*, *sep: str = ','*) → NicePathType

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

neuron_morphology.transforms

Subpackages

neuron_morphology.transforms.affine_transformer

Submodules

neuron_morphology.transforms.affine_transformer._schemas

Module Contents

Classes

AffineDictSchema	affine_dict: keys and values corresponding to the following
ApplyAffineSchema	Arg Schema for apply_affine_transform module
OutputParameters	mm.Schema class with support for making fields default to

Functions

validate_input_affine(data)	

neuron_morphology.transforms.affine_transformer._schemas.**validate_input_affine**(*data*)

class neuron_morphology.transforms.affine_transformer._schemas.**AffineDictSchema**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

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]]

tvr_00

tvr_01

tvr_02

tvr_03

tvr_04

tvr_05

tvr_06

tvr_07

tvr_08

tvr_09

tvr_10

tvr_11

class neuron_morphology.transforms.affine_transformer._schemas.**ApplyAffineSchema**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.ArgSchema

Arg Schema for apply_affine_transform module

affine_dict

affine_list

input_swc

output_swc

validate_schema_input(*self*, *data*, ***kwargs*)

class neuron_morphology.transforms.affine_transformer._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

transformed_swc

neuron_morphology.transforms.affine_transformer.apply_affine_transform

Module Contents

Functions

main()	

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

neuron_morphology.transforms.pia_wm_streamlines

Submodules

neuron_morphology.transforms.pia_wm_streamlines._schemas

Module Contents

Classes

PiaWmStreamlineSchema	Arg Schema for run_pia_wm_streamlines
OutputParameters	mm.Schema class with support for making fields default to

class neuron_morphology.transforms.pia_wm_streamlines._schemas.**PiaWmStreamlineSchema**(*only=No ex- clude=(many=F con- text=No load_on dump_o par- tial=Fal un- known=*

Bases: argschema.ArgSchema

Arg Schema for run_pia_wm_streamlines

pia_path_str

wm_path_str

soma_path_str

resolution

pia_fixed_value

wm_fixed_value

mesh_res

output_dir

class neuron_morphology.transforms.pia_wm_streamlines._schemas.**OutputParameters**(*only=None, ex- clude=(), many=False, con- text=None, load_only=(), dump_only=(), par- tial=False, un- known=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

depth_field_file

gradient_field_file

translation

neuron_morphology.transforms.pia_wm_streamlines.calculate_pia_wm_streamlines

Module Contents

Functions

convert_path_str_to_list(path_str: str, resolution: float = 1.0) → List[Tuple[float, float]]	
run_streamlines(pia_path_str: str, wm_path_str: str, resolution: float, soma_path_str: Optional[str] = None, mesh_res: int = 20, pia_fixed_value: float = 1.0, wm_fixed_value: float = 0.0)	
main()	

neuron_morphology.transforms.pia_wm_streamlines.calculate_pia_wm_streamlines.**convert_path_s**

neuron_morphology.transforms.pia_wm_streamlines.calculate_pia_wm_streamlines.**run_streamline**

neuron_morphology.transforms.pia_wm_streamlines.calculate_pia_wm_streamlines.**main**()

neuron_morphology.transforms.scale_correction

Submodules

neuron_morphology.transforms.scale_correction._schemas

Module Contents

Classes

InputParameters	The base marshmallow schema used by ArgSchema-Parser to identify
OutputParameters	mm.Schema class with support for making fields default to

class neuron_morphology.transforms.scale_correction._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

The base marshmallow schema used by ArgSchemaParser to identify input_json and output_json files and the log_level

swc_path

marker_path

soma_depth

cut_thickness

class neuron_morphology.transforms.scale_correction._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

scale_correction

scale_transform

neuron_morphology.transforms.scale_correction.compute_scale_correction

Module Contents

Functions

estimate_scale_correction(morphology: Morphology, soma_depth: float, soma_marker_z: float, cut_thickness: Optional[float] = 350)	Estimate a scale factor to correct the reconstructed morphology
get_soma_marker_from_marker_file(marker_path: str)	
run_scale_correction(morphology: Morphology, soma_marker_z: float, soma_depth: float, cut_thickness: float)	
collect_inputs(args: Dict[str, Any]) → Dict[str, Any]	**Parameters**
main()	

