libNeuroML Documentation

Release 0.2.58

libNeuroML authors and contributors

Oct 29, 2021

CONTENTS

Welcome to the libNeuroML documentation. Here you will find information on installing, using, and contributing to libNeuroML. For more information on NeuroML standard, other tools in the NeuroML eco-system, the NeuroML community and how to get in touch with us, please see the documentation at https://docs.neuroml.org.

CHAPTER

ONE

USER GUIDE

1.1 Introduction

This package provides Python libNeuroML, for working with neuronal models specified in NeuroML 2.

Warning: libNeuroML targets NeuroML v2.0

libNeuroML targets NeuroML v2.0, which is described in Cannon et al, 2014). NeuroML v1.8.1 (Gleeson et al. 2010) is now deprecated and not supported by libNeuroML.

For a detailed description of libNeuroML see Vella *et al.* [VCC+14]. *Please cite the paper if you use libNeuroML.*

1.1.1 NeuroML

NeuroML provides an object model for describing neuronal morphologies, ion channels, synapses and 3D network structure. For more information on NeuroML 2 and LEMS please see the NeuroML documentation.

1.1.2 Serialisations

The XML serialisation will be the "natural" serialisation and will follow closely the NeuroML object model. The format of the XML will be specified by the XML Schema definition (XSD file).

Other serialisations have been developed (HDF5, JSON, SWC). Please see Vella *et al.* [VCC+14] for more details.

1.2 Installation

1.2.1 Using Pip

On most systems with a Python installation, libNeuroML can be installed using the default Python package manager, Pip:

```
pip install libNeuroML
```

It is recommended to use a virtual environment when installing Python packages using *pip* to prevent these from conflicting with other system libraries.

This will support the default XML serialization. To install all of requirements to include the other serialisations, use

```
# On Ubuntu based systems
sudo apt-get install libhdf5-dev
pip install libNeuroML[full]
```

The `apt` line is required at time of writing because PyTables' wheels for python 3.7 depend on the system libhdf5.

1.2.2 On Fedora based systems

On Fedora Linux systems, the NeuroFedora community provides libNeuroML in the standard Fedora repos and can be installed using the following commands:

```
sudo dnf install python3-libNeuroML
```

1.2.3 Install from source

You can clone the GitHub repository and also build libNeuroML from the sources. For this, you will need git:

```
git clone git://github.com/NeuralEnsemble/libNeuroML.git
cd libNeuroML
```

More details about the git repository and making your own branch/fork are here. To build and install libNeuroML, you can use the standard install method for Python packages (preferably in a virtual environment):

```
python setup.py install
```

To use the **latest development version of libNeuroML**, switch to the development branch:

```
git checkout development
sudo python setup.py install
```

1.2.4 Run an example

Some sample scripts are included in *neuroml/examples*, e.g. :

```
cd neuroml/examples
python build_network.py
```

The standard examples can also be found *Examples*.

1.2.5 Unit tests

To run unit tests cd to the directory *neuroml/test* and use the Python unittest module discover method:

```
cd neuroml/test/
python -m unittest discover
```

If all tests passed correctly, your output should look something like this:

```
.......
----------------------------------------------------------------------
Ran 55 tests in 40.1s

OK
```

You can also use PyTest to run tests.

```
pip install pytest
pytest -v --strict -W all
```

To ignore some tests, like the MongoDB test which requires a MongoDB setup, run:

```
pytest -v -k "not mongodb" --strict -W all
```

1.3 API documentation

The libNeuroML API includes the core NeuroML classes and various utilities. You can find information on these in the pages below.

1.3.1 `nml` Module (NeuroML Core classes)

These NeuroML core classes are Python representations of the Component Types defined in the NeuroML standard . These can be used to build NeuroML models in Python, and these models can then be exported to the standard XML NeuroML representation. These core classes also contain some utility functions to make it easier for users to carry out common tasks.

Each NeuroML Component Type is represented here as a Python class. Due to implementation limitations, whereas NeuroML Component Types use lower camel case naming, the Python classes here use upper camel case naming. So, for example, the `adExIaFCell` Component Type in the NeuroML schema becomes the `AdExIaFCell` class here, and `expTwoSynapse` becomes the `ExpTwoSynapse` class.

The `child` and `children` elements that NeuroML Component Types can have are represented in the Python classes as variables. The variable names, to distinguish them from class names, use snake case. So for example, the `cell` NeuroML Component Type has a corresponding `Cell` Python class here. The `biophysicalProperties` child Component Type in `cell` is represented as the `biophysical_properties` list variable in the `Cell` Python class. The class signatures list all the child/children elements and text fields that the corresponding Component Type possesses. To again use the `Cell` class as an example, the construction signature is this:

```
class neuroml.nml.nml.Cell(neuro_lex_id=None, id=None, metaid=None, notes=None,␣
↪properties=None, annotation=None, morphology_attr=None, biophysical_properties_
↪attr=None, morphology=None, biophysical_properties=None, extensiontype_=None, **kwargs_
↪)
```

As can be seen here, it includes both the `biophysical_properties` and `morphology` child elements as variables.

Please see the examples in the NeuroML documentation to see usage examples of libNeuroML. Please also note that this module is also included in the top level of the *neuroml* package, so you can use these classes by importing neuroml:

```
from neuroml import AdExIaFCell
```

List of Component classes

AdExIaFCell

class neuroml.nml.nml.**AdExIaFCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *C=None*, *g_l=None*, *EL=None*, *reset=None*, *VT=None*, *thresh=None*, *del_t=None*, *tauw=None*, *refract=None*, *a=None*, *b=None*, ***kwargs_*)

AlphaCondSynapse

class neuroml.nml.nml.**AlphaCondSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *tau_syn=None*, *e_rev=None*, ***kwargs_*)

AlphaCurrSynapse

class neuroml.nml.nml.**AlphaCurrSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *tau_syn=None*, ***kwargs_*)

AlphaCurrentSynapse

class neuroml.nml.nml.**AlphaCurrentSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *tau=None*, *ibase=None*, ***kwargs_*)

AlphaSynapse

class neuroml.nml.nml.**AlphaSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase=None*, *erev=None*, *tau=None*, ***kwargs_*)

Annotation

class neuroml.nml.nml.**Annotation**(*anytypeobjs_=None*, ***kwargs_*)

Placeholder for MIRIAM related metadata, among others.

Base

class neuroml.nml.nml.**Base**(*neuro_lex_id=None*, *id=None*, *extensiontype_=None*, ***kwargs_*)

Anything which can have a unique (within its parent) id of the form NmlId (spaceless combination of letters, numbers and underscore).

BaseCell

class neuroml.nml.nml.**BaseCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *extensiontype_=None*, ***kwargs_*)

BaseCellMembPotCap

class neuroml.nml.nml.**BaseCellMembPotCap**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *C=None*, *extensiontype_=None*, ***kwargs_*)

This is to prevent it conflicting with attribute c (lowercase) e.g. in izhikevichCell2007

BaseConductanceBasedSynapse

class neuroml.nml.nml.**BaseConductanceBasedSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase=None*, *erev=None*, *extensiontype_=None*, ***kwargs_*)

BaseConductanceBasedSynapseTwo

class neuroml.nml.nml.**BaseConductanceBasedSynapseTwo**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase1=None*, *gbase2=None*, *erev=None*, *extensiontype_=None*, ***kwargs_*)

BaseConnection

class neuroml.nml.nml.**BaseConnection**(*neuro_lex_id=None*, *id=None*, *extensiontype_=None*, ***kwargs_*)

Base of all synaptic connections (chemical/electrical/analog, etc.) inside projections

BaseConnectionNewFormat

class neuroml.nml.nml.**BaseConnectionNewFormat**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *extensiontype_=None*, ***kwargs_*)

Base of all synaptic connections with preCell, postSegment, etc. See BaseConnectionOldFormat

BaseConnectionOldFormat

class neuroml.nml.nml.**BaseConnectionOldFormat**(*neuro_lex_id=None*, *id=None*, *pre_cell_id=None*, *pre_segment_id='0'*, *pre_fraction_along='0.5'*, *post_cell_id=None*, *post_segment_id='0'*, *post_fraction_along='0.5'*, *extensiontype_=None*, ***kwargs_*)

Base of all synaptic connections with preCellId, postSegmentId, etc. Note: this is not the best name for these attributes, since Id is superfluous, hence BaseConnectionNewFormat

BaseCurrentBasedSynapse

class neuroml.nml.nml.**BaseCurrentBasedSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *extensiontype_=None*, ***kwargs_*)

BaseNonNegativeIntegerId

class neuroml.nml.nml.**BaseNonNegativeIntegerId**(*neuro_lex_id=None*, *id=None*, *extensiontype_=None*, ***kwargs_*)

Anything which can have a unique (within its parent) id, which must be an integer zero or greater.

BaseProjection

class neuroml.nml.nml.**BaseProjection**(*neuro_lex_id=None*, *id=None*, *presynaptic_population=None*, *postsynaptic_population=None*, *extensiontype_=None*, ***kwargs_*)

Base for projection (set of synaptic connections) between two populations

BasePynnSynapse

class neuroml.nml.nml.**BasePynnSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *tau_syn=None*, *extensiontype_=None*, ***kwargs_*)

BaseSynapse

class neuroml.nml.nml.**BaseSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *extensiontype_=None*, ***kwargs_*)

BaseVoltageDepSynapse

class neuroml.nml.nml.**BaseVoltageDepSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *extensiontype_=None*, ***kwargs_*)

BaseWithoutId

class neuroml.nml.nml.**BaseWithoutId**(*neuro_lex_id=None*, *extensiontype_=None*, ***kwargs_*)

Base element without ID specified *yet*, e.g. for an element with a particular requirement on its id which does not comply with NmlId (e.g. Segment needs nonNegativeInteger).

BiophysicalProperties

class neuroml.nml.nml.**BiophysicalProperties**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *membrane_properties=None*, *intracellular_properties=None*, *extracellular_properties=None*, ***kwargs_*)

Standalone element which is usually inside a single cell, but could be outside and referenced by id.

BiophysicalProperties2CaPools

class neuroml.nml.nml.**BiophysicalProperties2CaPools**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *membrane_properties2_ca_pools=None*, *intracellular_properties2_ca_pools=None*, *extracellular_properties=None*, ***kwargs_*)

Standalone element which is usually inside a single cell, but could be outside and referenced by id.

BlockMechanism

class neuroml.nml.nml.**BlockMechanism**(*type=None*, *species=None*, *block_concentration=None*, *scaling_conc=None*, *scaling_volt=None*, ***kwargs_*)

BlockingPlasticSynapse

class neuroml.nml.nml.**BlockingPlasticSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase=None*, *erev=None*, *tau_decay=None*, *tau_rise=None*, *plasticity_mechanism=None*, *block_mechanism=None*, ***kwargs_*)

Case

class neuroml.nml.nml.**Case**(*condition=None*, *value=None*, ***kwargs_*)

Cell

class neuroml.nml.nml.**Cell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *morphology_attr=None*, *biophysical_properties_attr=None*, *morphology=None*, *biophysical_properties=None*, *extensiontype_=None*, ***kwargs_*)

Should only be used if morphology element is outside the cell. This points to the id of the morphology Should only be used if biophysicalProperties element is outside the cell. This points to the id of the biophysicalProperties

get_actual_proximal(*segment_id*)

Get the proximal point of a segment.

Get the proximal point of a segment, even the proximal field is None and so the proximal point is on the parent (at a point set by fraction_along).

Parameters **segment_id** – ID of segment

Returns proximal point

get_all_segments_in_group(*segment_group*, *assume_all_means_all=True*)

Get all the segments in a segment group of the cell.

Parameters

- **segment_group** – segment group to get all segments of
- **assume_all_means_all** – return all segments if the segment group wasn't explicitly defined

Todo check docstring

Returns list of segments

Raises **Exception** – if no segment group is found in the cell.

get_ordered_segments_in_groups(*group_list*, *check_parentage=False*, *include_cumulative_lengths=False*, *include_path_lengths=False*, *path_length_metric='Path Length from root'*)

Get ordered list of segments in specified groups

Parameters

- **group_list** – list of groups to get segments from
- **check_parentage** – verify parentage
- **include_commulative_lengths** – also include cummulative lengths
- **include_path_lengths** – also include path lengths
- **path_length_metric** –

Returns dictionary of segments with additional information depending on what parameters were used:

Raises Exception if check_parentage is True and parentage cannot be verified

get_segment(*segment_id*)
Get segment object by its id

Parameters **segment_id** – ID of segment

Returns segment

Raises **Exception** – if the segment is not found in the cell

get_segment_group(*sg_id*)
Return the SegmentGroup object for the specified segment group id.

Parameters **sg_id** (*str*) – id of segment group to find

Returns SegmentGroup object of specified ID

Raises **Exception** – if segment group is not found in cell

get_segment_groups_by_substring(*substring*)
Get a dictionary of segment group IDs and the segment groups matching the specified substring

Parameters **substring** (*str*) – substring to match

Returns dictionary with segment group ID as key, and segment group as value

Raises **Exception** – if no segment groups are not found in cell

get_segment_ids_vs_segments()
Get a dictionary of segment IDs and the segments in the cell.

Returns dictionary with segment ID as key, and segment as value

get_segment_length(*segment_id*)
Get the length of the segment.

Parameters **segment_id** – ID of segment

Returns length of segment

get_segment_surface_area(*segment_id*)
Get the surface area of the segment.

Parameters **segment_id** – ID of the segment

Returns surface area of segment

get_segment_volume(*segment_id*)
Get volume of segment

Parameters **segment_id** – ID of the segment

Returns volume of the segment

get_segments_by_substring(*substring*)
Get a dictionary of segment IDs and the segment matching the specified substring

Parameters **substring** (*str*) – substring to match

Returns dictionary with segment ID as key, and segment as value

Raises **Exception** – if no segments are found

summary()
Print cell summary.

Cell2CaPools

class neuroml.nml.nml.**Cell2CaPools**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *morphology_attr=None*, *biophysical_properties_attr=None*, *morphology=None*, *biophysical_properties=None*, *biophysical_properties2_ca_pools=None*, ***kwargs_*)

CellSet

class neuroml.nml.nml.**CellSet**(*neuro_lex_id=None*, *id=None*, *select=None*, *anytypeobjs_=None*, ***kwargs_*)

ChannelDensity

class neuroml.nml.nml.**ChannelDensity**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *cond_density=None*, *erev=None*, *segment_groups='all'*, *segments=None*, *ion=None*, *variable_parameters=None*, *extensiontype_=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityGHK

class neuroml.nml.nml.**ChannelDensityGHK**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *permeability=None*, *segment_groups='all'*, *segments=None*, *ion=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityGHK2

class neuroml.nml.nml.**ChannelDensityGHK2**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *cond_density=None*, *segment_groups='all'*, *segments=None*, *ion=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityNernst

class neuroml.nml.nml.**ChannelDensityNernst**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *cond_density=None*, *segment_groups='all'*, *segments=None*, *ion=None*, *variable_parameters=None*, *extensiontype_=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityNernstCa2

class neuroml.nml.nml.**ChannelDensityNernstCa2**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *cond_density=None*, *segment_groups='all'*, *segments=None*, *ion=None*, *variable_parameters=None*, ***kwargs_*)

ChannelDensityNonUniform

class neuroml.nml.nml.**ChannelDensityNonUniform**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *erev=None*, *ion=None*, *variable_parameters=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityNonUniformGHK

class neuroml.nml.nml.**ChannelDensityNonUniformGHK**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *ion=None*, *variable_parameters=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityNonUniformNernst

class neuroml.nml.nml.**ChannelDensityNonUniformNernst**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *ion=None*, *variable_parameters=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ChannelDensityVShift

class neuroml.nml.nml.**ChannelDensityVShift**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *cond_density=None*, *erev=None*, *segment_groups='all'*, *segments=None*, *ion=None*, *variable_parameters=None*, *v_shift=None*, ***kwargs_*)

ChannelPopulation

class neuroml.nml.nml.**ChannelPopulation**(*neuro_lex_id=None*, *id=None*, *ion_channel=None*, *number=None*, *erev=None*, *segment_groups='all'*, *segments=None*, *ion=None*, *variable_parameters=None*, ***kwargs_*)

Specifying the ion here again is redundant, this will be set in ionChannel definition. It is added here TEMPORARILY since selecting all ca or na conducting channel populations/densities in a cell would be difficult otherwise. Also, it will make it easier to set the correct native simulator value for erev (e.g. ek for ion = k in NEURON). Currently a required attribute. It should be removed in the longer term, due to possible inconsistencies in this value and that in the ionChannel element. TODO: remove.

ClosedState

class neuroml.nml.nml.**ClosedState**(*neuro_lex_id=None*, *id=None*, ***kwargs_*)

ComponentType

class neuroml.nml.nml.**ComponentType**(*name=None*, *extends=None*, *description=None*, *Property=None*, *Parameter=None*, *Constant=None*, *Exposure=None*, *Requirement=None*, *InstanceRequirement=None*, *Dynamics=None*, ***kwargs_*)

Contains an extension to NeuroML by creating custom LEMS ComponentType.

CompoundInput

class neuroml.nml.nml.**CompoundInput**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *pulse_generators=None*, *sine_generators=None*, *ramp_generators=None*, ***kwargs_*)

CompoundInputDL

class neuroml.nml.nml.**CompoundInputDL**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *pulse_generator_dls=None*, *sine_generator_dls=None*, *ramp_generator_dls=None*, ***kwargs_*)

ConcentrationModel_D

class neuroml.nml.nml.**ConcentrationModel_D**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *ion=None*, *resting_conc=None*, *decay_constant=None*, *shell_thickness=None*, *type='decayingPoolConcentrationModel'*, ***kwargs_*)

ConditionalDerivedVariable

class neuroml.nml.nml.**ConditionalDerivedVariable**(*name=None*, *dimension=None*, *description=None*, *exposure=None*, *Case=None*, ***kwargs_*)

LEMS ComponentType for ConditionalDerivedVariable

Connection

class neuroml.nml.nml.**Connection**(*neuro_lex_id=None*, *id=None*, *pre_cell_id=None*, *pre_segment_id='0'*, *pre_fraction_along='0.5'*, *post_cell_id=None*, *post_segment_id='0'*, *post_fraction_along='0.5'*, ***kwargs_*)

Individual chemical (event based) synaptic connection, weight==1 and no delay

get_post_cell_id()

Get the ID of the post-synaptic cell

Returns ID of post-synaptic cell

Return type str

get_post_fraction_along()

Get post-synaptic fraction along information

get_post_info()

Get post-synaptic information summary

get_post_segment_id()

Get the ID of the post-synpatic segment

Returns ID of post-synaptic segment.

Return type str

get_pre_cell_id()
Get the ID of the pre-synaptic cell

Returns ID of pre-synaptic cell

Return type str

get_pre_fraction_along()
Get pre-synaptic fraction along information

get_pre_info()
Get pre-synaptic information summary

get_pre_segment_id()
Get the ID of the pre-synpatic segment

Returns ID of pre-synaptic segment.

Return type str

ConnectionWD

class neuroml.nml.nml.**ConnectionWD**(*neuro_lex_id=None*, *id=None*, *pre_cell_id=None*, *pre_segment_id='0'*, *pre_fraction_along='0.5'*, *post_cell_id=None*, *post_segment_id='0'*, *post_fraction_along='0.5'*, *weight=None*, *delay=None*, ***kwargs_*)
Individual synaptic connection with weight and delay

get_delay_in_ms()
Get connection delay in milli seconds

Returns connection delay in milli seconds

Return type float

get_post_cell_id()
Get the ID of the post-synaptic cell

Returns ID of post-synaptic cell

Return type str

get_post_fraction_along()
Get post-synaptic fraction along information

get_post_info()
Get post-synaptic information summary

get_post_segment_id()
Get the ID of the post-synpatic segment

Returns ID of post-synaptic segment.

Return type str

get_pre_cell_id()
Get the ID of the pre-synaptic cell

Returns ID of pre-synaptic cell

Return type str

get_pre_fraction_along()
Get pre-synaptic fraction along information

get_pre_info()
Get pre-synaptic information summary

get_pre_segment_id()
Get the ID of the pre-synpatic segment

Returns ID of pre-synaptic segment.

Return type str

Constant

class neuroml.nml.nml.**Constant**(*name=None*, *dimension=None*, *value=None*, *description=None*, ***kwargs_*)
LEMS ComponentType for Constant.

ContinuousConnection

class neuroml.nml.nml.**ContinuousConnection**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *pre_component=None*, *post_component=None*, *extensiontype_=None*, ***kwargs_*)
Individual continuous/analog synaptic connection

get_post_cell_id()
Get the ID of the post-synaptic cell

Returns ID of post-synaptic cell

Return type str

get_post_fraction_along()
Get post-synaptic fraction along information

get_post_info()
Get post-synaptic information summary

get_post_segment_id()
Get the ID of the post-synpatic segment

Returns ID of post-synaptic segment.

Return type str

get_pre_cell_id()
Get the ID of the pre-synaptic cell

Returns ID of pre-synaptic cell

Return type str

get_pre_fraction_along()
Get pre-synaptic fraction along information

get_pre_info()
Get pre-synaptic information summary

get_pre_segment_id()
Get the ID of the pre-synpatic segment

Returns ID of pre-synaptic segment.

Return type str

ContinuousConnectionInstance

class neuroml.nml.nml.**ContinuousConnectionInstance**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *pre_component=None*, *post_component=None*, *extensiontype_=None*, ***kwargs_*)

Individual continuous/analog synaptic connection - instance based

ContinuousConnectionInstanceW

class neuroml.nml.nml.**ContinuousConnectionInstanceW**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *pre_component=None*, *post_component=None*, *weight=None*, ***kwargs_*)

Individual continuous/analog synaptic connection - instance based. Includes setting of _weight for the connection

get_weight()

Get weight.

If weight is not set, the default value of 1.0 is returned.

ContinuousProjection

class neuroml.nml.nml.**ContinuousProjection**(*neuro_lex_id=None*, *id=None*, *presynaptic_population=None*, *postsynaptic_population=None*, *continuous_connections=None*, *continuous_connection_instances=None*, *continuous_connection_instance_ws=None*, ***kwargs_*)

Projection between two populations consisting of analog connections (e.g. graded synapses)

exportHdf5(*h5file*, *h5Group*)

Export to HDF5 file.

DecayingPoolConcentrationModel

class neuroml.nml.nml.**DecayingPoolConcentrationModel**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *ion=None*, *resting_conc=None*, *decay_constant=None*, *shell_thickness=None*, *extensiontype_=None*, ***kwargs_*)

Should not be required, as it's present on the species element!

DerivedVariable

class neuroml.nml.nml.**DerivedVariable**(*name=None*, *dimension=None*, *description=None*, *exposure=None*, *value=None*, *select=None*, ***kwargs_*)

LEMS ComponentType for DerivedVariable

DistalDetails

class neuroml.nml.nml.**DistalDetails**(*normalization_end=None*, ***kwargs_*)

DoubleSynapse

class neuroml.nml.nml.**DoubleSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *synapse1=None*, *synapse2=None*, *synapse1_path=None*, *synapse2_path=None*, ***kwargs_*)

Dynamics

class neuroml.nml.nml.**Dynamics**(*StateVariable=None*, *DerivedVariable=None*, *ConditionalDerivedVariable=None*, *TimeDerivative=None*, ***kwargs_*)

LEMS ComponentType for Dynamics

EIF_cond_alpha_isfa_ista

class neuroml.nml.nml.**EIF_cond_alpha_isfa_ista**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, *e_rev_E=None*, *e_rev_I=None*, *a=None*, *b=None*, *delta_T=None*, *tau_w=None*, *v_spike=None*, ***kwargs_*)

EIF_cond_exp_isfa_ista

class neuroml.nml.nml.**EIF_cond_exp_isfa_ista**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, *e_rev_E=None*, *e_rev_I=None*, *a=None*, *b=None*, *delta_T=None*, *tau_w=None*, *v_spike=None*, *extensiontype_=None*, ***kwargs_*)

ElectricalConnection

class neuroml.nml.nml.**ElectricalConnection**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *synapse=None*, *extensiontype_=None*, ***kwargs_*)

Individual electrical synaptic connection

get_post_cell_id()
Get the ID of the post-synaptic cell

Returns ID of post-synaptic cell

Return type str

get_post_fraction_along()
Get post-synaptic fraction along information

get_post_info()
Get post-synaptic information summary

get_post_segment_id()
Get the ID of the post-synpatic segment

Returns ID of post-synaptic segment.