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.**run_scale_correctio**

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.**main**()

neuron_morphology.transforms.tilt_correction

Submodules

neuron_morphology.transforms.tilt_correction._schemas

Module Contents

Classes

InputParameters	The base marshmallow schema used by ArgSchema-Parser to identify
OutputParameters	mm.Schema class with support for making fields default to

Functions

validate_input_affine(data)

neuron_morphology.transforms.tilt_correction._schemas.**validate_input_affine**(*data*)

class neuron_morphology.transforms.tilt_correction._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

The base marshmallow schema used by ArgSchemaParser to identify input_json and output_json files and the log_level

swc_path

marker_path

slice_image_flip

ccf_soma_location

slice_transform_list

slice_transform_dict

ccf_path

validate_schema_input(*self*, *data*, ***kwargs*)

class neuron_morphology.transforms.tilt_correction._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

tilt_correction

tilt_transform_dict

neuron_morphology.transforms.tilt_correction.compute_tilt_correction

Module Contents

Functions

get_tilt_correction(morphology: Morphology, soma_voxel: List[int], slice_angle_matrix: float, closest_path)	Find the tilt angle between the slice plane and the nearest streamline
find_closest_path(soma_voxel: List[int], ccf_path: Union[str, IO], n_sublists: int = 2)	Finds closest path to soma_voxel
determine_slice_flip(morphology: Morphology, soma_marker: Dict, slice_image_flip: bool)	Determines whether the tilt correction should be positive or negative
read_soma_marker(marker_path: str)	
run_tilt_correction(morphology: Morphology, soma_marker: Dict, ccf_soma_location: Dict, slice_transform: aff.AffineTransform, slice_image_flip: bool, ccf_path: Union[str, IO])	
main()	

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**CCF_SHAPE = [1320, 80**

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**CCF_RESOLUTION = 10**

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**get_tilt_correction**(*m*
M
ph
og
so
Li
sli
flo
cl
es

Find the tilt angle between the slice plane and the nearest streamline

Parameters

morphology: Morphology object

soma_voxel: soma voxel in ccf {'x': , 'y': , 'z': }

slice_angle_matrix: 4 x 4 affine matrix of the slice plane relative to ccf

closest_path: 3 x N array of voxel coordinates in closest streamline, only first (wm end) and last (pia end) coordinates are used. In future the tilt correction may be refined to use the entire path.

Returns

tilt angle correction (radians)

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**find_closest_path**(*soma*
List[
ccf_p
Unio
IO],
n_su
int
=
2)

Finds closest path to soma_voxel

Parameters

soma_voxel: List containing soma voxel ccf coordinates

ccf_path: str, FilePath, or File-like object openable by H5PY

n_sublists: will seperate path_ids into n sublists to load in at a time. Higher values decrease memory usage but increase processing time. n = 1 uses about 16GB

Returns

closest_path: array of voxel coordinates of the closest streamline

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**determine_slice_flip**(*m*
l
l
c
s
l
s
l

Determines whether the tilt correction should be positive or negative

Parameters

morphology: Morphology object

soma_marker: soma marker dictionary from reconstruction marker file

slice_image_flip: indicates whether the image was flipped relative to the slice (e.g the z axis of the image is opposite to the z axis in the slice)

Returns

flip_toggle -1 or 1 to be multiplied against tilt correction

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**read_soma_marker**(*marke*
str)

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**run_tilt_correction**(*m*
M
ph
og
so
D
co
D
sl
af
sl
bo
co
U
IO

neuron_morphology.transforms.tilt_correction.compute_tilt_correction.**main**()

neuron_morphology.transforms.upright_angle

Submodules

neuron_morphology.transforms.upright_angle._schemas

Module Contents

Classes

InputParameters	The base marshmallow schema used by ArgSchema-Parser to identify
OutputParameters	mm.Schema class with support for making fields default to

Functions

validate_neighbors(num)	

neuron_morphology.transforms.upright_angle._schemas.**validate_neighbors**(*num*)

class neuron_morphology.transforms.upright_angle._schemas.**InputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.ArgSchema

The base marshmallow schema used by ArgSchemaParser to identify input_json and output_json files and the log_level

gradient_path

node

step

neighbors

swc_path

validate_schema_input(*self*, *data*, ***kwargs*)

class neuron_morphology.transforms.upright_angle._schemas.**OutputParameters**(*only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Bases: argschema.schemas.DefaultSchema

mm.Schema class with support for making fields default to values defined by that field's arguments.

inputs

upright_angle

upright_transform_dict

neuron_morphology.transforms.upright_angle.compute_angle

Module Contents

Functions

get_upright_angle(gradient: xr.DataArray, point: Optional[List[float]] = None, n_win: int = 2) → float	Calculate the upright angle at a position, e.g. soma, given a vector field
calculate_transform(gradient_field: xr.DataArray, morph: Morphology, node: Optional[List[float]] = None)	
run_upright_angle(gradient_path: str, swc_path: str, node: Optional[List[float]] = None)	
main()	

neuron_morphology.transforms.upright_angle.compute_angle.**get_upright_angle**(*gradient: xr.DataArray*, *point: Optional[List[float]] = None*, *n_win: int = 2*) → float

Calculate the upright angle at a position, e.g. soma, given a vector field

Parameters

gradient: xarray of the the vector field

point: list [x,y,z] coordinates

n_win: number of grid points to define the interpolation window

Returns

angle

neuron_morphology.transforms.upright_angle.compute_angle.**calculate_transform**(*gradient_field: xr.DataArray*, *morph: Morphology*, *node: Optional[List[float]] = None*)

neuron_morphology.transforms.upright_angle.compute_angle.**run_upright_angle**(*gradient_path: str*, *swc_path: str*, *node: Optional[List[float]] = None*)

neuron_morphology.transforms.upright_angle.compute_angle.**main**()

Submodules

neuron_morphology.transforms.affine_transform

Module Contents

Classes

AffineTransform	Handles transformations to a pia/wm aligned coordinate frame.

Functions

affine_from_transform_translation(transform: Optional[Any] = None, translation: Optional[Any] = None, translate_first: bool = False)	Create affine from linear transformation and translation.
rotation_from_angle(angle: float, axis: int = 2)	Create an affine matrix from a rotation about a specific axis.
affine_from_translation(translation: Any)	Create an affine translation.
affine_from_transform(transform: Any)	Create affine transformation.

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.

classmethod from_dict(*cls, 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 (*cls, 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*)
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*)
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*)
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

_get_scaling_factor (*self*)
Calculate the scaling factor from the affine matrix.

Returns

Scaling factor: 3rd root of the determinant.

transform_morphology (*self*, *morphology: Morphology*, *clone: bool = False*, *scale_radius: bool = True*)
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_translation**(*transform: Optional[Any] = None*, *translation: Optional[Any] = 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.**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.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.**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.geometry

Some handy utilities for working with vector geometries

Module Contents

Functions

get_ccw_vertices_from_two_lines(line1: List[Tuple], line2: List[Tuple])	Convenience method two do both get_vertices_from_two_lines()
prune_two_lines(line1: List[Tuple], line2: List[Tuple])	check the boundary to avoid intersections with side lines
get_vertices_from_two_lines(line1: List[Tuple], line2: List[Tuple])	Generates circular vertices from two lines
get_ccw_vertices(vertices: List[Tuple])	Generates counter clockwise vertices from vertices describing

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.**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.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.**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.streamline

Module Contents

Functions

solve_laplace_2d(V: fem.FunctionSpace, bcs: List[fem.bcs.DirichletBCMetaClass])	Solves the laplace equation with boundary conditions bcs on V
compute_gradient(uh, W, bcs=[])	
generate_laplace_field(top_line: List[Tuple], bottom_line: List[Tuple], mesh_res: float = 20, top_value: float = 1.0, bottom_value: float = 0.0, eps_bounds: float = 1e-08)	Solve Laplace equation inside a polygon bounded

neuron_morphology.transforms.streamline.**solve_laplace_2d**(*V: fem.FunctionSpace, bcs: List[fem.bcs.DirichletBCMetaClass]*)