Return type str

get_pre_cell_id()
Get the ID of the pre-synaptic cell

Returns ID of pre-synaptic cell

Return type str

get_pre_fraction_along()
Get pre-synaptic fraction along information

get_pre_info()
Get pre-synaptic information summary

get_pre_segment_id()
Get the ID of the pre-synpatic segment

Returns ID of pre-synaptic segment.

Return type str

ElectricalConnectionInstance

class neuroml.nml.nml.**ElectricalConnectionInstance**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *synapse=None*, *extensiontype_=None*, ***kwargs_*)

Projection between two populations consisting of analog connections (e.g. graded synapses)

ElectricalConnectionInstanceW

class neuroml.nml.nml.**ElectricalConnectionInstanceW**(*neuro_lex_id=None*, *id=None*, *pre_cell=None*, *pre_segment='0'*, *pre_fraction_along='0.5'*, *post_cell=None*, *post_segment='0'*, *post_fraction_along='0.5'*, *synapse=None*, *weight=None*, ***kwargs_*)

Projection between two populations consisting of analog connections (e.g. graded synapses). Includes setting of weight for the connection

get_weight()

Get the weight of the connection

If a weight is not set (or is set to None), returns the default value of 1.0.

Returns weight of connection or 1.0 if not set

Return type float

ElectricalProjection

class neuroml.nml.nml.**ElectricalProjection**(*neuro_lex_id=None*, *id=None*, *presynaptic_population=None*, *postsynaptic_population=None*, *electrical_connections=None*, *electrical_connection_instances=None*, *electrical_connection_instance_ws=None*, ***kwargs_*)

Projection between two populations consisting of electrical connections (gap junctions)

exportHdf5(*h5file*, *h5Group*)

Export to HDF5 file.

ExpCondSynapse

class neuroml.nml.nml.**ExpCondSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *tau_syn=None*, *e_rev=None*, ***kwargs_*)

ExpCurrSynapse

class neuroml.nml.nml.**ExpCurrSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *tau_syn=None*, ***kwargs_*)

ExpOneSynapse

class neuroml.nml.nml.**ExpOneSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase=None*, *erev=None*, *tau_decay=None*, ***kwargs_*)

ExpThreeSynapse

class neuroml.nml.nml.**ExpThreeSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase1=None*, *gbase2=None*, *erev=None*, *tau_decay1=None*, *tau_decay2=None*, *tau_rise=None*, ***kwargs_*)

ExpTwoSynapse

class neuroml.nml.nml.**ExpTwoSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *gbase=None*, *erev=None*, *tau_decay=None*, *tau_rise=None*, *extensiontype_=None*, ***kwargs_*)

ExplicitInput

class neuroml.nml.nml.**ExplicitInput**(*target=None*, *input=None*, *destination=None*, ***kwargs_*)

Single explicit input. Introduced to test inputs in LEMS. Will probably be removed in favour of inputs wrapped in inputList element

get_fraction_along()

Get fraction along.

Returns 0.5 is fraction_along was not set.

get_segment_id()

Get the ID of the segment.

Returns 0 if segment_id was not set.

get_target_cell_id()

Get target cell ID

get_target_population()

Get target population.

Exposure

class neuroml.nml.nml.**Exposure**(*name=None*, *dimension=None*, *description=None*, ***kwargs_*)
LEMS Exposure (ComponentType property)

ExtracellularProperties

class neuroml.nml.nml.**ExtracellularProperties**(*neuro_lex_id=None*, *id=None*, *species=None*, ***kwargs_*)

ExtracellularPropertiesLocal

class neuroml.nml.nml.**ExtracellularPropertiesLocal**(*species=None*, ***kwargs_*)

FitzHughNagumo1969Cell

class neuroml.nml.nml.**FitzHughNagumo1969Cell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *a=None*, *b=None*, *I=None*, *phi=None*, *V0=None*, *W0=None*, ***kwargs_*)

FitzHughNagumoCell

class neuroml.nml.nml.**FitzHughNagumoCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *I=None*, ***kwargs_*)

FixedFactorConcentrationModel

class neuroml.nml.nml.**FixedFactorConcentrationModel**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *ion=None*, *resting_conc=None*, *decay_constant=None*, *rho=None*, ***kwargs_*)
Should not be required, as it's present on the species element!

ForwardTransition

class neuroml.nml.nml.**ForwardTransition**(*neuro_lex_id=None*, *id=None*, *from_=None*, *to=None*, *anytypeobjs_=None*, ***kwargs_*)

GapJunction

class neuroml.nml.nml.**GapJunction**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *conductance=None*, ***kwargs_*)

Gap junction/single electrical connection

GateFractional

class neuroml.nml.nml.**GateFractional**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *sub_gates=None*, ***kwargs_*)

GateFractionalSubgate

class neuroml.nml.nml.**GateFractionalSubgate**(*neuro_lex_id=None*, *id=None*, *fractional_conductance=None*, *notes=None*, *q10_settings=None*, *steady_state=None*, *time_course=None*, ***kwargs_*)

GateHHInstantaneous

class neuroml.nml.nml.**GateHHInstantaneous**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *steady_state=None*, ***kwargs_*)

GateHHRates

class neuroml.nml.nml.**GateHHRates**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *forward_rate=None*, *reverse_rate=None*, ***kwargs_*)

GateHHRatesInf

class neuroml.nml.nml.**GateHHRatesInf**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *forward_rate=None*, *reverse_rate=None*, *steady_state=None*, ***kwargs_*)

GateHHRatesTau

class neuroml.nml.nml.**GateHHRatesTau**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *forward_rate=None*, *reverse_rate=None*, *time_course=None*, ***kwargs_*)

GateHHRatesTauInf

class neuroml.nml.nml.**GateHHRatesTauInf**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *forward_rate=None*, *reverse_rate=None*, *time_course=None*, *steady_state=None*, ***kwargs_*)

GateHHTauInf

class neuroml.nml.nml.**GateHHTauInf**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *time_course=None*, *steady_state=None*, ***kwargs_*)

GateHHUndetermined

class neuroml.nml.nml.**GateHHUndetermined**(*neuro_lex_id=None*, *id=None*, *instances=None*, *type=None*, *notes=None*, *q10_settings=None*, *forward_rate=None*, *reverse_rate=None*, *time_course=None*, *steady_state=None*, *sub_gates=None*, ***kwargs_*)

Note all sub elements for gateHHrates, gateHHratesTau, gateFractional etc. allowed here. Which are valid should be constrained by what type is set

GateKS

class neuroml.nml.nml.**GateKS**(*neuro_lex_id=None*, *id=None*, *instances=None*, *notes=None*, *q10_settings=None*, *closed_states=None*, *open_states=None*, *forward_transition=None*, *reverse_transition=None*, *tau_inf_transition=None*, ***kwargs_*)

GradedSynapse

class neuroml.nml.nml.**GradedSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *conductance=None*, *delta=None*, *Vth=None*, *k=None*, *erev=None*, ***kwargs_*)

Based on synapse in Methods of http://www.nature.com/neuro/journal/v7/n12/abs/nn1352.html.

GridLayout

class neuroml.nml.nml.**GridLayout**(*x_size=None*, *y_size=None*, *z_size=None*, ***kwargs_*)

HHRate

class neuroml.nml.nml.**HHRate**(*type=None*, *rate=None*, *midpoint=None*, *scale=None*, ***kwargs_*)

HHTime

class neuroml.nml.nml.**HHTime**(*type=None*, *rate=None*, *midpoint=None*, *scale=None*, *tau=None*, ***kwargs_*)

HHVariable

class neuroml.nml.nml.**HHVariable**(*type=None*, *rate=None*, *midpoint=None*, *scale=None*, ***kwargs_*)

HH_cond_exp

class neuroml.nml.nml.**HH_cond_exp**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *v_offset=None*, *e_rev_E=None*, *e_rev_I=None*, *e_rev_K=None*, *e_rev_Na=None*, *e_rev_leak=None*, *g_leak=None*, *gbar_K=None*, *gbar_Na=None*, ***kwargs_*)

IF_cond_alpha

class neuroml.nml.nml.**IF_cond_alpha**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, *e_rev_E=None*, *e_rev_I=None*, ***kwargs_*)

IF_cond_exp

class neuroml.nml.nml.**IF_cond_exp**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, *e_rev_E=None*, *e_rev_I=None*, ***kwargs_*)

IF_curr_alpha

class neuroml.nml.nml.**IF_curr_alpha**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, ***kwargs_*)

IF_curr_exp

class neuroml.nml.nml.**IF_curr_exp**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, ***kwargs_*)

IafCell

class neuroml.nml.nml.**IafCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *leak_reversal=None*, *thresh=None*, *reset=None*, *C=None*, *leak_conductance=None*, *extensiontype_=None*, ***kwargs_*)

IafRefCell

class neuroml.nml.nml.**IafRefCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *leak_reversal=None*, *thresh=None*, *reset=None*, *C=None*, *leak_conductance=None*, *refract=None*, ***kwargs_*)

IafTauCell

class neuroml.nml.nml.**IafTauCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *leak_reversal=None*, *thresh=None*, *reset=None*, *tau=None*, *extensiontype_=None*, ***kwargs_*)

IafTauRefCell

class neuroml.nml.nml.**IafTauRefCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *leak_reversal=None*, *thresh=None*, *reset=None*, *tau=None*, *refract=None*, ***kwargs_*)

Include

class neuroml.nml.nml.**Include**(*segment_groups=None*, ***kwargs_*)

IncludeType

class neuroml.nml.nml.**IncludeType**(*href=None*, ***kwargs_*)

InhomogeneousParameter

class neuroml.nml.nml.**InhomogeneousParameter**(*neuro_lex_id=None*, *id=None*, *variable=None*, *metric=None*, *proximal=None*, *distal=None*, ***kwargs_*)

InhomogeneousValue

class neuroml.nml.nml.**InhomogeneousValue**(*inhomogeneous_parameters=None*, *value=None*, ***kwargs_*)

InitMembPotential

class neuroml.nml.nml.**InitMembPotential**(*value=None*, *segment_groups='all'*, ***kwargs_*)

Explicitly set initial membrane potential for the cell

Input

class neuroml.nml.nml.**Input**(*id=None*, *target=None*, *destination=None*, *segment_id=None*, *fraction_along=None*, *extensiontype_=None*, ***kwargs_*)

Individual input to the cell specified by target

get_fraction_along()

Get fraction along.

Returns 0.5 is fraction_along was not set.

get_segment_id()

Get the ID of the segment.

Returns 0 if segment_id was not set.

get_target_cell_id()

Get ID of target cell.

InputList

class neuroml.nml.nml.**InputList**(*neuro_lex_id=None*, *id=None*, *populations=None*, *component=None*, *input=None*, *input_ws=None*, ***kwargs_*)

List of inputs to a population. Currents will be provided by the specified component.

exportHdf5(*h5file*, *h5Group*)

Export to HDF5 file.

InputW

class neuroml.nml.nml.**InputW**(*id=None*, *target=None*, *destination=None*, *segment_id=None*, *fraction_along=None*, *weight=None*, ***kwargs_*)

Individual input to the cell specified by target. Includes setting of _weight for the connection

get_weight()

Get weight.

If weight is not set, the default value of 1.0 is returned.

Instance

class neuroml.nml.nml.**Instance**(*id=None*, *i=None*, *j=None*, *k=None*, *location=None*, ***kwargs_*)

InstanceRequirement

class neuroml.nml.nml.**InstanceRequirement**(*name=None*, *type=None*, ***kwargs_*)

IntracellularProperties

class neuroml.nml.nml.**IntracellularProperties**(*species=None*, *resistivities=None*, *extensiontype_=None*, ***kwargs_*)

IntracellularProperties2CaPools

class neuroml.nml.nml.**IntracellularProperties2CaPools**(*species=None*, *resistivities=None*, ***kwargs_*)

IonChannel

class neuroml.nml.nml.**IonChannel**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *q10_conductance_scalings=None*, *species=None*, *type=None*, *conductance=None*, *gates=None*, *gate_hh_rates=None*, *gate_h_hrates_taus=None*, *gate_hh_tau_infs=None*, *gate_h_hrates_infs=None*, *gate_h_hrates_tau_infs=None*, *gate_hh_instantaneouses=None*, *gate_fractionals=None*, *extensiontype_=None*, ***kwargs_*)

Note ionChannel and ionChannelHH are currently functionally identical. This is needed since many existing examples use ionChannel, some use ionChannelHH. One of these should be removed, probably ionChannelHH.