Solves the laplace equation with boundary conditions bcs on V

Parameters

V: Fenics FunctionSpace object created from a mesh

bcs: List of Fenics DirichletBC Boundary Conditions

neuron_morphology.transforms.streamline.**compute_gradient**(*uh*, *W*, *bcs=[]*)

neuron_morphology.transforms.streamline.**generate_laplace_field**(*top_line: List[Tuple], bottom_line: List[Tuple], mesh_res: float = 20, top_value: float = 1.0, bottom_value: float = 0.0, eps_bounds: float = 1e-08*)

Solve Laplace equation inside a polygon bounded by two lines (top_line and bottom_line) and the artificial straight side lines connecting the ends of the two lines. Apply Dirichlet BC on top_line and bottom_line and zero Neuman BC on the side lines.

Demo of fenics: https://github.com/hplgit/fenics-tutorial/ blob/master/pub/python/vol1/ft01_poisson.py

If top_value and bottom_value defaults are used, value_field will be the normalized distance to the top_line

Parameters

top_line: line that will have top_value boundary condition

bottom_line: line that will have bottom_value boundary condition

mesh_res: resolution of the mesh

top_value: value for top Dirichlet Boundary

bottom_value: value for bottom Dirichlet Boundary

Returns

u: returns value at input point e.g. 0.5 = u((0.5, 0.5))

grad_u: returns gradient at input point e.g. [0, 1] = u((0.5, 0.5))

mesh_coords: coordinates of each vertex in the mesh

mesh_values: values at each vertex in the mesh

gradient_mesh: gradient at each vertex in the mesh

neuron_morphology.transforms.transform_base

Module Contents

Classes

TransformBase	Abstract base class for implementing swc transforms.

class neuron_morphology.transforms.transform_base.**TransformBase**

Bases: abc.ABC

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

transform_morphology(*self*)

Apply this transform to all nodes in a morphology.

Returns

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

neuron_morphology.validation

Submodules

neuron_morphology.validation.bits_validation

Module Contents

Functions

validate_independent_axon_has_more_than_four_nodes(morphology)	This function checks if an independent (parent is -1)
validate_types_three_four_traceable_back_to_soma(morphology)	This function checks if types 3,4 are traceable
validate(morphology)	

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

neuron_morphology.validation.marker_validation

Module Contents

Functions

validate_coordinates_corresponding_to_dendrite(marker_file, morphology)	This function checks whether the coordinates for each dendrite marker
validate_coordinates_corresponding_to_axon(marker_file, morphology)	This function checks whether the coordinates for each axon marker
validate_expected_name(marker_file)	This function checks whether the markers have the expected types
validate_type_thirty_count(marker_file)	This function checks whether there is exactly one type 30 in the file
validate_no_reconstruction_count(marker_file)	This function checks whether there is exactly one type 20 in the file
validate(marker_file, 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_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_expected_name**(*marker_file*)
This function checks whether the markers have the expected types

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.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**(*marker_file*, *morphology*)

neuron_morphology.validation.morphology_statistics

Module Contents

Functions

count_number_of_independent_axons(morphology)	This functions counts the number of independent axons (parent is -1)
morphology_statistics(morphology)	

neuron_morphology.validation.morphology_statistics.**count_number_of_independent_axons**(*morpholo*
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 Contents

Functions

validate_radius_threshold(morphology)	This function validates the radius for types 1, 3, and 4
validate_extreme_taper(morphology)	This function checks whether there is an extreme taper.
validate_radius_has_negative_slope_dendrite(morphology, dendrite)	This function checks whether the radius for dendrite nodes decreases

Continued on next page

Table 68 – continued from previous page

slope_linear_regression_branch_order_avg_radius(morphology, avg_radius)	Use linear regression to find the slope of the best fit line
validate_constrictions(morphology)	This function checks if the radius of basal dendrite and apical dendrite
validate(morphology)	

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

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.**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_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**(*morphology*)

neuron_morphology.validation.report

Module Contents

Classes

Report

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

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

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

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

to_json(*self*)

has_results(*self*)

neuron_morphology.validation.resample_validation

Module Contents

Functions

validate_distance_between_connected_nodes(morphology)	
validate(morphology)	

neuron_morphology.validation.resample_validation.**validate_distance_between_connected_nodes**