IonChannelHH

class neuroml.nml.nml.**IonChannelHH**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *q10_conductance_scalings=None*, *species=None*, *type=None*, *conductance=None*, *gates=None*, *gate_hh_rates=None*, *gate_h_hrates_taus=None*, *gate_hh_tau_infs=None*, *gate_h_hrates_infs=None*, *gate_h_hrates_tau_infs=None*, *gate_hh_instantaneouses=None*, *gate_fractionals=None*, ***kwargs_*)

Note ionChannel and ionChannelHH are currently functionally identical. This is needed since many existing examples use ionChannel, some use ionChannelHH. One of these should be removed, probably ionChannelHH.

IonChannelKS

class neuroml.nml.nml.**IonChannelKS**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *species=None*, *conductance=None*, *gate_kses=None*, ***kwargs_*)

Kinetic scheme based ion channel.

IonChannelScalable

class neuroml.nml.nml.**IonChannelScalable**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *q10_conductance_scalings=None*, *extensiontype_=None*, ***kwargs_*)

IonChannelVShift

class neuroml.nml.nml.**IonChannelVShift**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *q10_conductance_scalings=None*, *species=None*, *type=None*, *conductance=None*, *gates=None*, *gate_hh_rates=None*, *gate_h_hrates_taus=None*, *gate_hh_tau_infs=None*, *gate_h_hrates_infs=None*, *gate_h_hrates_tau_infs=None*, *gate_hh_instantaneouses=None*, *gate_fractionals=None*, *v_shift=None*, ***kwargs_*)

Same as ionChannel, but with a vShift parameter to change voltage activation of gates. The exact usage of vShift in expressions for rates is determined by the individual gates.

Izhikevich2007Cell

class neuroml.nml.nml.**Izhikevich2007Cell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *C=None*, *v0=None*, *k=None*, *vr=None*, *vt=None*, *vpeak=None*, *a=None*, *b=None*, *c=None*, *d=None*, ***kwargs_*)

IzhikevichCell

class neuroml.nml.nml.**IzhikevichCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *v0=None*, *thresh=None*, *a=None*, *b=None*, *c=None*, *d=None*, ***kwargs_*)

LEMS_Property

class neuroml.nml.nml.**LEMS_Property**(*name=None*, *dimension=None*, *description=None*, *default_value=None*, ***kwargs_*)

Layout

class neuroml.nml.nml.**Layout**(*spaces=None*, *random=None*, *grid=None*, *unstructured=None*, ***kwargs_*)

LinearGradedSynapse

class neuroml.nml.nml.**LinearGradedSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *conductance=None*, ***kwargs_*)

Behaves just like a one way gap junction.

Location

class neuroml.nml.nml.**Location**(*x=None*, *y=None*, *z=None*, ***kwargs_*)

Member

class neuroml.nml.nml.**Member**(*segments=None*, ***kwargs_*)

MembraneProperties

class neuroml.nml.nml.**MembraneProperties**(*channel_populations=None*, *channel_densities=None*, *channel_density_v_shifts=None*, *channel_density_nernsts=None*, *channel_density_ghks=None*, *channel_density_ghk2s=None*, *channel_density_non_uniforms=None*, *channel_density_non_uniform_nernsts=None*, *channel_density_non_uniform_ghks=None*, *spike_threshes=None*, *specific_capacitances=None*, *init_memb_potentials=None*, *extensiontype_=None*, ***kwargs_*)

MembraneProperties2CaPools

class neuroml.nml.nml.**MembraneProperties2CaPools**(*channel_populations=None*, *channel_densities=None*, *channel_density_v_shifts=None*, *channel_density_nernsts=None*, *channel_density_ghks=None*, *channel_density_ghk2s=None*, *channel_density_non_uniforms=None*, *channel_density_non_uniform_nernsts=None*, *channel_density_non_uniform_ghks=None*, *spike_threshes=None*, *specific_capacitances=None*, *init_memb_potentials=None*, *channel_density_nernst_ca2s=None*, ***kwargs_*)

Morphology

class neuroml.nml.nml.**Morphology**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *segments=None*, *segment_groups=None*, ***kwargs_*)

Standalone element which is usually inside a single cell, but could be outside and referenced by id.

property num_segments

Get the number of segments included in this cell morphology.

Returns number of segments

Return type int

NamedDimensionalType

class neuroml.nml.nml.**NamedDimensionalType**(*name=None*, *dimension=None*, *description=None*, *extensiontype_=None*, ***kwargs_*)

NamedDimensionalVariable

class neuroml.nml.nml.**NamedDimensionalVariable**(*name=None*, *dimension=None*, *description=None*, *exposure=None*, *extensiontype_=None*, ***kwargs_*)

Network

class neuroml.nml.nml.**Network**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *type=None*, *temperature=None*, *spaces=None*, *regions=None*, *extracellular_properties=None*, *populations=None*, *cell_sets=None*, *synaptic_connections=None*, *projections=None*, *electrical_projections=None*, *continuous_projections=None*, *explicit_inputs=None*, *input_lists=None*, ***kwargs_*)

exportHdf5(*h5file*, *h5Group*)

Export to HDF5 file.

get_by_id(*id*)
Get a component by its ID

Parameters **id** (*str*) – ID of component to find

Returns component with specified ID or None if no component with specified ID found

NeuroMLDocument

class neuroml.nml.nml.**NeuroMLDocument**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *includes=None*, *extracellular_properties=None*, *intracellular_properties=None*, *morphology=None*, *ion_channel=None*, *ion_channel_hhs=None*, *ion_channel_v_shifts=None*, *ion_channel_kses=None*, *decaying_pool_concentration_models=None*, *fixed_factor_concentration_models=None*, *alpha_current_synapses=None*, *alpha_synapses=None*, *exp_one_synapses=None*, *exp_two_synapses=None*, *exp_three_synapses=None*, *blocking_plastic_synapses=None*, *double_synapses=None*, *gap_junctions=None*, *silent_synapses=None*, *linear_graded_synapses=None*, *graded_synapses=None*, *biophysical_properties=None*, *cells=None*, *cell2_ca_poolses=None*, *base_cells=None*, *iaf_tau_cells=None*, *iaf_tau_ref_cells=None*, *iaf_cells=None*, *iaf_ref_cells=None*, *izhikevich_cells=None*, *izhikevich2007_cells=None*, *ad_ex_ia_f_cells=None*, *fitz_hugh_nagumo_cells=None*, *fitz_hugh_nagumo1969_cells=None*, *pinsky_rinzel_ca3_cells=None*, *pulse_generators=None*, *pulse_generator_dls=None*, *sine_generators=None*, *sine_generator_dls=None*, *ramp_generators=None*, *ramp_generator_dls=None*, *compound_inputs=None*, *compound_input_dls=None*, *voltage_clamps=None*, *voltage_clamp_triples=None*, *spike_arrays=None*, *timed_synaptic_inputs=None*, *spike_generators=None*, *spike_generator_randoms=None*, *spike_generator_poissons=None*, *spike_generator_ref_poissons=None*, *poisson_firing_synapses=None*, *transient_poisson_firing_synapses=None*, *IF_curr_alpha=None*, *IF_curr_exp=None*, *IF_cond_alpha=None*, *IF_cond_exp=None*, *EIF_cond_exp_isfa_ista=None*, *EIF_cond_alpha_isfa_ista=None*, *HH_cond_exp=None*, *exp_cond_synapses=None*, *alpha_cond_synapses=None*, *exp_curr_synapses=None*, *alpha_curr_synapses=None*, *SpikeSourcePoisson=None*, *networks=None*, *ComponentType=None*, ***kwargs_*)

append(*element*)
Append an element

Parameters **element** (*Object*) – element to append

get_by_id(*id*)
Get a component by specifying its ID.

Parameters **id** (*str*) – id of Component to get

Returns Component with given ID or None if no Component with provided ID was found

summary(*show_includes=True*, *show_non_network=True*)
Get a pretty-printed summary of the complete NeuroMLDocument.

This includes information on the various Components included in the NeuroMLDocument: networks, cells, projections, synapses, and so on.

OpenState

class neuroml.nml.nml.**OpenState**(*neuro_lex_id=None*, *id=None*, ***kwargs_*)

Parameter

class neuroml.nml.nml.**Parameter**(*name=None*, *dimension=None*, *description=None*, ***kwargs_*)

Path

class neuroml.nml.nml.**Path**(*from_=None*, *to=None*, ***kwargs_*)

PinskyRinzelCA3Cell

class neuroml.nml.nml.**PinskyRinzelCA3Cell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *i_soma=None*, *i_dend=None*, *gc=None*, *g_ls=None*, *g_ld=None*, *g_na=None*, *g_kdr=None*, *g_ca=None*, *g_kahp=None*, *g_kc=None*, *g_nmda=None*, *g_ampa=None*, *e_na=None*, *e_ca=None*, *e_k=None*, *e_l=None*, *qd0=None*, *pp=None*, *alphac=None*, *betac=None*, *cm=None*, ***kwargs_*)

PlasticityMechanism

class neuroml.nml.nml.**PlasticityMechanism**(*type=None*, *init_release_prob=None*, *tau_rec=None*, *tau_fac=None*, ***kwargs_*)

Point3DWithDiam

class neuroml.nml.nml.**Point3DWithDiam**(*x=None*, *y=None*, *z=None*, *diameter=None*, ***kwargs_*)
A 3D point with diameter.

distance_to(*other_3d_point*)
Find the distance between this point and another.

Parameters **other_3d_point** (Point3DWithDiam) – other 3D point to calculate distance to

Returns distance between the two points

Return type float

PoissonFiringSynapse

class neuroml.nml.nml.**PoissonFiringSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *average_rate=None*, *synapse=None*, *spike_target=None*, ***kwargs_*)

Population

class neuroml.nml.nml.**Population**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *component=None*, *size=None*, *type=None*, *extracellular_properties=None*, *layout=None*, *instances=None*, ***kwargs_*)

exportHdf5(*h5file*, *h5Group*)
Export to HDF5 file.

get_size()

Projection

class neuroml.nml.nml.**Projection**(*neuro_lex_id=None*, *id=None*, *presynaptic_population=None*, *postsynaptic_population=None*, *synapse=None*, *connections=None*, *connection_wds=None*, ***kwargs_*)
Projection (set of synaptic connections) between two populations. Chemical/event based synaptic transmission

exportHdf5(*h5file*, *h5Group*)
Export to HDF5 file.

Property

class neuroml.nml.nml.**Property**(*tag=None*, *value=None*, ***kwargs_*)
Generic property with a tag and value

ProximalDetails

class neuroml.nml.nml.**ProximalDetails**(*translation_start=None*, ***kwargs_*)

PulseGenerator

class neuroml.nml.nml.**PulseGenerator**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *duration=None*, *amplitude=None*, ***kwargs_*)
Generates a constant current pulse of a certain amplitude (with dimensions for current) for a specified duration after a delay.

PulseGeneratorDL

class neuroml.nml.nml.**PulseGeneratorDL**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *duration=None*, *amplitude=None*, ***kwargs_*)

Generates a constant current pulse of a certain amplitude (non dimensional) for a specified duration after a delay.