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

neuron_morphology.validation.result

Module Contents

Classes

NodeValidationError	
MarkerValidationError	

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

Bases: object

message

node_ids

level

__repr__(*self*)
Return repr(self).

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

Bases: object

message

marker

level

__repr__(*self*)
Return repr(self).

exception neuron_morphology.validation.result.**InvalidMorphology**(*validation_errors*)

Bases: ValueError

Inappropriate argument value (of correct type).

validation_errors

__str__(*self*)
Return str(self).

exception neuron_morphology.validation.result.**InvalidMarkerFile**(*validation_errors*)

Bases: ValueError

Inappropriate argument value (of correct type).

validation_errors

__str__(*self*)
Return str(self).

neuron_morphology.validation.structure_validation

Module Contents

Functions

validate_children_nodes_appear_before_parent_nodes(morphology)	
validate(morphology)	

neuron_morphology.validation.structure_validation.**validate_children_nodes_appear_before_pa**

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

neuron_morphology.validation.type_validation

Module Contents

Functions

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
validate_number_of_soma_nodes(morphology)	This function validates the number of type 1 nodes
validate_expected_types(morphology)	This function validates the expected types of the nodes
valid_dendrite_parent(morphology, node, valid_parent_type)	
validate_node_parent(morphology)	This function validates the type of parent node for a specific type of child node
validate_immediate_children_of_soma_cannot_branch(morphology)	This function validates that immediate children of soma cannot branch
validate_multiple_axon_initiation_points(morphology)	This function validates that the parent of axon (either type 1 or 3) only happens once
validate(morphology)	

neuron_morphology.validation.type_validation.**valid_types**

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_number_of_soma_nodes**(*morphology*)
This function validates the number of type 1 nodes

neuron_morphology.validation.type_validation.**validate_expected_types**(*morphology*)
This function validates the expected types of the nodes

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

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_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**(*morphology*)

neuron_morphology.validation.validate_reconstruction

Module Contents

Functions

parse_arguments(args)	This function parses command line arguments
main()	

neuron_morphology.validation.validate_reconstruction.**logger**

neuron_morphology.validation.validate_reconstruction.**parse_arguments**(*args*)
This function parses command line arguments

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

Package Contents

Functions

validate_morphology(morphology)	
validate_marker(marker, morphology)	

neuron_morphology.validation.**swc_validators**

neuron_morphology.validation.**marker_validators**

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

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

neuron_morphology.vis

Submodules

neuron_morphology.vis.morphovis

Module Contents

Functions

plot_morphology_xy(morphology, ax)	
plot_morphology_zy(morphology, ax)	
plot_cortical_boundary(pia_coords, wm_coords, ax)	
plot_soma(soma_center, ax)	
plot_depth_field(depth_field, ax)	
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_cortical_boundary**(*pia_coords*, *wm_coords*, *ax*)

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

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

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

4.1.2 Submodules

neuron_morphology.constants

Module Contents

neuron_morphology.constants.**SOMA = 1**

neuron_morphology.constants.**AXON = 2**

neuron_morphology.constants.**BASAL_DENDRITE = 3**

neuron_morphology.constants.**APICAL_DENDRITE = 4**

neuron_morphology.constants.**CUT_DENDRITE = 10**

neuron_morphology.constants.**NO_RECONSTRUCTION = 20**

neuron_morphology.constants.**TYPE_30 = 30**

neuron_morphology.constants.**SPACING = [0.1144, 0.1144, 0.28]**

neuron_morphology.lims_apical_queries

Module Contents

Functions

convert_coords_str(coords_str: str, resolution=None)	Convert a comma seperated string of coordinate pairs
get_data(query)	
get_all_intact_apical()	
get_benchmark_apical()	

neuron_morphology.lims_apical_queries.**convert_coords_str**(*coords_str: str*, *resolution=None*)

Convert a comma seperated string of coordinate pairs

neuron_morphology.lims_apical_queries.**get_data**(*query*)

neuron_morphology.lims_apical_queries.**get_all_intact_apical**()

neuron_morphology.lims_apical_queries.**get_benchmark_apical**()

neuron_morphology.marker

Module Contents

Classes

Marker	Simple dictionary class for handling reconstruction marker objects.

Functions

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

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

Bases: dict

Simple dictionary class for handling reconstruction marker objects.

neuron_morphology.marker.**read_marker_file**(*file_name*)

read in a marker file and return a list of dictionaries

neuron_morphology.morphology

Module Contents

Classes

Morphology	

class neuron_morphology.morphology.**Morphology**(*nodes*, *node_id_cb*, *parent_id_cb*)

Bases: allensdk.core.simple_tree.SimpleTree

__len__ (*self*)

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

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

children_of (*self*, *node*)

parent_of (*self*, *node*)

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

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

node_by_id (*self*, *node_id*)

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_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_roots (*self*)

get_root_id (*self*)

get_roots_for_nodes (*self*, *nodes*)

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

get_tree_list (*self*)

get_root_for_tree (*self*, *tree_number*)

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

has_type (*self*, *node_type*)

get_non_soma_nodes (*self*)

get_max_id (*self*)

is_soma_child (*self*, *node*)

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

_build_segment (*self*, *end_node*)

is_node_at_beginning_of_segment (*self*, *node*)

is_node_at_end_of_segment (*self*, *node*)

get_segment_length (*self*, *segment*)

get_branch_order_for_node (*self*, *node*)

get_branch_order_for_segment (*self*, *segment*)

_create_compartment_dictionary (*self*)

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

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

get_compartment_length (*self*, *compartment*)

get_compartment_surface_area (*self, compartment: Sequence[Dict]*)
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]*)
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_compartment_midpoint (*self*, *compartment*)

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

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

clone(*self*)

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

_insert_between(*self*, *new_node*, *parent_id*, *child_id*, *set_parent_id_cb*)

_make_and_insert_intermediate(*self*, *make_intermediates_cb*, *set_parent_id_cb*, *child*)

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

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

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

_get_edge_and_merge(*self*, *merge_cb*, *new_nodes*, *node*)
Used by swap_nodes_edges

static _get_node_attributes(*attributes*, *nodes*)

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

static euclidean_distance(*node1*, *node2*)

static midpoint(*node1*, *node2*)

neuron_morphology.morphology_builder

Module Contents

Classes

MorphologyBuilder

class neuron_morphology.morphology_builder.**MorphologyBuilder**

next_id

parent_id

active_node_id

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

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

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.

_add_node(*self*, *x*, *y*, *z*, *node_type*, *radius*)
Add a new node to this morphology.

Parameters

x [x coordinate of this node's position]

y [y coordinate of this node's position]

z [z coordinate of this node's position]

node_type [one of AXON, SOMA, APICAL_DENDRITE, BASAL_DENDRITE]

radius [describe the size of this node]

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

apical_dendrite (*self*, *x=None*, *y=None*, *z=None*, *radius=1*)
Convenvience for creating an apical dendrite 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.

neuron_morphology.swc_io

Module Contents

Functions

read_swc(path, columns=SWC_COLUMNS, sep=' ', casts=COLUMN_CASTS)	Read an swc file into a pandas dataframe
write_swc(data, path, comments=None, sep=' ', columns=SWC_COLUMNS, casts=COLUMN_CASTS)	Write an swc file from a pandas dataframe
apply_casts(df, casts)	
morphology_from_swc(swc_path)	
morphology_to_swc(morphology, swc_path, comments=None)	Write an swc file from a morphology object

neuron_morphology.swc_io.**SWC_COLUMNS = ['id', 'type', 'x', 'y', 'z', 'radius', 'parent']**

neuron_morphology.swc_io.**COLUMN_CASTS**

neuron_morphology.swc_io.**read_swc** (*path*, *columns=SWC_COLUMNS*, *sep=' '*, *casts=COLUMN_CASTS*)
Read an swc file into a pandas dataframe

neuron_morphology.swc_io.**write_swc** (*data*, *path*, *comments=None*, *sep=' '*, *columns=SWC_COLUMNS*, *casts=COLUMN_CASTS*)
Write an swc file from a pandas dataframe

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

4.1.3 Package Contents

neuron_morphology.**version_path**

neuron_morphology.**__version__**

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

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z