Q10ConductanceScaling

class neuroml.nml.nml.**Q10ConductanceScaling**(*q10_factor=None*, *experimental_temp=None*, ***kwargs_*)

Q10Settings

class neuroml.nml.nml.**Q10Settings**(*type=None*, *fixed_q10=None*, *q10_factor=None*, *experimental_temp=None*, ***kwargs_*)

RampGenerator

class neuroml.nml.nml.**RampGenerator**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *duration=None*, *start_amplitude=None*, *finish_amplitude=None*, *baseline_amplitude=None*, ***kwargs_*)

RampGeneratorDL

class neuroml.nml.nml.**RampGeneratorDL**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *duration=None*, *start_amplitude=None*, *finish_amplitude=None*, *baseline_amplitude=None*, ***kwargs_*)

RandomLayout

class neuroml.nml.nml.**RandomLayout**(*number=None*, *regions=None*, ***kwargs_*)

ReactionScheme

class neuroml.nml.nml.**ReactionScheme**(*neuro_lex_id=None*, *id=None*, *source=None*, *type=None*, *anytypeobjs_=None*, ***kwargs_*)

Region

class neuroml.nml.nml.**Region**(*neuro_lex_id=None*, *id=None*, *spaces=None*, *anytypeobjs_=None*, ***kwargs_*)

Requirement

class neuroml.nml.nml.**Requirement**(*name=None*, *dimension=None*, *description=None*, ***kwargs_*)

Resistivity

class neuroml.nml.nml.**Resistivity**(*value=None*, *segment_groups='all'*, ***kwargs_*)

The resistivity, or specific axial resistance, of the cytoplasm

validate_Nml2Quantity_resistivity(*value*)

validate_Nml2Quantity_resistivity_patterns_ = [['^-?([0-9]*(\\.[0-9]+)?)([eE]-?[0-9]+)?[\\s]*(ohm_cm|kohm_cm|ohm_m)$']]

ReverseTransition

class neuroml.nml.nml.**ReverseTransition**(*neuro_lex_id=None*, *id=None*, *from_=None*, *to=None*, *anytypeobjs_=None*, ***kwargs_*)

Segment

class neuroml.nml.nml.**Segment**(*neuro_lex_id=None*, *id=None*, *name=None*, *parent=None*, *proximal=None*, *distal=None*, ***kwargs_*)

property length

Get the length of the segment.

Returns length of the segment

Return type float

property surface_area

Get the surface area of the segment.

Returns surface area of segment

Return type float

property volume

Get the volume of the segment.

Returns volume of segment

Return type float

SegmentEndPoint

class neuroml.nml.nml.**SegmentEndPoint**(*segments=None*, ***kwargs_*)

SegmentGroup

class neuroml.nml.nml.**SegmentGroup**(*neuro_lex_id=None*, *id=None*, *notes=None*, *properties=None*, *annotation=None*, *members=None*, *includes=None*, *paths=None*, *sub_trees=None*, *inhomogeneous_parameters=None*, ***kwargs_*)

SegmentParent

class neuroml.nml.nml.**SegmentParent**(*segments=None*, *fraction_along='1'*, ***kwargs_*)

SilentSynapse

class neuroml.nml.nml.**SilentSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, ***kwargs_*)

Dummy synapse which emits no current. Used as presynaptic endpoint for analog synaptic connection (continuousConnection).

SineGenerator

class neuroml.nml.nml.**SineGenerator**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *phase=None*, *duration=None*, *amplitude=None*, *period=None*, ***kwargs_*)

SineGeneratorDL

class neuroml.nml.nml.**SineGeneratorDL**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *phase=None*, *duration=None*, *amplitude=None*, *period=None*, ***kwargs_*)

Space

class neuroml.nml.nml.**Space**(*neuro_lex_id=None*, *id=None*, *based_on=None*, *structure=None*, ***kwargs_*)

SpaceStructure

class neuroml.nml.nml.**SpaceStructure**(*x_spacing=None*, *y_spacing=None*, *z_spacing=None*, *x_start=0*, *y_start=0*, *z_start=0*, ***kwargs_*)

Species

class neuroml.nml.nml.**Species**(*id=None*, *concentration_model=None*, *ion=None*, *initial_concentration=None*, *initial_ext_concentration=None*, *segment_groups='all'*, ***kwargs_*)

Specifying the ion here again is redundant, the ion name should be the same as id. Kept for now until LEMS implementation can select by id. TODO: remove.

SpecificCapacitance

class neuroml.nml.nml.**SpecificCapacitance**(*value=None*, *segment_groups='all'*, ***kwargs_*)

Capacitance per unit area

Spike

class neuroml.nml.nml.**Spike**(*neuro_lex_id=None*, *id=None*, *time=None*, ***kwargs_*)

SpikeArray

class neuroml.nml.nml.**SpikeArray**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *spikes=None*, ***kwargs_*)

SpikeGenerator

class neuroml.nml.nml.**SpikeGenerator**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *period=None*, ***kwargs_*)

SpikeGeneratorPoisson

class neuroml.nml.nml.**SpikeGeneratorPoisson**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *average_rate=None*, *extensiontype_=None*, ***kwargs_*)

SpikeGeneratorRandom

class neuroml.nml.nml.**SpikeGeneratorRandom**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *max_isi=None*, *min_isi=None*, ***kwargs_*)

SpikeGeneratorRefPoisson

class neuroml.nml.nml.**SpikeGeneratorRefPoisson**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *average_rate=None*, *minimum_isi=None*, ***kwargs_*)

SpikeSourcePoisson

class neuroml.nml.nml.**SpikeSourcePoisson**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *start=None*, *duration=None*, *rate=None*, ***kwargs_*)

SpikeThresh

class neuroml.nml.nml.**SpikeThresh**(*value=None*, *segment_groups='all'*, ***kwargs_*)

Membrane potential at which to emit a spiking event. Note, usually the spiking event will not be emitted again until the membrane potential has fallen below this value and rises again to cross it in a positive direction.

Standalone

class neuroml.nml.nml.**Standalone**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *extensiontype_=None*, ***kwargs_*)

Elements which can stand alone and be referenced by id, e.g. cell, morphology.

StateVariable

class neuroml.nml.nml.**StateVariable**(*name=None*, *dimension=None*, *description=None*, *exposure=None*, ***kwargs_*)

SubTree

class neuroml.nml.nml.**SubTree**(*from_=None*, *to=None*, ***kwargs_*)

SynapticConnection

class neuroml.nml.nml.**SynapticConnection**(*from_=None*, *to=None*, *synapse=None*, *destination=None*, ***kwargs_*)

Single explicit connection. Introduced to test connections in LEMS. Will probably be removed in favour of connections wrapped in projection element

TauInfTransition

class neuroml.nml.nml.**TauInfTransition**(*neuro_lex_id=None*, *id=None*, *from_=None*, *to=None*, *steady_state=None*, *time_course=None*, ***kwargs_*)

TimeDerivative

class neuroml.nml.nml.**TimeDerivative**(*variable=None*, *value=None*, ***kwargs_*)

TimedSynapticInput

class neuroml.nml.nml.**TimedSynapticInput**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *synapse=None*, *spike_target=None*, *spikes=None*, ***kwargs_*)

TransientPoissonFiringSynapse

class neuroml.nml.nml.**TransientPoissonFiringSynapse**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *average_rate=None*, *delay=None*, *duration=None*, *synapse=None*, *spike_target=None*, ***kwargs_*)

UnstructuredLayout

class neuroml.nml.nml.**UnstructuredLayout**(*number=None*, ***kwargs_*)

VariableParameter

class neuroml.nml.nml.**VariableParameter**(*parameter=None*, *segment_groups=None*, *inhomogeneous_value=None*, ***kwargs_*)

VoltageClamp

class neuroml.nml.nml.**VoltageClamp**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *delay=None*, *duration=None*, *target_voltage=None*, *simple_series_resistance=None*, ***kwargs_*)

VoltageClampTriple

class neuroml.nml.nml.**VoltageClampTriple**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *active=None*, *delay=None*, *duration=None*, *conditioning_voltage=None*, *testing_voltage=None*, *return_voltage=None*, *simple_series_resistance=None*, ***kwargs_*)

basePyNNCell

class neuroml.nml.nml.**basePyNNCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *extensiontype_=None*, ***kwargs_*)

basePyNNIaFCell

class neuroml.nml.nml.**basePyNNIaFCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, *extensiontype_=None*, ***kwargs_*)

basePyNNIaFCondCell

class neuroml.nml.nml.**basePyNNIaFCondCell**(*neuro_lex_id=None*, *id=None*, *metaid=None*, *notes=None*, *properties=None*, *annotation=None*, *cm=None*, *i_offset=None*, *tau_syn_E=None*, *tau_syn_I=None*, *v_init=None*, *tau_m=None*, *tau_refrac=None*, *v_reset=None*, *v_rest=None*, *v_thresh=None*, *e_rev_E=None*, *e_rev_I=None*, *extensiontype_=None*, ***kwargs_*)

1.3.2 loaders Module

class neuroml.loaders.**ArrayMorphLoader**
Bases: object

classmethod load(*filepath*)
Right now this load method isn't done in a very nice way. TODO: Complete refactoring.

class neuroml.loaders.**JSONLoader**
Bases: object

classmethod load(*file*)

classmethod load_from_mongodb(*db*, *id*, *host=None*, *port=None*)

class neuroml.loaders.**NeuroMLHdf5Loader**
Bases: object

classmethod load(*src*, *optimized=False*)

class neuroml.loaders.**NeuroMLLoader**
Bases: object

classmethod load(*src*)

class neuroml.loaders.**SWCLoader**
Bases: object

WARNING: Class defunct

classmethod load_swc_single(*src*, *name=None*)

neuroml.loaders.**print_**(*text*, *verbose=True*)

neuroml.loaders.**read_neuroml2_file**(*nml2_file_name*, *include_includes=False*, *verbose=False*, *already_included=[]*, *print_method=<function print_>*, *optimized=False*)

neuroml.loaders.**read_neuroml2_string**(*nml2_string*, *include_includes=False*, *verbose=False*, *already_included=[]*, *print_method=<function print_>*, *optimized=False*, *base_path=None*)

1.3.3 writers Module

class neuroml.writers.**ArrayMorphWriter**
Bases: object

For now just testing a simple method which can write a morphology, not a NeuroMLDocument.

classmethod write(*data*, *filepath*)

class neuroml.writers.**JSONWriter**
Bases: object

Write a NeuroMLDocument to JSON, particularly useful when dealing with lots of ArrayMorphs.

classmethod write(*neuroml_document*, *file*)

classmethod write_to_mongodb(*neuroml_document*, *db*, *host=None*, *port=None*, *id=None*)

class neuroml.writers.**NeuroMLHdf5Writer**
Bases: object

classmethod write(*nml_doc*, *h5_file_name*, *embed_xml=True*, *compress=True*)

class neuroml.writers.**NeuroMLWriter**
Bases: object

classmethod write(*nmldoc*, *file*, *close=True*)
Writes from NeuroMLDocument to nml file in future can implement from other types via chain of responsibility pattern.

1.3.4 utils Module

Utilities for checking generated code

neuroml.utils.**add_all_to_document**(*nml_doc_src*, *nml_doc_tgt*, *verbose=False*)
Add all members of the source NeuroML document to the target NeuroML document.

Parameters

- **nml_doc_src** (NeuroMLDocument) – source NeuroML document to copy from
- **nml_doc_tgt** (NeuroMLDocument) – target NeuroML document to copy to
- **verbose** (*bool*) – control verbosity of working

Raises **Exception** – if a member could not be copied.

neuroml.utils.**append_to_element**(*parent*, *child*)
Append a child element to a parent Component

Parameters

- **parent** (*Object*) – parent NeuroML component to add element to
- **child** (*Object*) – child NeuroML component to be added to parent

Raises **Exception** – when the child could not be added to the parent

neuroml.utils.**get_summary**(*nml_file_name*)
Get a summary of the given NeuroML file.

Parameters **nml_file_name** (*str*) – name of NeuroML file to get summary of

Returns summary of provided file

Return type str

neuroml.utils.**has_segment_fraction_info**(*connections*)
Check if connections include fraction information

Parameters **connections** (*list*) – list of connection objects

Returns True if connections include fragment information, otherwise False

Return type Boolean

neuroml.utils.**is_valid_neuroml2**(*file_name*)
Check if a file is valid NeuroML2.

Parameters **file_name** (*str*) – name of NeuroML file to check

Returns True if file is valid, False if not.

Return type Boolean

neuroml.utils.**main**()

neuroml.utils.**print_summary**(*nml_file_name*)
Print a summary of the NeuroML model in the given file.

Parameters **nml_file_name** (*str*) – name of NeuroML file to print summary of

neuroml.utils.**validate_neuroml2**(*file_name*)
Validate a NeuroML document against the NeuroML schema specification.

Parameters **file_name** (*str*) – name of NeuroML file to validate.

1.3.5 arraymorph Module

1.4 Examples

The examples in this section are intended to give in depth overviews of how to accomplish specific tasks with libNeuroML.

These examples are located in the neuroml/examples directory and can be tested to confirm they work by running the run_all.py script.

Examples

1.4.1 Creating a NeuroML morphology

```
"""
Example of connecting segments together to create a
multicompartmental model of a cell.
"""

import neuroml
import neuroml.writers as writers

p = neuroml.Point3DWithDiam(x=0, y=0, z=0, diameter=50)
d = neuroml.Point3DWithDiam(x=50, y=0, z=0, diameter=50)
soma = neuroml.Segment(proximal=p, distal=d)
soma.name = "Soma"
soma.id = 0

# Make an axon with 100 compartments:

parent = neuroml.SegmentParent(segments=soma.id)
parent_segment = soma
axon_segments = []
```

(continues on next page)

(continued from previous page)

```
seg_id = 1

for i in range(100):
    p = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    d = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x + 10,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    axon_segment = neuroml.Segment(proximal=p, distal=d, parent=parent)

    axon_segment.id = seg_id

    axon_segment.name = "axon_segment_" + str(axon_segment.id)

    # now reset everything:
    parent = neuroml.SegmentParent(segments=axon_segment.id)
    parent_segment = axon_segment
    seg_id += 1

    axon_segments.append(axon_segment)

test_morphology = neuroml.Morphology()
test_morphology.segments.append(soma)
test_morphology.segments += axon_segments
test_morphology.id = "TestMorphology"

cell = neuroml.Cell()
cell.name = "TestCell"
cell.id = "TestCell"
cell.morphology = test_morphology


doc = neuroml.NeuroMLDocument(id="TestNeuroMLDocument")

doc.cells.append(cell)

nml_file = "tmp/testmorphwrite.nml"

writers.NeuroMLWriter.write(doc, nml_file)

print("Written morphology file to: " + nml_file)

###### Validate the NeuroML ######
```

(continues on next page)

(continued from previous page)

```
from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.2 Loading and modifying a file

```
"""
In this example an axon is built, a morphology is loaded, the axon is
then connected to the loadeed morphology.
"""

import neuroml
import neuroml.loaders as loaders
import neuroml.writers as writers

fn = "./test_files/Purk2M9s.nml"
doc = loaders.NeuroMLLoader.load(fn)
print("Loaded morphology file from: " + fn)

# get the parent segment:
parent_segment = doc.cells[0].morphology.segments[0]

parent = neuroml.SegmentParent(segments=parent_segment.id)

# make an axon:
seg_id = 5000  # need a way to get a unique id from a morphology
axon_segments = []
for i in range(10):
    p = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    d = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x + 10,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    axon_segment = neuroml.Segment(proximal=p, distal=d, parent=parent)

    axon_segment.id = seg_id

    axon_segment.name = "axon_segment_" + str(axon_segment.id)

    # now reset everything:
```

(continues on next page)

(continued from previous page)

```
    parent = neuroml.SegmentParent(segments=axon_segment.id)
    parent_segment = axon_segment
    seg_id += 1

    axon_segments.append(axon_segment)

doc.cells[0].morphology.segments += axon_segments

nml_file = "./tmp/modified_morphology.nml"

writers.NeuroMLWriter.write(doc, nml_file)

print("Saved modified morphology file to: " + nml_file)


###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.3 Building a network

```
"""

Example to build a full spiking IaF network
through libNeuroML, save it as XML and validate it

"""

from neuroml import NeuroMLDocument
from neuroml import IafCell
from neuroml import Network
from neuroml import ExpOneSynapse
from neuroml import Population
from neuroml import PulseGenerator
from neuroml import ExplicitInput
from neuroml import SynapticConnection
import neuroml.writers as writers
from random import random


nml_doc = NeuroMLDocument(id="IafNet")

IafCell0 = IafCell(
    id="iaf0",
    C="1.0 nF",
    thresh="-50mV",
    reset="-65mV",
    leak_conductance="10 nS",
```

(continues on next page)

(continued from previous page)

```
    leak_reversal="-65mV",
)

nml_doc.iaf_cells.append(IafCell0)

IafCell1 = IafCell(
    id="iaf1",
    C="1.0 nF",
    thresh="-50mV",
    reset="-65mV",
    leak_conductance="20 nS",
    leak_reversal="-65mV",
)

nml_doc.iaf_cells.append(IafCell1)

syn0 = ExpOneSynapse(id="syn0", gbase="65nS", erev="0mV", tau_decay="3ms")

nml_doc.exp_one_synapses.append(syn0)

net = Network(id="IafNet")

nml_doc.networks.append(net)

size0 = 5
pop0 = Population(id="IafPop0", component=IafCell0.id, size=size0)

net.populations.append(pop0)

size1 = 5
pop1 = Population(id="IafPop1", component=IafCell0.id, size=size1)

net.populations.append(pop1)

prob_connection = 0.5

for pre in range(0, size0):

    pg = PulseGenerator(
        id="pulseGen_%i" % pre,
        delay="0ms",
        duration="100ms",
        amplitude="%f nA" % (0.1 * random()),
    )

    nml_doc.pulse_generators.append(pg)

    exp_input = ExplicitInput(target="%s[%i]" % (pop0.id, pre), input=pg.id)

    net.explicit_inputs.append(exp_input)

    for post in range(0, size1):
```

(continues on next page)

(continued from previous page)

```
        # fromxx is used since from is Python keyword
        if random() <= prob_connection:
            syn = SynapticConnection(
                from_="%s[%i]" % (pop0.id, pre),
                synapse=syn0.id,
                to="%s[%i]" % (pop1.id, post),
            )
            net.synaptic_connections.append(syn)

nml_file = "tmp/testnet.nml"
writers.NeuroMLWriter.write(nml_doc, nml_file)


print("Written network file to: " + nml_file)


###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.4 Building a 3D network

```
"""

Example to build a full spiking IaF network throught libNeuroML & save it as XML &␣
↪validate it

"""

from neuroml import NeuroMLDocument
from neuroml import Network
from neuroml import ExpOneSynapse
from neuroml import Population
from neuroml import Property
from neuroml import Cell
from neuroml import Location
from neuroml import Instance
from neuroml import Morphology
from neuroml import Point3DWithDiam
from neuroml import Segment
from neuroml import SegmentParent
from neuroml import Projection
from neuroml import Connection

import neuroml.writers as writers
from random import random

soma_diam = 10
```

(continues on next page)

(continued from previous page)

```
soma_len = 10
dend_diam = 2
dend_len = 10
dend_num = 10


def generateRandomMorphology():

    morphology = Morphology()

    p = Point3DWithDiam(x=0, y=0, z=0, diameter=soma_diam)
    d = Point3DWithDiam(x=soma_len, y=0, z=0, diameter=soma_diam)
    soma = Segment(proximal=p, distal=d, name="Soma", id=0)

    morphology.segments.append(soma)
    parent_seg = soma

    for dend_id in range(0, dend_num):

        p = Point3DWithDiam(x=d.x, y=d.y, z=d.z, diameter=dend_diam)
        d = Point3DWithDiam(x=p.x, y=p.y + dend_len, z=p.z, diameter=dend_diam)
        dend = Segment(proximal=p, distal=d, name="Dend_%i" % dend_id, id=1 + dend_id)
        dend.parent = SegmentParent(segments=parent_seg.id)
        parent_seg = dend

        morphology.segments.append(dend)

    morphology.id = "TestMorphology"

    return morphology


def run():

    cell_num = 10
    x_size = 500
    y_size = 500
    z_size = 500

    nml_doc = NeuroMLDocument(id="Net3DExample")

    syn0 = ExpOneSynapse(id="syn0", gbase="65nS", erev="0mV", tau_decay="3ms")
    nml_doc.exp_one_synapses.append(syn0)

    net = Network(id="Net3D")
    nml_doc.networks.append(net)

    proj_count = 0
    # conn_count = 0

    for cell_id in range(0, cell_num):
```

(continues on next page)

(continued from previous page)

```
        cell = Cell(id="Cell_%i" % cell_id)

        cell.morphology = generateRandomMorphology()

        nml_doc.cells.append(cell)

        pop = Population(
            id="Pop_%i" % cell_id, component=cell.id, type="populationList"
        )
        net.populations.append(pop)
        pop.properties.append(Property(tag="color", value="1 0 0"))

        inst = Instance(id="0")
        pop.instances.append(inst)

        inst.location = Location(
            x=str(x_size * random()), y=str(y_size * random()), z=str(z_size * random())
        )

        prob_connection = 0.5
        for post in range(0, cell_num):
            if post is not cell_id and random() <= prob_connection:

                from_pop = "Pop_%i" % cell_id
                to_pop = "Pop_%i" % post

                pre_seg_id = 0
                post_seg_id = 1

                projection = Projection(
                    id="Proj_%i" % proj_count,
                    presynaptic_population=from_pop,
                    postsynaptic_population=to_pop,
                    synapse=syn0.id,
                )
                net.projections.append(projection)
                connection = Connection(
                    id=proj_count,
                    pre_cell_id="%s[%i]" % (from_pop, 0),
                    pre_segment_id=pre_seg_id,
                    pre_fraction_along=random(),
                    post_cell_id="%s[%i]" % (to_pop, 0),
                    post_segment_id=post_seg_id,
                    post_fraction_along=random(),
                )

                projection.connections.append(connection)
                proj_count += 1
                # net.synaptic_connections.append(SynapticConnection(from_="%s[%i]"
↪%(from_pop,0),  to="%s[%i]"%(to_pop,0)))

    #######   Write to file  ######
```

(continues on next page)

(continued from previous page)

```
    nml_file = "tmp/net3d.nml"
    writers.NeuroMLWriter.write(nml_doc, nml_file)

    print("Written network file to: " + nml_file)

    ###### Validate the NeuroML ######

    from neuroml.utils import validate_neuroml2

    validate_neuroml2(nml_file)


run()
```

1.4.5 Ion channels

```
"""
Generating a Hodgkin-Huxley Ion Channel and writing it to NeuroML
"""

import neuroml
import neuroml.writers as writers

chan = neuroml.IonChannelHH(
    id="na",
    conductance="10pS",
    species="na",
    notes="This is an example voltage-gated Na channel",
)

m_gate = neuroml.GateHHRates(id="m", instances="3")
h_gate = neuroml.GateHHRates(id="h", instances="1")

m_gate.forward_rate = neuroml.HHRate(
    type="HHExpRate", rate="0.07per_ms", midpoint="-65mV", scale="-20mV"
)

m_gate.reverse_rate = neuroml.HHRate(
    type="HHSigmoidRate", rate="1per_ms", midpoint="-35mV", scale="10mV"
)

h_gate.forward_rate = neuroml.HHRate(
    type="HHExpLinearRate", rate="0.1per_ms", midpoint="-55mV", scale="10mV"
)

h_gate.reverse_rate = neuroml.HHRate(
    type="HHExpRate", rate="0.125per_ms", midpoint="-65mV", scale="-80mV"
)
```

(continues on next page)

(continued from previous page)

```
chan.gate_hh_rates.append(m_gate)
chan.gate_hh_rates.append(h_gate)

doc = neuroml.NeuroMLDocument()
doc.ion_channel_hhs.append(chan)

doc.id = "ChannelMLDemo"

nml_file = "./tmp/ionChannelTest.xml"
writers.NeuroMLWriter.write(doc, nml_file)

print("Written channel file to: " + nml_file)


###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.6 PyNN models

```
"""

Example to build a PyNN based network

"""

from neuroml import NeuroMLDocument
from neuroml import *
import neuroml.writers as writers
from random import random


#########################   Build the network   ###################################

nml_doc = NeuroMLDocument(id="IafNet")


pynn0 = IF_curr_alpha(
    id="IF_curr_alpha_pop_IF_curr_alpha",
    cm="1.0",
    i_offset="0.9",
    tau_m="20.0",
    tau_refrac="10.0",
    tau_syn_E="0.5",
    tau_syn_I="0.5",
    v_init="-65",
    v_reset="-62.0",
    v_rest="-65.0",
```

(continues on next page)

(continued from previous page)

```
    v_thresh="-52.0",
)
nml_doc.IF_curr_alpha.append(pynn0)

pynn1 = HH_cond_exp(
    id="HH_cond_exp_pop_HH_cond_exp",
    cm="0.2",
    e_rev_E="0.0",
    e_rev_I="-80.0",
    e_rev_K="-90.0",
    e_rev_Na="50.0",
    e_rev_leak="-65.0",
    g_leak="0.01",
    gbar_K="6.0",
    gbar_Na="20.0",
    i_offset="0.2",
    tau_syn_E="0.2",
    tau_syn_I="2.0",
    v_init="-65",
    v_offset="-63.0",
)
nml_doc.HH_cond_exp.append(pynn1)

pynnSynn0 = ExpCondSynapse(id="ps1", tau_syn="5", e_rev="0")
nml_doc.exp_cond_synapses.append(pynnSynn0)

nml_file = "tmp/pynn_network.xml"
writers.NeuroMLWriter.write(nml_doc, nml_file)
print("Saved to: " + nml_file)


###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.7 Synapses

```
"""

Example to create a file with multiple synapse types

"""

from neuroml import NeuroMLDocument
from neuroml import *
import neuroml.writers as writers
from random import random
```

(continues on next page)

(continued from previous page)

```
nml_doc = NeuroMLDocument(id="SomeSynapses")

expOneSyn0 = ExpOneSynapse(id="ampa", tau_decay="5ms", gbase="1nS", erev="0mV")
nml_doc.exp_one_synapses.append(expOneSyn0)

expTwoSyn0 = ExpTwoSynapse(
    id="gaba", tau_decay="12ms", tau_rise="3ms", gbase="1nS", erev="-70mV"
)
nml_doc.exp_two_synapses.append(expTwoSyn0)

bpSyn = BlockingPlasticSynapse(
    id="blockStpSynDep", gbase="1nS", erev="0mV", tau_rise="0.1ms", tau_decay="2ms"
)
bpSyn.notes = "This is a note"
bpSyn.plasticity_mechanism = PlasticityMechanism(
    type="tsodyksMarkramDepMechanism", init_release_prob="0.5", tau_rec="120 ms"
)
bpSyn.block_mechanism = BlockMechanism(
    type="voltageConcDepBlockMechanism",
    species="mg",
    block_concentration="1.2 mM",
    scaling_conc="1.920544 mM",
    scaling_volt="16.129 mV",
)

nml_doc.blocking_plastic_synapses.append(bpSyn)


nml_file = "tmp/synapses.xml"
writers.NeuroMLWriter.write(nml_doc, nml_file)
print("Saved to: " + nml_file)


###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.8 Working with JSON serialization

One thing to note is that the JSONWriter, unlike NeuroMLWriter, will serializing using array-based (Arraymorph) representation if this has been used.

```
"""
In this example an axon is built, a morphology is loaded, the axon is
then connected to the loadeed morphology. The whole thing is serialized
in JSON format, reloaded and validated.
"""
```

(continues on next page)

(continued from previous page)

```
import neuroml
import neuroml.loaders as loaders
import neuroml.writers as writers

fn = "./test_files/Purk2M9s.nml"
doc = loaders.NeuroMLLoader.load(fn)
print("Loaded morphology file from: " + fn)

# get the parent segment:
parent_segment = doc.cells[0].morphology.segments[0]

parent = neuroml.SegmentParent(segments=parent_segment.id)

# make an axon:
seg_id = 5000  # need a way to get a unique id from a morphology
axon_segments = []
for i in range(10):
    p = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    d = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x + 10,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    axon_segment = neuroml.Segment(proximal=p, distal=d, parent=parent)

    axon_segment.id = seg_id

    axon_segment.name = "axon_segment_" + str(axon_segment.id)

    # now reset everything:
    parent = neuroml.SegmentParent(segments=axon_segment.id)
    parent_segment = axon_segment
    seg_id += 1

    axon_segments.append(axon_segment)

doc.cells[0].morphology.segments += axon_segments

json_file = "./tmp/modified_morphology.json"

writers.JSONWriter.write(doc, json_file)

print("Saved modified morphology in JSON format to: " + json_file)
```

(continues on next page)

(continued from previous page)

```
##### load it again, this time write it to a normal neuroml file ###

neuroml_document_from_json = loaders.JSONLoader.load(json_file)

print("Re-loaded neuroml document in JSON format to NeuroMLDocument object")

nml_file = "./tmp/modified_morphology_from_json.nml"

writers.NeuroMLWriter.write(neuroml_document_from_json, nml_file)

###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.9 Working with arraymorphs

```
"""
Example of connecting segments together to create a
multicompartmental model of a cell.

In this case ArrayMorphology will be used rather than
Morphology - demonstrating its similarity and
ability to save in HDF5 format
"""

import neuroml
import neuroml.writers as writers
import neuroml.arraymorph as am

p = neuroml.Point3DWithDiam(x=0, y=0, z=0, diameter=50)
d = neuroml.Point3DWithDiam(x=50, y=0, z=0, diameter=50)
soma = neuroml.Segment(proximal=p, distal=d)
soma.name = "Soma"
soma.id = 0

# now make an axon with 100 compartments:

parent = neuroml.SegmentParent(segments=soma.id)
parent_segment = soma
axon_segments = []
seg_id = 1
for i in range(100):
    p = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
```

(continues on next page)

(continued from previous page)

```
    )

    d = neuroml.Point3DWithDiam(
        x=parent_segment.distal.x + 10,
        y=parent_segment.distal.y,
        z=parent_segment.distal.z,
        diameter=0.1,
    )

    axon_segment = neuroml.Segment(proximal=p, distal=d, parent=parent)

    axon_segment.id = seg_id

    axon_segment.name = "axon_segment_" + str(axon_segment.id)

    # now reset everything:
    parent = neuroml.SegmentParent(segments=axon_segment.id)
    parent_segment = axon_segment
    seg_id += 1

    axon_segments.append(axon_segment)

test_morphology = am.ArrayMorphology()
test_morphology.segments.append(soma)
test_morphology.segments += axon_segments
test_morphology.id = "TestMorphology"

cell = neuroml.Cell()
cell.name = "TestCell"
cell.id = "TestCell"
cell.morphology = test_morphology


doc = neuroml.NeuroMLDocument()
# doc.name = "Test neuroML document"

doc.cells.append(cell)
doc.id = "TestNeuroMLDocument"

nml_file = "tmp/arraymorph.nml"

writers.NeuroMLWriter.write(doc, nml_file)

print("Written morphology file to: " + nml_file)

###### Validate the NeuroML ######

from neuroml.utils import validate_neuroml2

validate_neuroml2(nml_file)
```

1.4.10 Working with Izhikevich Cells

These examples were kindly contributed by Steve Marsh

```
# from neuroml import NeuroMLDocument
from neuroml import IzhikevichCell
from neuroml.loaders import NeuroMLLoader
from neuroml.utils import validate_neuroml2


def load_izhikevich(filename="./test_files/SingleIzhikevich.nml"):
    nml_filename = filename
    validate_neuroml2(nml_filename)
    nml_doc = NeuroMLLoader.load(nml_filename)

    iz_cells = nml_doc.izhikevich_cells
    for i, iz in enumerate(iz_cells):
        if isinstance(iz, IzhikevichCell):
            neuron_string = "%d %s %s %s %s %s (%s)" % (
                i,
                iz.v0,
                iz.a,
                iz.b,
                iz.c,
                iz.d,
                iz.id,
            )
            print(neuron_string)
        else:
            print("Error: Cell %d is not an IzhikevichCell" % i)


load_izhikevich()
```

```
from neuroml import NeuroMLDocument
from neuroml import IzhikevichCell
from neuroml.writers import NeuroMLWriter
from neuroml.utils import validate_neuroml2


def write_izhikevich(filename="./tmp/SingleIzhikevich_test.nml"):
    nml_doc = NeuroMLDocument(id="SingleIzhikevich")
    nml_filename = filename

    iz0 = IzhikevichCell(
        id="iz0", v0="-70mV", thresh="30mV", a="0.02", b="0.2", c="-65.0", d="6"
    )

    nml_doc.izhikevich_cells.append(iz0)

    NeuroMLWriter.write(nml_doc, nml_filename)
    validate_neuroml2(nml_filename)
```

(continues on next page)

(continued from previous page)

```
write_izhikevich()
```

1.5 References

CHAPTER

TWO

CONTRIBUTING

2.1 How to contribute

libNeuroML development happens on GitHub, so you will need a GitHub account to contribute to the repository. Contributions are made using the standard Pull Request workflow.

2.1.1 Setting up

Please take a look at the GitHub documentation here: http://help.github.com/fork-a-repo/

To begin, please fork the repo on the GitHub website. You should now have a libNeuroML under you username. Next, we clone our fork to get a local copy on our computer:

```
git clone git@github.com:_username_/libNeuroML.git
```

While not necessary, it is good practice to add the upstream repository as a remote that you will follow:

```
cd libNeuroML
git remote add upstream https://github.com/NeuralEnsemble/libNeuroML.git
git fetch upstream
```

You can check which branch are you following doing:

```
git branch -a
```

You should have something like:

```
git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/upstream/master
```

2.1.2 Sync with upstream

Before starting to do some work, please check to see that you have the latest copy of the sources in your local repository:

```
git fetch upstream
git checkout development
git merge upstream/development
```

2.1.3 Working locally on a dedicated branch

Now that we have a fork, we can start making our changes to the source code. The best way to do it is to create a branch with a descriptive name to indicate what are you working on. Generally, your will branch off from the upstream *development* branch, which will contain the latest code.

For example, just for the sake of this guide, I'm going to work on issue #2.

```
git checkout development
git checkout -b fix-2
```

We can work in this branch, and make as many commits as we need to:

```
# hack hack hack
git commit -am "some decent commit message here"
```

Once we have finished working, we can push the branch online to our fork:

```
git push origin fix-2
```

We can then open a pull-request to merge our `fix-2` branch into `upstream/development`. If your code is not ready to be included, you can update the code on your branch and any more commits you add there will be added to the Pull Request. Members of the libNeuroML development team will then discuss your changes with you, perhaps suggest tweaks, and then merge it when ready.

2.1.4 Continuous integration

libNeuroML uses continuous integration (Wikipedia). Each commit to the master or development branches is tested, along with all commits to pull requests. The latest status of the continuous integration tests can be seen here on GitHub Actions.

2.1.5 Release process

libNeuroML is part of the official NeuroML release cycle. When a new libNeuroML release is ready the following needs to happen:

- Update version number in setup.py
- update version number in doc/conf.py
- update release number in doc/conf.py (same as version number)
- update changelog in README.md
- merge development branch with master (This should happen via pull request - do not do the merge yourself even if you are an owner of the repository.

- push latest release to PyPi

More information on the NeuroML release process can be found on the NeuroML documentation page.

2.2 Regenerating documentation

Please create a virtual environment and use the *requirements.txt* file to install the necessary bits.

In most cases, running *make html* should be sufficient to regenerate the documentation. However, if any changes to *nml.py* have been made, the *nml-core-docs.py* file in the *helpers* directory will also need to be run. This script manually adds each class from *nml.py* to the documentation as a sub-section using the *autoclass* sphinx directive instead of the *automodule* directive which does not allow us to do this.

2.3 Implementation of XML bindings for libNeuroML

The GenerateDS Python package is used to automatically generate the NeuroML XML-bindings in libNeuroML from the NeuroML Schema. This technique can be utilized for any XML Schema and is outlined in this section. The addition of helper methods and enforcement of correct naming conventions is also described. For more detail on how Python bindings for XML are generated, the reader is directed to the GenerateDS and libNeuroML documentation. In the following subsections it is assumed that all commands are executed in a top level directory nml and that GenerateDS is installed. It should be noted that enforcement of naming conventions and addition of helper methods are not required by GenerateDS and default values may be used.

2.3.1 Correct naming conventions

A module named generateds_config.py is placed in the nml directory. This module contains a Python dictionary called NameTable which maps the original names specified in the XML Schema to user-specified ones. The NameTable dictionary can be defined explicitly or generated programmatically, for example using regular expressions.

2.3.2 Addition of helper methods

Helper methods associated with a class can be added to a Python module as string objects. In the case of libNeuroML the module is called helper_methods.py. The precise implementation details are esoteric and the user is referred to the GenerateDS documentation for details of how this functionality is implemented.

2.3.3 Generation of bindings

Once generateds_config.py and a helper methods module are present in the nml directory a valid XML Schema is required by GenerateDS. The following command generates the nml.py module which contains the XML-bindings:

```
$ generateDS.py -o nml.py --use-getter-setter=none --user-methods=helper_methods NeuroML_
↪v2beta1.xsd
```

The -o flag sets the file which the module containing the bindings is to be written to. The –use-getter-setter=none option disables getters and setters for class attributes. The –user-methods flag indicates the name of the helper methods module (See section “Addition of helper methods”). The final parameter (NeuroML_v2beta1.xsd) is the name of the XML Schema used for generating the bindings.

2.4 Multicompartmental Python API Meeting

2.4.1 Organisation

Dates: 25 & 26 June 2012

Location: Room 336, Rockefeller building, UCL, London

Attendees: Sandra Berger, Andrew Davison, Padraig Gleeson, Mike Hull, Steve Marsh, Michele Mattioni, Eugenio Piasini, Mike Vella

Sponsors: This meeting was generously supported by the INCF Multi Scale Modelling Program.

2.4.2 Minutes

Agreeing on terminology (segments, etc.) & scope

A discussion on the definitions of the key terms Node, Segment and Section is here, and was the basis for discussions on these definitions at the meeting:

Nodes, Segments and Sections

Agreements

The Python libNeuroML API will use Node as a key building block for morphologies.

Segment is agreed on as the basis for defining morphologies in NeuroML and will be a top level object in libNeuroML, where it will be the part of a neurite between two Nodes (proximal & distal).

Segment Group will be the basis for the grouping of these, and will be used to define dendrites, axons, etc.

Section is a term for the cable-like building block in NEURON, and will not be formally used in NeuroML or libNeuroML.

There was a discussion on whether it would be useful to be able to include this concept "by the back door" to enable lossless import & export of morphologies from NEURON. Padraig's proposal was to add an attribute (e.g. primary) to the segmentGroup element to flag a core set of non overlapping segmentGroups, which are continuous (all children are connected to distal point of parent) which would correspond to the old "cable" concept in NeuroML v1.x.

There was much discussion on the usefulness of this concept and whether it should be a different element/object in the API from segmentGroup. The outcome was not fully resolved, but as a first test of this concept, Padraig will add the new attribute to NeuroML, Mike V will add a flag (boolean?) to the API, and at a later point, when the API begins to interact with native simulators, we can reevaluate the usefulness of the term.

Mike Vella's current implementation

This is under development at: https://github.com/NeuralEnsemble/libNeuroML/tree/master/neuroml

Mike will continue on this (almost) full time for the next 2 months.

Following the meeting, he will perform a refactoring operation on the code base to better reflect the names used in NeuroML, e.g.

```
neuroml_doc

    cells

        morphology # not entirely sure how this works- contains segment groups and is itself
        a segment group?
```

segments

segment_groups

segment_groups

biophysical_properties

notes

morphologies

networks

point currents

ion channels

synapses

extracellular properties

It was also decided that certain SegmentGroup names should have reserved names in libNeuroML, the exact implementation of this is undecided:

Segment groups with reserved names:

soma_group
axon_group
apical_dendrite_group
basal_dendrite_group

It was also decided that a segment should only be able to connect to the root of a morphology, the syntax should be something along the lines of:

segment can only connect to root of a morphology

connect syntax examples:

morph2.attach(2,cell2,0.5) (default frac along = None)

and:

morph[2].attach(cell2,0.5)

Mike V was asked to add a clone method to a morphology.

It was decided that fraction_along should be a property of segment.

The syntax for segment groups should be as follows: group=morph.segment_groups['axon_group'] (in connect merge groups should be false by default - throw an exception, tell the user setting merge_groups = True or rename group will fix this)

This was a subject of great debate and has not been completely settled.

Morphforge latest developments

Mike Hull gave a brief overview of the latest developments with Morphforge:

https://github.com/mikehulluk/morphforge

He pointed out that it's still undergoing refactoring, but it can be used by other interested parties, and there is detailed documentation online regarding installation, examples, etc.

Neuronvisio latest developments

Michele Mattioni gave a status update on Neuronvisio:

http://neuronvisio.org

The application has been closely linked to the NEURON simulator but hopefully use of libNeuroML will allow it to be used independently of NEURON.

Michele showed Neuronvisio's native HDF5 format as just one possible way to encode model structure + simulation results: https://github.com/NeuralEnsemble/libNeuroML/blob/master/hdf5Examples/Neuronvisio_medium_cell_example_10ms.h5

Current Python & NeuroML support in MOOSE

A Skype call/Google Hangout was held on Tues at 9:30 to get an update from Bangalore.

The slides from this discussion are here:

https://github.com/NeuralEnsemble/libNeuroML/blob/master/doc/2012_06_26_neuroml_with_pymoose.pdf

As outlined there there are a number of areas in which MOOSE and Moogli import/export NeuroML version 1.x. A number of issues and desired features missing in v1.x were highlighted, most of which are implemented or planned for NeuroML v2.0.

There was general enthusiasm about the libNeuroML project, and it was felt that MOOSE should eventually transition to using libNeuroML to import NeuroML models. This will happen in parallel with updating of the MOOSE PyNN implementation.

The MOOSE developers were also keen to see how the new ComponentTypes in NeuroML 2 will map to inbuilt objects in MOOSE (e.g. Integrate-and-Fire neurons, Markov channel, Izhikevich). They will add simple examples to the latest MOOSE code to demonstrate their current implementation and discussion can continue on the mailing lists.

Saving to & loading from XML

There was not any detailed discussion on the various strategies for reading/saving XML in Python.

Padraig's suggestion based on generateDS.py: https://github.com/NeuralEnsemble/libNeuroML/tree/master/ideas/padraig/generatedFromV2Schema produces a very big file, which while usable as an API, e.g. see:

https://github.com/NeuralEnsemble/libNeuroML/blob/master/hhExample/hh_NEUROML2.py

could do with a lot of refactoring. It was felt that a version of this with a very efficient description of morphologies (and network instances) based on the current work of Mike V is the way forward.

Storing simulation data as HDF5

The examples at: https://github.com/NeuralEnsemble/libNeuroML/tree/master/hdf5Examples have been updated.

The long term aim would be to arrive at a common format here that can be saved by simulators and that visualisation packages like Moogli and Neuronvisio can read and display. This may be based on Neo: http://packages.python.org/neo/, but that package's current lack of ability to deal with data with nonuniform time points (e.g. produced by variable time step simulations) may be a limiting factor.

General PyNN & NeuroML v2.0 interoperability

There was agreement that libNeuroML will form the basis of the multicompartmental neuron support in PyNN. The extra functionality needed to interact with simulators is currently termed "Pyramidal", but this will eventually be fully merged into PyNN.

http://neuralensemble.org/trac/PyNN http://www.neuroml.org/NeuroML2CoreTypes/PyNN.html http://www.neuroml.org/pynn.php

2.5 Nodes, Segments and Sections

An attempt to clarify these interrelated terms used in describing morphologies. Names in **bold type** are used for elements of the NeuroML object model.

2.5.1 Nodes

A node is a 3D point with diameter information which forms the basis for 3D morphological reconstructions.

These nodes (or points) are the fundamental building blocks in the SWC and Neurolucida formats. This method of description is based on the assumption that each node is physically connected to another node.

2.5.2 Segments

A **segment** (according to NeuroML v1&2) is a part of a neuronal tree between two 3D points with diameters (**proximal** & **distal**). The term node isn't used in NeuroML but the above description describes perfectly well the **proximal** & **distal** points. Cell **morphology** elements consist of lists of **segments** (each with unique integer id, and optional name).

All segments, apart from the root segment, have a **parent** segment. If the **proximal** point of the segment is not specified, the **distal** point of the parent segment is used for the **proximal** point of the child.

A special case is defined where **proximal** == **distal**, and the **segment** is assumed to be a sphere at that location with the specified diameter.

Segments can be grouped into **segmentGroups** in NeuroML v2.0. These can be used to specify "apical_dendrites", "axon_group", etc., which in turn can be used for placing channels on the cell.

An example of a NeuroML v2.0 cell is here.

libNeuroML will allow low level access to create and modify morphologies by handling nodes. Segments will also be top level objects in the API. The XML serialisation will only specify **segments** with **proximal** & **distal** points, but the HDF5 version may have an efficient serialisation of nodes & segments.

2.5.3 Sections

The concept of section is fundamentally important in NEURON. A section in this simulator is an unbranched cable which can have multiple 3D points outlining the structure of a neurite in 3D. These points are used to determine the surface area along the section. NEURON can vary the spatial discretisation of the neurite by varying the "nseg" value of the section, e.g. a section with 20 3D points and nseg =4 will be split into 4 parts of equal length for simulating (as isopotential compartments), with the surface area (and so total channel conductance) of each determined by the set of 3D points in that part.

There was a similar concept to this in NeuroML v1.x, the **cable**. Each **segment** had an attribute for the cable id, and these were used for mapping to and from NEURON. Cables were unbranched, and so all segments after the first in the cable only had distal points, see this example.

The cable concept was removed in NeuroML v2.0, as this is was seen as imposing concepts from compartmental modelling on the basic morphological descriptions of cells. There is only a **segmentGroup** element for grouping segments, though a **segment** can belong to multiple **segmentGroups**, which don't need to be unbranched (unlike **cables**). There may need to be a new attribute in **segmentGroup** (e.g. primary or unbranched or cable="true") which defines a nonoverlapping set of unbranched segmentGroups, which can be used as the basis for sections in any parsing application which is interested in them, or be ignored by any other application.

In libNeuroML, a section-like concept can be added at API level, to facilitate building cells, to facilitate import/export to/from simulators supporting this concept, and to serve as a basis for recompartmentalisation of cells.

2.5.4 Issues

Dendrites in space

One major issue to address is that in many neuronal reconstructions, the soma is not included (or perhaps just an outline of the soma is given), only the dendrites are. These dendrites' 3D start points are on the edge of the soma membrane "floating in space". Normal procedure for a modeller in this case is to create a spherical soma at this central point and electrically attach the dendrites to the centre of this.

In this case (and many others) the physical location of the start of the child segments do not correspond to the electrical (or logical) connection point on the parent. This has advantages and disadvantages:

(+) It allows the real 3D points of the neuronal reconstruction to be retained (useful for visualisation)

(-) This is not unambiguously captured in the simplest morphological formats like SWC, which assume physical connectivity between nodes/points

This scenario is supported in NeuroML v1&2, where a child **segment** has the option to redefine its start point (by adding a **proximal**) with the child <-> parent relationship defining the electrical connection. This allows lossless import & export from NEURON and removes the ambiguity of more compact formats like SWC and Neurolucida.

Connections mid segment

Another option for electrical connections (also influences by NEURON sections) is the ability for **segments** to (electrically/logically) connect to a point inside a **segment**. This is specified by adding a fractionAlong attribute to the **parent** element, i.e.

```
<parent segment="2" fractionAlong="0.5"/>
```

This is not possible in a node based format, but represents a logically consistent description of what the modeller wants.

What to do?

Two options are available then for a serialisation format or API: should it try to support all of these scenarios, or try to enforce "best practice"?

PG: I'd argue for the first approach, as it retains as much as possible of what the original reconstructor/simulator specified. An API which enforces a policy when it encounters a non optimal morphology (e.g. moving all dendrites to connection points, inserting new nodes) will alter the original data in perhaps unintended ways, and that information will be lost by subsequent readers. It should be up to each parsing application to decide what to do with the extra information when it reads in a file.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[VCC+14] Michael Vella, Robert C. Cannon, Sharon Crook, Andrew P. Davison, Gautham Ganapathy, Hugh P. C. Robinson, R. Angus Silver, and Padraig Gleeson. Libneuroml and pylems: using python to combine procedural and declarative modeling approaches in computational neuroscience. *Frontiers in neuroinformatics*, 8:38, 2014. doi:10.3389/fninf.2014.00038.

PYTHON MODULE INDEX

n

INDEX

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W